

[диск с материалами чтобы не смотреть в ТИМС](#)  
[шпаргалка по паттернам](#)

## 1. Понятие архитектуры, профессия «Архитектор».

[конспект преза](#)

### Понятие архитектуры

Общепринятое определения и даже понимания понятия «Архитектура» не существует. На интуитивном уровне **архитектура** — это набор важнейших решений об организации программной системы:

- того, из каких компонентов она состоит,
- как эти компоненты взаимосвязаны друг с другом и с окружением,
- какие ограничения действуют на сами компоненты и взаимосвязи между ними,
- мотивация принятых решений.

В отличие от архитектуры здания, архитектура программной системы меняется и эволюционирует вместе с самой системой.

Архитектура у системы есть всегда, даже если о ней никто не думал и никак её не описывал, просто она может оказаться не очень хорошей.

### Для чего тратить усилия на явное описание архитектуры системы?

- **Продуманная архитектура может в разы сократить затраты на кодирование.**  
С создания первого варианта архитектуры начинается реализация системы, это та деятельность, которая даёт понять, что и примерно как нужно писать.
- **Архитектура — это инструмент управления проектом.**  
Оценить проект невозможно без декомпозиции задачи на подзадачи и выделения основных компонентов, крупнозернистая декомпозиция может определить состав команды и даже структуру организации, разрабатывающей систему. И по ходу выполнения проекта надо иметь чёткое представление о компонентах системы и связях между ними, чтобы понять, на каком этапе разработки мы сейчас находимся.
- **Архитектура — это средство обеспечения переиспользования, как третьесторонних компонентов, так и внутри системы (или группы систем).**  
Именно архитектура позволяет выделить возможные кандидаты на переиспользование, определить требования к третьесторонним компонентам, определить критерии выбора, выделить и инкапсулировать функциональность своих компонентов, чтобы сделать их переиспользуемыми.
- **Явное описание архитектуры системы позволяет делать выводы о некоторых её качествах ещё до того, как написана первая строчка кода.**

Формальные средства описания архитектуры позволяют иногда получать не только качественные, но и количественные характеристики системы, например, оценивать ожидаемую производительность или требуемую память.

Стоит заметить, что заказчику архитектура системы совершенно не интересна.

## Профессия “Архитектор”

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

Таким образом, архитектор — это ключевой технический специалист в команде.

Часть знаний и умений архитектора может прийти только с опытом, поэтому обычно будущему архитектору сначала приходится поработать на позиции разработчика.

**Профессиональный стандарт определяет цель деятельности архитектора так:**

**Создание и сопровождение архитектуры программных средств, заключающейся**

- в синтезе и документировании решений о структуре;
- компонентном устройстве;
- основных показателях назначения;
- порядке и способах реализации программных средств в рамках системной архитектуры;
- реализации требований к программным средствам;
- контроле реализации и ревизии решений

**Некоторые трудовые функции архитектора согласно этому же стандарту:**

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

**Архитектор vs разработчик**



Рис. 3: Знания архитектора и знания разработчика

- Широта знаний

Хороший архитектор должен обладать большим кругозором и уметь быстро погружаться в предметную область, в которой разрабатывается проект — в отличие от программиста, которому обычно достаточно разбираться в той технологии, которую он использует для реализации.

- Коммуникационные навыки

На архитектора чаще всего возлагают обязанности по коммуникации между разработчиками и руководством, часто архитектор принимает участие в общении с заказчиком, поэтому для архитектора важны не только технические, но и социальные навыки.

- Часто архитектор играет роль разработчика и наоборот
  - Архитектор в «башне из слоновой кости» — это плохо

Если архитектор вообще не принимает участия в разработке и только говорит разработчикам, что делать, он быстро теряет связь с кодом и утрачивает понимание реализации.

## 2. Архитектурные виды.

### конспект преза

Поскольку программные системы, как правило, очень сложны, их архитектура разделяется на части, каждая из которых описывает систему с какой-то определённой точки зрения. Эти части называются **архитектурными видами (architectural view)**.

Каждый вид может состоять из нескольких описаний (неформальных, полуформальных или на каком-нибудь формальном языке), и эти описания могут различаться по уровню детализации.

Совокупность видов, рассматривающая систему единообразно, называется **архитектурной точкой зрения (architectural viewpoint)**.

Стандарт IEEE 1016-2009 выделяет аж 12 точек зрения на систему:

# красное - что используется для описания данного вида

- **Контекст** — описывает, что система должна делать, фиксирует окружение системы
  - **Диаграммы активностей UML, IDEF0 (SADT)**
  - Состоит из сервисов и акторов, которые могут быть связаны информационными потоками
  - Система представляет собой «чёрный ящик»
  - Корень иерархии уточняющих дизайн системы видов, стартовая точка при проектировании системы
- **Композиция** — описывает крупные части системы и их предназначение
  - **Диаграммы компонентов UML, IDEF0**
  - Предназначена для локализации и распределения функциональности системы по её структурным элементам, облегчения переиспользования, оценки, планирования, управления проектом, определения
- **Логическая структура** — структура системы в терминах классов, интерфейсов и отношений между ними
  - **Диаграммы классов UML, диаграммы объектов UML**
- **Зависимости** — определяет связи по данным между элементами
  - **Диаграммы компонентов UML, диаграммы пакетов UML**
  - Необходим для анализа изменений, идентификации узких мест производительности, планирования, интеграционного тестирования.
- **Информационная структура** — определяет персистентные данные в системе
  - **Диаграммы классов UML, IDEF1x, ER, ORM**
- **Использование шаблонов** — документирование использования локальных паттернов проектирования
  - **Диаграммы классов UML, диаграммы пакетов UML, диаграммы коллaborаций UML**
- **Интерфейсы** — специфицирует информацию о внешних и внутренних интерфейсах
  - **IDL, диаграммы компонентов UML, макеты пользовательского интерфейса, неформальные описания сценариев использования**
- **Структура системы** — рекурсивное описание внутренней структуры компонентов системы
  - **Диаграммы композитных структур UML, диаграммы классов UML, диаграммы пакетов UML**
- **Взаимодействия** — описывает взаимодействие между сущностями

- **Диаграммы композитных структур UML, диаграммы взаимодействия UML, диаграммы последовательностей UML**
- **Динамика состояний** — описание состояний и правил переходов между состояниями
  - **Диаграммы конечных автоматов UML, диаграммы Харела, сети Петри**
- **Алгоритмы** — описывает в деталях поведение каждой сущности
  - **Диаграммы активностей UML, псевдокод, настоящие языки программирования**
- **Ресурсы** — описывает использование внешних ресурсов
  - **Диаграммы развёртывания UML, диаграммы классов UML, OCL**

## Комментарии

- Ни один вид не обязательен
  - Стандарт требует только общие сведения о системе (назначение, границы системы, контекст, в котором система существует) и те виды, которые архитектор считает важными
- Активно используются визуальные языки
  - В основном как дополнение к тексту
- Моделирование принципиально важно для архитектуры
  - ПО само по себе очень сложно, а модели упрощают сложное, оставляя существенное
  - Сложность ПО существенна для архитектуры, поэтому от нее нельзя абстрагироваться
  - Но можно декомпозировать систему на виды (тем самым декомпозируя сложность)

## 3. Роль архитектуры в жизненном цикле ПО.

### конспект преза

Есть распространённое заблуждение, что работа архитектора напрямую связана с фазой проектирования жизненного цикла ПО.

Но современная точка зрения на архитектуру больше склоняется к тому, что **архитектура - это непрерывная активность, находящаяся в центре процесса разработки**.

Создание, поддержка и эволюция архитектуры — это деятельности, необходимые на всех этапах жизненного цикла (как контроль версий или управление проектом).

## Архитектура и требования

Сбор и анализ требований — это архитектурная активность, в этот момент рисуются первые модели системы, пишутся первые документы, которые лягут в основу будущей архитектуры, а затем и реализации.

Напомним типы требований:

- **Функциональные** — то, что система должна делать.
- **Нефункциональные** — то, как система должна это делать.

- Требования на эффективность по времени работы и по используемым ресурсам.
  - Требования на масштабируемость — насколько адекватно система справляется с ростом нагрузки.
  - Требования на удобство использования — насколько легко учиться пользоваться системой и насколько удобно собственно пользоваться ею.
  - Требования на надёжность.
    - Время наработки на отказ — сколько времени в среднем система работает без отказов.
    - Время восстановления после сбоя — за сколько времени систему можно восстановить, если она всё-таки отказалась.
    - Корректность поведения при сбое (failsafe) — есть системы, в которых даже критическая ошибка должна приводить к корректному завершению работы  
(классический пример с ядерным реактором и с банковской системой).
  - Требования на безопасность.
  - Требования на сопровождаемость и расширяемость.
  - ...
- **Ограничения**
- Технические — например, если вы программируете под iPhone, то ограничены в выборе инструментария.
  - Бизнес-ограничения — например, система должна быть сделана за месяц, иначе на уже назначенной пресс-конференции будет нечего показать.

Наибольшее влияние на архитектуру оказывают нефункциональные требования: архитектурный стиль выбирается исходя из приоритетов нефункциональных требований, и архитектура проектируется в большой степени с их учётом.

Это связано с тем, что написать работающую программу, в общем-то, не сложно, сложно сделать так, чтобы она была ещё и быстрой, сопровождаемой, надёжной и т.д.

## Архитектура и проектирование

На стадии проектирования архитектурные вопросы оказываются в центре внимания.

Сейчас мало где используется водопадная модель разработки, поэтому стадию проектирования проекты чаще всего проходят несколько раз (например, в начале каждой итерации в agile-методологиях).

На фазе проектирования:

- Выполняется декомпозиция задачи
- Определяются границы компонентов
- Определяются интерфейсы и протоколы общения
- Решаются общие для всей системы вопросы:
  - Стратегия обработки ошибок,
  - Стратегия логирования,
  - Стратегия выкатывания обновлений

- Стратегия разделения доступа
- Вопросы локализации
- ...
- Проводится анализ и верификация архитектуры

У каждой системы есть:

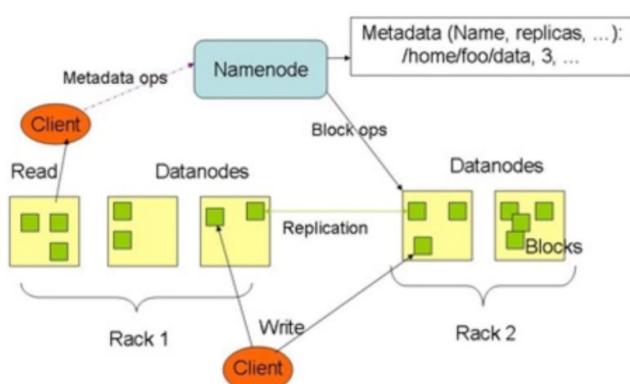
- **предписывающая архитектура (prescriptive architecture)** — архитектура, как её определил архитектор
- **описывающая архитектура (descriptive architecture)** — архитектура, как она есть в системе
  - Архитектура у ПО есть всегда, как вес у камня

Архитектура может быть подвержена двум видам проблем:

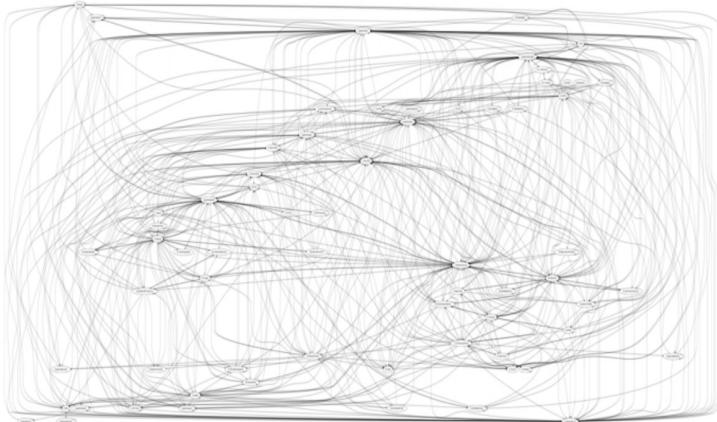
- **architectural drift** — «сползание» фактической архитектуры
  - **появление в ней важных решений, которых нет в описательной архитектуре**
  - вносит хаос в систему
- **architectural erosion** — «размывание» архитектуры
  - **отклонения от описательной архитектуры, нарушения ограничений**
  - возникает на практике очень часто в процессе багфиксов, рефакторингов, добавления новой функциональности
  - вносит еще больше хаоса в систему
  - антипаттерн «Big ball of mud» — результат эрозии

Хороший пример архитектурной эрозии и того, во что красивая на бумаге архитектура превращается при реализации — архитектура системы Apache Hadoop.

Проектировали все красиво аккуратно



Получилось страшна вырубай



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

[конспект преза](#)

### Сложность

Сложность является неотъемлемой частью программного обеспечения, и если попытаться от неё избавиться, разрабатываемое ПО потеряет и свою ценность.

- **Существенная сложность (essential complexity)** — сложность, присущая решаемой проблеме; ею можно управлять, но от неё нельзя избавиться

Например, если мы решаем дифференциальное уравнение, вряд ли наш код может быть проще, чем математический метод его решения.

- **Случайная сложность (accidental complexity)** — сложность, привнесённая способом решения проблемы

От случайной сложности можно и нужно избавляться.

Деятельность архитектора направлена не на борьбу со сложностью, а на управление ею. И основной приём управления сложностью — это её сокрытие.

### Свойства сложных систем

Не все сложные системы обладают этими характеристиками, но ими обладают те системы, о которых мы можем хоть как-то рассуждать.

- Иерархичность — свойство системы состоять из иерархии подсистем или компонентов

- применяется **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
- Наличие относительно небольшого количества видов компонентов, экземпляры которых сложно связаны друг с другом
  - применяется **абстрагирование** — выделение общих свойств компонентов, их классификация, рассуждение не о конкретных элементах системы, а об их типах
- Сложная система, как правило, является результатом эволюции простой системы
- Сложность вполне может превосходить человеческие интеллектуальные возможности

## Подходы к декомпозиции

- **Восходящее проектирование**
  - Сначала создаём отдельные компоненты, потом собираем из них всё более сложные системы
  - Такой подход целесообразно применять в исследовательских проектах, когда непонятно, что в конечном итоге может получиться, либо в случае, когда сама предметная область содержит небольшие обособленные задачи, которые можно по-разному комбинировать.
  - Так имеет смысл проектировать библиотеки — рассматривать их как набор кирпичиков, из которых пользователь может сам сложить всё, что ему нужно.
- **Нисходящее проектирование**
  - Сначала рассматриваем задачу целиком, разделяем её на подзадачи
  - Строго задаем интерфейсы для модулей, решающих эти подзадачи
  - Используем “заглушки” для еще не реализованных модулей
  - Постепенно реализуем модули
  - Такой подход используется, когда есть чёткое видение конечного результата и надо минимальными усилиями этого результата достичь (то есть, на самом деле, почти всегда).

## Модули

**Модуль** - структурная единица кода, соответствующая подзадачам, на которые разбита система.

### Примеры

- В объектно-ориентированных языках: классы, компоненты
- В функциональных языках: функции или какие-либо способы их группировки
- В структурных языках: пара из .h и .c-файлов в C, модули в Паскале, пакеты в Аде и т.д.

Модули характеризуются своим **интерфейсом и реализацией**.

Для модулей есть следующие общепринятые правила их проектирования:

- Четкая декомпозиция
  - каждый модуль занимает своё место в системе, знает, какую задачу он решает, должно быть понятно, как им пользоваться;
- Минимизация интерфейса
  - использование модуля должно быть максимально простым, но всё ещё позволять решать задачу;
- Один модуль — одна функциональность
  - если модуль делает что-то и что-то ещё, разбейте этот модуль на два;
- Отсутствие побочных эффектов
  - очень желательное, но не всегда достижимое свойство
- Независимость от других модулей
  - модуль может использовать другие модули для своей реализации, но он не может знать о деталях реализации других модулей и как-то рассчитывать на эти детали;
- Принцип сокрытия данных
  - внутреннее представление всех данных модуля должно быть известно только ему самому, наружу могут быть видны только абстрактные типы данных, манипулировать которыми можно только через функции модуля.

## Модульность

Модульность - способ разбиения системы на компоненты.

- Потенциально позволяет создавать сколь угодно сложные системы
- Строгое определение контрактов позволяет разрабатывать независимо
- Необходим баланс между количеством и размером модулей
  - если модули слишком большие, то сложность каждого модуля неоправданно велика, и, соответственно, неоправданно велики затраты на его разработку.
  - если модули слишком маленькие, то неоправданно велики затраты на их интеграцию — модулей становится слишком много, взаимодействия между модулями становятся слишком сложны.

Есть знаменитое правило “7+2”: человек может одновременно удерживать в голове порядка 7 сущностей (плюс-минус, в зависимости от индивидуальных способностей). Это правило может быть хорошей отправной точкой при проектировании системы и разбиении на модули.

## Сопряжение и связность

Численные метрики, которые можно посчитать автоматически; хороший показатель качества разбиения на модули.

**Сопряжение (Coupling)** — мера того, насколько взаимосвязаны разные модули в программе (то есть насколько часто один модуль дёргает другие, насколько много этих других и насколько много они должны знать друг о друге).

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

**Цель:** слабое сопряжение и сильная связность

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

[конспект преза](#)

### Объекты и классы

Было много определений, больше всего подходит:

Each object looks quite a bit like a little computer — it has a state, and it has operations that you can ask it to perform — [Thinking in Java](#)

Иными словами, объекты имеют три важных свойства:

- **состояние**
  - **инвариант** — набор логических условий, которые должны выполняться всё время жизни объекта.
  - инвариант позволяет реализовывать методы будучи уверенными в том, что эти условия выполнены
  - каждый объект сам отвечает за поддержание своего инварианта и обязан не давать возможности нарушить его извне (поэтому public- поля запрещены, например).
- **поведение**
  - отличается от вызова функции модуля тем, что объект вправе сам решать, как обработать пришедший к нему запрос, так что правильнее всего считать, что объекты не вызывают методы друг друга, а отправляют друг другу сообщения
- **идентичность.**

**Класс** — это тип объекта. Классы определяют структуру данных, которые хранит объект, и методы (с их реализацией, обратите внимание), которые есть у каждого объекта этого класса.

Класс — это сущность времени компиляции (и именно с классами в основном работают программисты), объект — сущность времени выполнения (с ними программисты работают, например, отлаживая систему).

## Абстракция

**Абстракция** выделяет существенные характеристики объекта, отличающие его от остальных объектов, с точки зрения наблюдателя.

Абстракция касается не только ООП, но и всего остального — структурного, функционального программирования, алгебры и вообще способности людей к мышлению.

Один и тот же физический объект может обладать несколькими разными абстракциями одновременно.

Наличие хорошей абстракции в разы упрощает работу с системой.

Абстракция может иметь несколько взаимозаменяемых реализаций, и если абстракция достаточно содержательна, на ней могут базироваться реализации других абстракций.

Хороший пример использования абстракций — вся математика.

## Инкапсуляция

**Инкапсуляция** разделяет интерфейс (контракты) абстракции и её реализацию.

- Реализует принцип “меньше знаешь — крепче спиши”, позволяя пользователю не знать про реализацию вообще.
- Защищает инварианты абстракции от их порчи извне.

## Наследование и композиция

**Наследование** — один из способов реализации сабтайпинга в системе типов.

*“Объект типа-потомка является одновременно объектом типа-предка, поэтому может использоваться везде, где может использоваться предок”.*

- Отношение “Является” (is-a)
- Способ абстрагирования и классификации
- Средство обеспечения полиморфизма
- Потомок наследует все инварианты предка и не вправе их нарушать

**Композиция** — нынче модно использовать для переиспользования кода вместо наследования.

*“Общая функциональность выносится в отдельный класс и все, кому она нужна, просто включают объект этого класса себе как поле (если повезёт, класс окажется статическим, тогда даже поле не нужно).*

- Отношение “Имеет” (has-a)
- Способ создания динамических связей
- Средство обеспечения делегирования

Сравним между собой:

- Наследование и композиция более-менее взаимозаменяемы
  - так как объект-потомок на самом деле включает в себя объект-предок
- Можно переделать одно в другое

- Композиция предпочтительнее наследования
  - наследование фиксировано во время компиляции
  - композицию можно менять во время выполнения, что даёт большую гибкость

## 6. Принципы выделения абстракций предметной области.

### конспект преза

Допустим, у нас есть техническое задание, требующее разработать приложение для какой-то (возможно, незнакомой нам) предметной области.

Строим объектную модели предметной области и решаемой задачи

1. Определение объектов и их атрибутов
  - слушаем экспертов и собираем информацию о предметной области, классы — это существительные;
2. Определение действий, которые могут быть выполнены над каждым объектом (назначение ответственности)
  - слушаем экспертов и собираем информацию о предметной области, действия — это глаголы;
3. Определение связей между объектами
  - задаем вопросы в духе “а про это кто знает”, “а это где используется” и т.д.
4. Определение интерфейса каждого объекта
  - фиксируем и формализуем всё, что удалось сделать на предыдущих этапах

Результатом этой деятельности будет первый грубый набросок архитектуры системы, выражаемый обычно в виде диаграммы классов:

**Классы, конечно, появляются не только из предметной области, но и могут появляться в процессе реализации. Часто встречающиеся источники абстракций таковы:**

### Изоляция сложности

- Сложные алгоритмы могут быть инкапсулированы
- Сложные структуры данных — тоже
- И даже сложные подсистемы
- Делается это для того, чтобы:
  - спрятать сложность от остальной системы
  - иметь возможность менять алгоритмы, оптимизировать внутреннее представление и т.д., зная, что это ничего не сломает
- Надо внимательно следить за интерфейсами абстракций, скрывающих сложность
  - они имеют тенденцию сами становиться слишком сложными

### Изоляция возможных изменений

Потенциальные изменения могут быть инкапсулированы в отдельный класс или подсистему.

*Однако не надо переусердствовать, хорошая архитектура позволяет себя менять и без специальной поддержки изменчивости.*

- Источники изменений
  - Бизнес-правила
    - алгоритмы из предметной области, которые реализует программа
  - Зависимости от оборудования и операционной системы
    - меняются редко, но изменение может быть критично для жизни программного продукта
  - Ввод-вывод
    - формат файлов сохранения, так и формат сетевых пакетов, набор допустимых команд и т.д.
  - Нестандартные возможности языка
    - система должна иметь возможность пережить смену компилятора
  - Сложные аспекты проектирования и конструирования
    - сложность полезно прятать, но не только сложность реализации, но и сложность архитектуры
  - Третьесторонние компоненты
    - нельзя критически зависеть от чего-то, что перестанет поддерживаться через полгода или станет стоить бешеных денег
  - ...

## Изоляция служебной функциональности

Служебная функциональность может быть инкапсулирована.

Классы, отвечающие за бизнес-логику, должны содержать только бизнес-логику, не захламляя её деталями реализации, детали должны жить отдельно.

Примеры:

- Репозитории
- Фабрики
- Диспетчеры, медиаторы
- Статические классы (Сервисы)
- ...

## 7. Принципы SOLID.

**обязательно знать...**

[конспект преза](#)

Есть пять базовых принципов объектно-ориентированного проектирования, известные как принципы SOLID.

- **Single responsibility principle (принцип единственности ответственности)**
  - Каждый класс должен делать что-то одно.
  - Ответственность класса была полностью инкапсулирована в этом классе.
- **Open/closed principle (принцип открытости/закрытости)**

- Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения.
- **Liskov substitution principle (принцип подстановки Барбары Лисков)**
  - Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.
- **Interface segregation principle (принцип разделения интерфейсов)**
  - Клиенты не должны зависеть от методов, которые они не используют.
- **Dependency inversion principle (принцип инверсии зависимостей)**
  - Модули верхних уровней не должны зависеть от модулей нижних уровней, оба типа модулей должны зависеть от абстракций.

Подробнее:

### Single responsibility principle (принцип единственности ответственности)

Каждый класс должен делать что-то одно.

Кажется привлекательным иметь “швейцарский нож”, который бы один решал все возможные проблемы, но такой класс, во-первых, тяжёл в сопровождении, а во-вторых, сложно понять его роль в системе, сложно объяснить её новым людям в проекте.

Ответственность класса была полностью инкапсулирована в этом классе.

Если у вас есть “утилиты для работы со строками” и “ещё утилиты для работы со строками”, дела ваши плохи — вы никогда в жизни не запомните где нужный вам метод. Это всё касается не только классов, но и функций, и целых подсистем.

### Open/closed principle (принцип открытости/закрытости)

Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения.

То есть, если интерфейс уже стабилизировался и им начали пользоваться, менять его нельзя.

Если надо добавить в абстракцию новую функциональность, можно использовать наследование.

Если это правило не соблюдать, интерфейс абстракции начнёт увеличиваться в размерах, сам собой начнёт нарушаться принцип единственности ответственности, и в результате мы получим несопровождаемого и неузабельного монстра (см. антипаттерны “God object” и “Swiss army knife” далее в этом курсе).

### 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 выполнять все инварианты классов-предков, явные или неявные.



Interface segregation principle (принцип разделения интерфейсов)

Клиенты не должны зависеть от методов, которые они не используют.

Слишком “толстые” интерфейсы необходимо разделять на более мелкие и специфические.

Группировать в интерфейсы надо методы, которые реально кластеризуются по смыслу, а не чисто механически.

Dependency inversion principle (принцип инверсии зависимостей)

Модули верхних уровней не должны зависеть от модулей нижних уровней, оба типа модулей должны зависеть от абстракций.

## 8. Закон Деметры. Принципы хорошего объектно-ориентированного кода.

[конспект преза](#)

### Закон Деметры

Еще одно хорошее правило.

**Неформально:** “Не разговаривай с незнакомцами”.

**Формально:** 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.

**Смысл:** Закон Деметры говорит просто, что не надо выставлять напоказ структуру данных, которая не является частью абстракции.

### Пример

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

### Почему

Вызовы “по цепочке” раскрывают внутреннюю структуру данных класса и не дают её потом изменить.

**Главное не переусердствовать.**

Код вида book.firstPageText(), book.secondPageText(), book.thirdPageText() и т.д. гораздо хуже, чем “паровозик”.

### Некоторые принципы хорошего кода

Краткий обзор книжек Р. Мартина “Чистый код” и “Code Complete” С. Макконелла.

### Абстракция

- Про каждый класс знайте, реализацией какой абстракции он является
  - Принцип единственности ответственности имеет очень простой критерий — если про класс можно одним коротким предложением сказать, что он такое, то всё ок.
  - Может потребоваться разделить класс на несколько разных классов просто потому, что методы по смыслу слабо связаны, и это будет не овердизайн.
- Учитывайте противоположные методы (add/remove, on/off, ...)

- Даже если они сейчас не нужны, когда-то кому-то неизбежно понадобятся.
- Разделяйте команды и запросы, избегайте побочных эффектов
  - Команда только меняет состояние, не возвращая результат, запрос только возвращает результат, не меняя состояния
  - Хорошо для понимания программы
  - В многопоточной программе синхронизации требуют только команды. Сколько угодно запросов может исполняться параллельно.
- Не возвращайте null
- По возможности делайте некорректные состояния невыразимыми в системе типов
- Комментарии в духе “не пользуйтесь объектом, не вызвав init()” можно заменить конструктором
  - Семантику языка тоже надо заставить работать на себя в плане чистоты абстракций.
- При рефакторинге надо следить, чтобы интерфейсы не деградировали

## Инкапсуляция

- Принцип минимизации доступности методов
  - Чем меньше интерфейс объекта, тем проще им пользоваться и тем проще уследить за инвариантами объекта.
- Паблик-полей не бывает
  - Паблик-поля вообще не могут поддерживать никаких инвариантов
- Protected- и package- полей тоже не бывает
  - На самом деле, у класса два интерфейса — для внешних объектов и для потомков (может быть отдельно третий, для классов внутри пакета, но это может быть плохо)
- Класс не должен ничего знать о своих клиентах
- Лёгкость чтения кода важнее, чем удобство его написания
- Опасайтесь семантических нарушений инкапсуляции
  - “Не будем вызывать ConnectToDB(), потому что GetRow() сам его вызовет, если соединение не установлено” — это программирование сквозь интерфейс

## Наследование и полиморфизм

- Агрегация/композиция/включение лучше
  - Переконфигурируемо во время выполнения
  - Более гибко
  - Иногда более естественно
- Наследование — отношение “является”, закрытого наследования не бывает
  - принцип подстановки Барбары Лисков: любой код, который может работать с предком, должен мочь работать и с потомком.
- Наследование — это наследование интерфейса (полиморфизм подтипов, subtyping)
- Хороший тон — явно запрещать наследование (final- или sealed-классы)
- Не вводите новых методов с такими же именами, как у родителя
- Code smells:
  - Базовый класс, у которого только один потомок
  - Пустые переопределения

- предок декларирует контракты, которые потомок не может содержательно выполнить
  - Очень много уровней в иерархии наследования
  - Новые методы с такими же именами, как у родителей

## Вопросы инициализации (конструкторы)

- Инициализируйте все поля, которые надо инициализировать
  - После конструктора должны выполняться все инварианты
- НЕ вызывайте виртуальные методы из конструктора
- private-конструкторы для объектов, которые не должны быть созданы (или одиночек)
- protected-конструкторы для абстрактных классов
- Одиночек надо использовать с большой осторожностью,
- Deep copy предпочтительнее Shallow copy
  - Хотя второе может быть эффективнее

## Мутабельность

Мутабельность — способность объекта изменять своё состояние.

Правила хорошего тона говорят, что немутабельным должно быть всё, что может быть немутабельным без существенного вреда смыслу или скорости работы программы.

Тем не менее, основные современные языки программирования предполагают, что состояние мутабельно по умолчанию.

Вот про что надо помнить, делая немутабельный класс.

- Не предоставлять методы, модифицирующие состояние.
- Заменить модифицирующие состояние методы на методы, возвращающие копию объекта
- Не разрешать наследоваться от класса.
- Сделать все поля константными.
- Не давать никому ссылок на поля мутабельных типов.
  - Даже если поля константны, константное поле мутабельного ссылочного типа не позволяет только модифицировать саму ссылку, объект, на который она указывает — сколько угодно.

## Про оптимизацию

*Почти всегда не надо заниматься оптимизацией © - William A. Wulf,  
Donald E. Knuth,  
M. A. Jackson*

- Надо себя сдерживать и стараться предпочитать простое решение быстро работающему

- (ну, до разумных пределов, всё-таки выбирайте алгоритм с подходящей асимптотикой, не надо числа Фибоначчи рекурсивно считать).
- Тонкой настройкой производительности (типа управления сборкой мусора и т.п.) надо заниматься только после исследования системы профилятором.

Впрочем, есть противоположное оптимизации понятие — “пессимизация”, означающее откровенную глупость, приводящую к замедлению программы (типа случайного boxing-a внутри цикла в Java), этого тоже надо избегать.

## Общие рекомендации

- Fail Fast
  - Программа должна завершить работу при малейшем подозрении на ошибку, а программист должен прикладывать усилия при написании кода, чтобы обнаружить потенциальную ошибку как можно раньше.
  - Не доверяйте параметрам, переданным извне (“Защитное программирование”)
  - assert-ы – чем больше, тем лучше
- Документируйте все открытые элементы API
  - И заодно всё остальное, для тех, кто будет это сопровождать
  - Подробно: в том числе предусловия и постусловия, исключения, потокобезопасность
- Статические проверки и статический анализ лучше, чем проверки в рантайме
  - Используйте систему типов по максимуму
- Юнит-тесты
- Continious Integration
- Не надо бояться всё переписать

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

[конспект преза](#)

### Модели в общем

**Модель** — это упрощённое подобие некоторого объекта или явления.

Примеры:

- математические модели
- модели как реальные объекты (например, модели самолётов)
- модели в разработке ПО

Свойства:

- Содержат меньше информации, чем реальность
- Существуют для определённой цели

- Модели субъективны, что позволяет отделить существенные свойства от несущественных
- Модели ограничены

All models are wrong, some are useful

## Модели в проектировании ПО

- Предназначены прежде всего для управления сложностью
  - дают возможность взглянуть на систему с нескольких разных точек зрения, выделяя каждый раз разные существенные моменты
- Могут моделировать как саму систему, так и окружение
  - автоматизируемый бизнес-процесс или внешние системы или пользователи или etc.
- Позволяют понять, проанализировать и протестировать систему до её реализации
- Используемые нотации и способы моделирования зависят от целей моделирования
  - От неформальных набросков до исполнимых моделей

## Архитектурные модели

**Архитектура** — это набор основных решений, принятых для данной системы.

**Архитектурная модель** — это некоторый артефакт, который отражает некоторые или все эти решения.

**Архитектурное моделирование** — это процесс уточнения и документирования этих решений

**Нотация архитектурного моделирования** — это язык или другое средство описания архитектурных решений

Моделирование непосредственно связано с используемой нотацией.

При моделировании архитектуры ПО надо определиться с тем:

- какие архитектурные решения нуждаются в моделировании;
- на каком уровне детализации;
- насколько формально.

Необходимо учитывать соотношение трудозатрат и выгоды - стоимость создания и поддержания модели не должна быть больше преимуществ от её использования.

## Возможные преимущества моделей

- Инструмент, направляющий и облегчающий проектирование
- Средство коммуникации между разработчиками
- Наглядный инструмент для общения с заказчиком
- Средство документирования и фиксации принятых решений
- Исходник для генерации кода?
  - бывают очень формальные диаграммы, по которым можно генерировать работающее приложение
  - чаще так делать нет смысла, потому что модель содержит принципиально меньше информации, чем надо для того, чтобы её можно было выполнить.

## Виды моделей

### 1. Естественные языки

Обычный текст — вполне себе инструмент моделирования

- + Очень выразителен, не требует специальных знаний, максимально гибок
- Неоднозначен, неформален, не строг, слишком многословен, бесполезен для автоматической обработки

### 2. Неформальные графические модели

Диаграммы, рисуемые в PowerPoint, InkScape и подобном

- + Могут быть красивыми, как правило, простые, очень гибкая нотация
- Неформальны, неоднозначны, не строги
- Но часто воспринимаются наоборот
- Практически бесполезны для автоматической обработки

### 3. Формальные графические модели (UML и SysML и другие)

Несколько слабо связанных нотаций (“диаграмм”)

- + Формальность языков, поддерживают много точек зрения, общеприняты, широкая поддержка инструментами
- Нет строгой семантики, сложно обеспечить консистентность, сложно расширять

### 4. Формальные текстовые языки (AADL и другие)

Это языки, где компоненты системы и интерфейсы между компонентами описываются формально на языке программирования (но без реализации, конечно), записываются и доказываются различные утверждения про моделируемую систему.

- + Хороши для моделирования встроенных систем и систем реального времени
- + Описывают одновременно “железо” и “софт”, продвинутые инструменты анализа
- Слишком многословны и детальны, сложны в изучении и использовании

## Еще понятия

- **Метафора визуализации** — договорённость о том, как будут представляться сущности языка
- **Точка зрения моделирования** — какой аспект системы и для кого моделируется
- Бывают одноразовые модели, документация и графические исходники
  - **Семантический разрыв** — неспособность модели полностью специфицировать систему

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

[конспект преза](#)

**Unified Modeling Language (UML)** - самый популярный визуальный язык, используемый для разработки программного обеспечения.

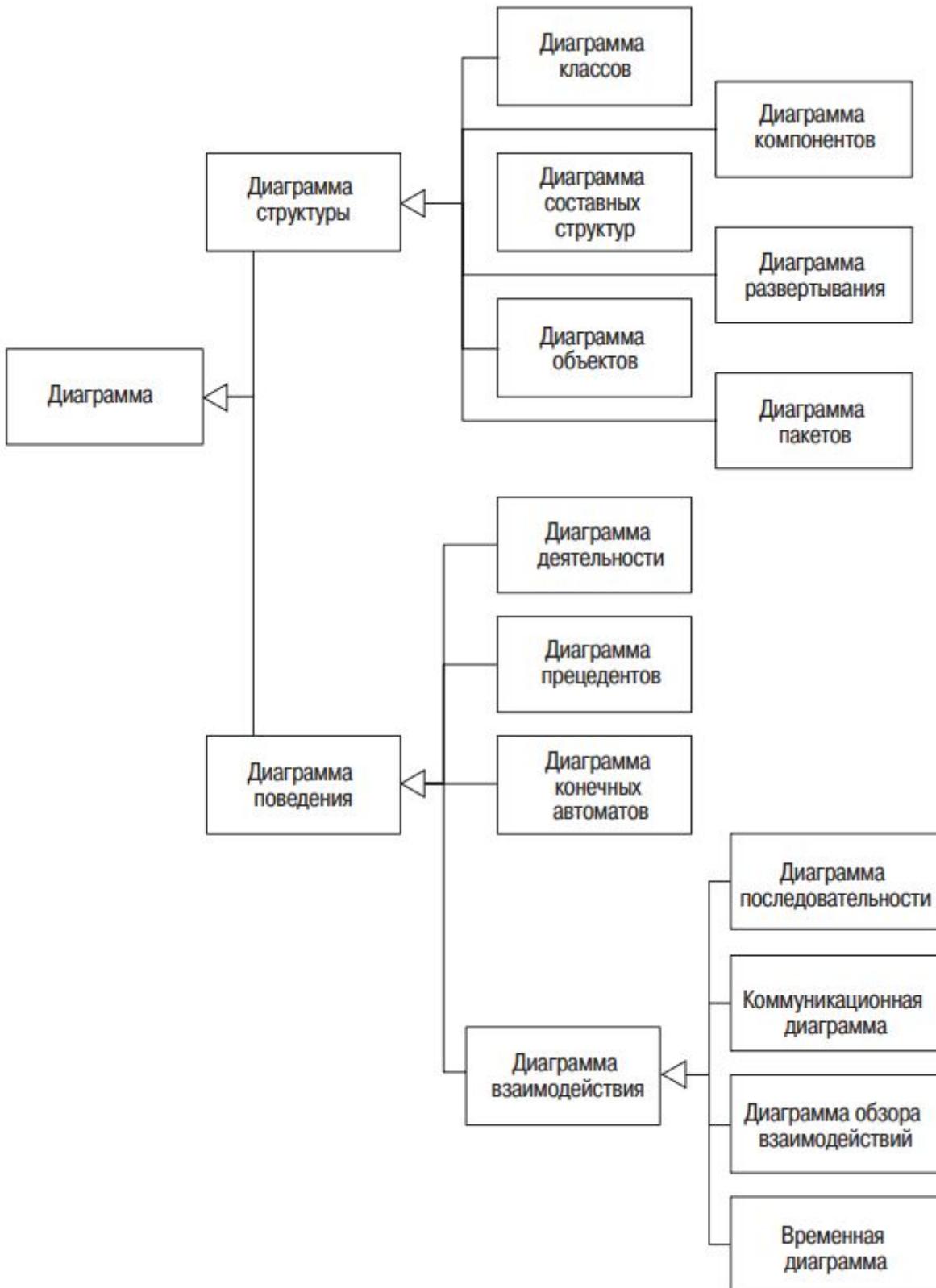
- Семейство графических нотаций
  - 14 видов диаграмм
- Общая метамодель
  - т.е. единое описание синтаксиса
- Стандарт под управлением Object Management Group
  - UML 1.1 — 1997 год
  - UML 2.0 — 2005 год
  - UML 2.5.1 — декабрь 2017 года
- Прежде всего, для проектирования ПО
  - После UML 2.0 стали появляться нотации и для инженеров
- Расширяем
  - **Профили** — механизм легковесного расширения, позволяют уточнить уже существующую нотацию UML для использования в какой-то конкретной предметной области.
  - **Метамоделирование** - позволяет взять стандарт UML и его подредактировать, добавив новые элементы или убрав существующие.

## История UML

ну на всякий случай

- Различного рода диаграммы использовались давным-давно (еще до появления ЭВМ)
  - Сети Петри, Flow Chart
- **Конец 60-х годов** - возникла серьёзная потребность в моделировании при разработке ПО
  - Data Flow Diagram, SADT (Structured Analysis and Design Technique), диаграммы Насси-Шнейдермана
- **Конец 70-х годов** - первый активный всплеск интереса к визуальным языкам
  - SDL (Specification and Description Language), Entity-Relationship, IDEF
- **Конец 80-х - начало 90-х годов** - второй пик развития
  - нотация Буча, OOSE, OMT, SDL, State charts, G, ДРАКОН
- **1995** - первые нестандартизированные спецификации UML
  - окончил эпоху языкового зоопарка
  - дальше плавно развивался
- **2005** - версия UML 2.0
  - дало толчок к развитию профилей UML
  - Executable UML, BPMN, SysML и т.д.
- **Начало-середина 00х годов** - стали модны предметно-ориентированные визуальные языки
- Постепенно интерес к визуальным языкам пошёл на спад
  - Agile Manifesto: "документация — это хорошо, но работающий код лучше"
  - Визуальные языки так и не показали обещанного прироста продуктивности
- Но UML всё ещё активно применяется в индустрии

## Виды диаграмм



Диаграммы делятся на две крупные категории — **структурные и поведенческие**.

- **Структурные диаграммы** показывают структуру системы времени компиляции

- это прежде всего диаграмма классов, диаграмма компонентов, диаграмма пакетов и другие.
- **Поведенческие диаграммы** показывают, как система себя ведёт во время работы
  - это диаграммы состояний, диаграммы последовательностей, диаграммы активностей, сюда же относят диаграммы случаев использования и другие, более специализированные диаграммы.

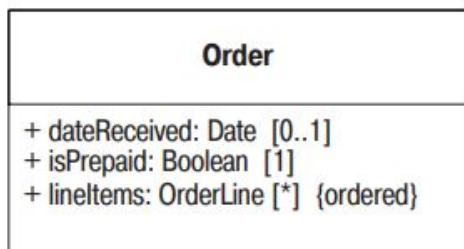
## Диаграмма классов, общий синтаксис

### САМАЯ ПОПУЛЯРНАЯ ДИАГРАММА UML

**Диаграмма классов - структурная** диаграмма, описывает типы объектов системы и различного рода статические отношения, которые существуют между ними. На диаграммах классов отображаются также свойства классов, операции классов и ограничения, которые накладываются на связи между объектами.

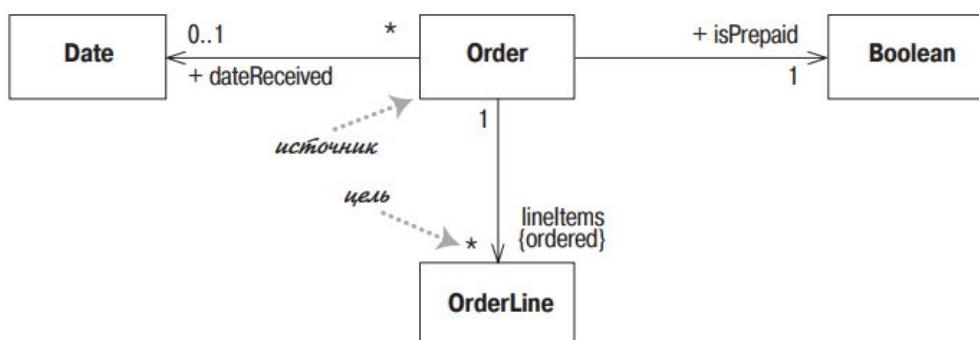
### Свойства

**Атрибут** описывает свойство в виде строки текста внутри прямоугольника класса.



- Полная форма атрибута:  
*видимость имя: тип кратность = значение по умолчанию {строка свойств}*
- Видимость:
  - + (public), - (private), # (protected), ~(package)
- Кратность:
  - 1 (ровно 1 объект), 0..1 (ни одного или один), \* (сколько угодно), 1..\*, 2..\*

**Ассоциация** – это непрерывная линия между двумя классами, направленная от исходного класса к целевому классу.

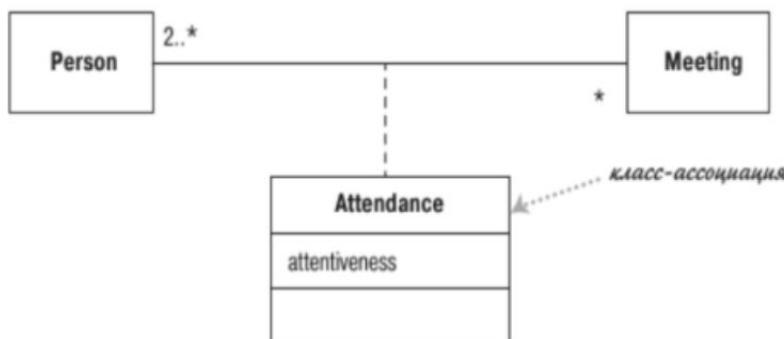


- Кратность ассоциации пишется у того конца ассоциации, к которому относится кратность.

Пример с картинки:

Date может иметь много Order, но у каждого Order ровно один Date.

- Видимость:
  - + (public), – (private), # (protected), ~(package)
- Кратность:
  - 1 (ровно 1 объект), 0..1 (ни одного или один), \* (сколько угодно), 1..\*, 2..\*
- Ассоциациям самим разрешается иметь атрибуты



- Может быть двунаправленной



Атрибуты и ассоциации представляют собой с точки зрения синтаксиса языка одно и то же, просто отображаются по-разному.

- Атрибуты обычно используются, когда связи между классами не важны
- Ассоциации — когда связи между классами важны для понимания архитектуры

Наиболее распространенным представлением свойств является поле или свойство языка программирования.

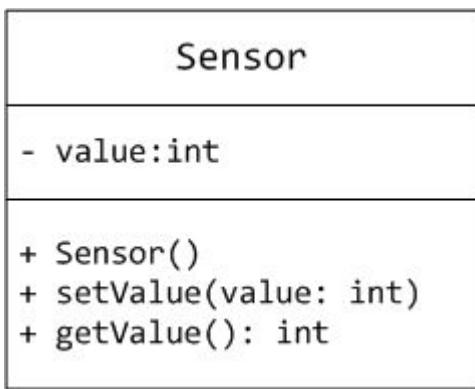
## Операции

**Операции (operations)** представляют собой действия, реализуемые некоторым классом.

Существует очевидное соответствие между операциями и методами класса.

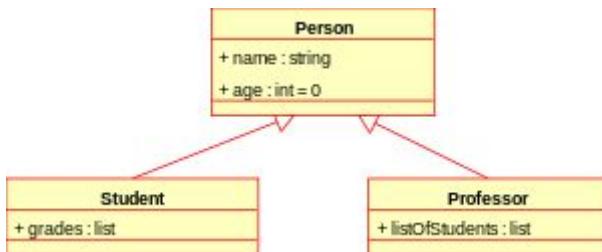
Полный синтаксис операций в языке UML выглядит следующим образом:  
**видимость имя (список параметров) : возвращаемый тип {строка свойств}**

Пишутся в нижней части прямоугольника класса:

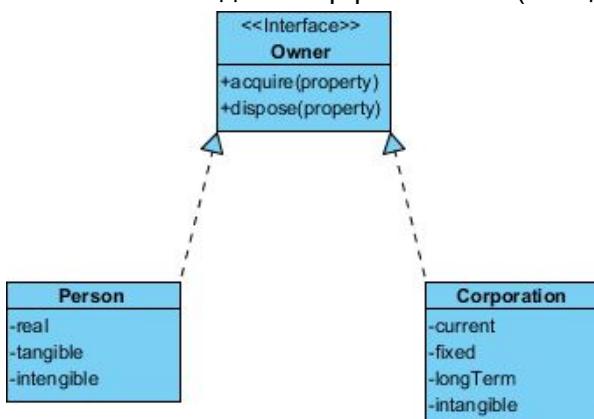


## Интерфейсы

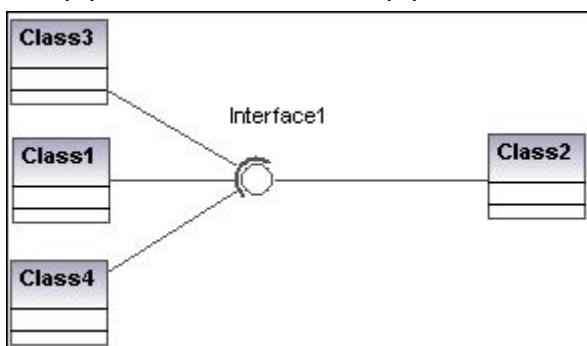
- Наследование в UML называется “**обобщение**” (**generalization**) и стрелка указывает на предка, а не на потомка (с этим часто путаются).



- Реализация** интерфейса и наследование от класса в UML суть разные вещи, реализация интерфейса рисуется пунктирной линией.
- Нотация со словом <<интерфейс>>** (<<interface>>) используется, когда надо показать и методы интерфейса тоже (и тогда они пишутся как методы класса).



- “**Леденцовая**” нотация предпочтительней, если нам не очень важны методы интерфейса либо если интерфейсов много.



## Зависимости

Литвинов:

**Зависимость** — наиболее общий вид связи между классами, она означает просто, что один класс (или интерфейс) что-то знает про другой.

М. Фаулер. Основы UML:

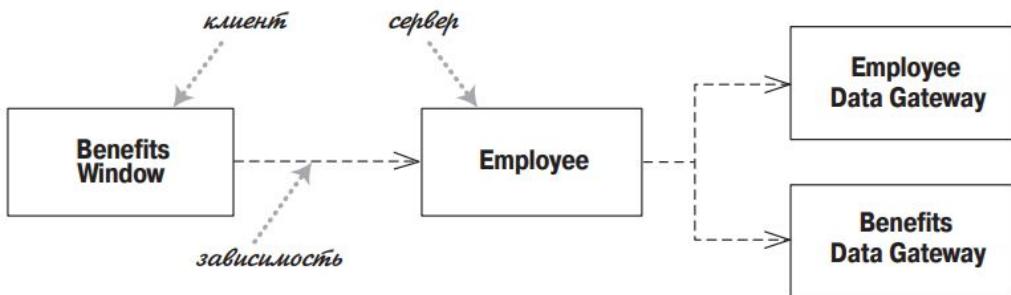
Считается, что между двумя элементами существует **зависимость (dependency)**, если изменения в определении одного элемента (**сервера**) могут вызвать изменения в другом элементе (**клиенте**).

Зависимости могут быть уточнены стереотипами, некоторые из которых прописаны в стандарте.

**стереотип** = <<call>> и т.д. на стрелочке

- **call** — один класс вызывает метод другого;
- **create** или **instantiate** — один класс создаёт в каком-то из своих методов объект другого;
- **derive** — один класс представляет значение, которое может быть вычислено по другому;
- **realize** — один класс реализует другой (например, абстрактный класс);
- **responsibility** — контракт, который класс обязуется исполнять;
- **refine** — зависимость, которая может быть установлена даже между элементами с разных диаграмм, один элемент является уточнением другого
- **trace** — зависимость, которая может быть установлена даже между элементами с разных диаграмм, означает, что один элемент как-то влияет на другой. Нужны такие зависимости прежде всего для автоматического отслеживания изменений.

Обозначается пунктирной стрелочкой. Если надо уточнить зависимость, то одно из упомянутых выше слов пишется <<вот так>> над линией по центру.



## Агрегация и композиция

Агрегация и композиция — дальнейшие уточнения ассоциации. На самом деле, если на диаграмме нарисована просто ассоциация, то при реализации мы вправе выбрать, агрегацию или композицию использовать.

Разница между агрегацией и композицией — во владении объектами, которые участвуют в ассоциации.

**Агрегация** не предполагает владения.

РОМБИК НЕ ЗАКРАШЕН

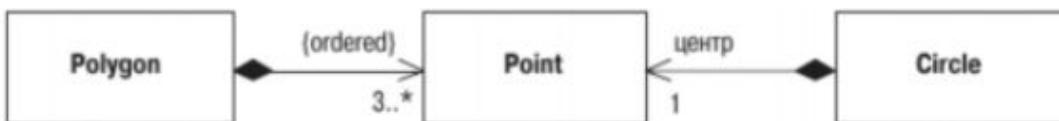


Человек и клуб могут существовать независимо друг от друга, и когда закрывается клуб, его члены продолжают существовать. На диаграмме выше нарисовано, что в клубе может состоять несколько человек, и человек может состоять в нескольких клубах, при этом клуб знает о человеке, но не владеет им.

**Композиция** говорит, что один объект владеет другим объектом, то есть время их жизни связано и “хозяин” отвечает за удаление подчинённого ему объекта.

РОМБИК ЗАКРАШЕН

Например, если мы пишем графический редактор, вполне может быть, что мы сделаем так:



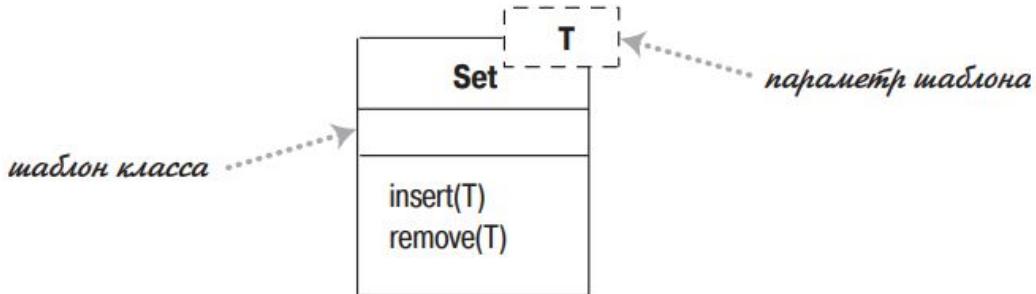
Здесь многоугольник состоит из точек, и когда мы удаляем многоугольник, удаляются и все точки, которые его определяют.

С точки зрения нотации надо запомнить, что агрегация — более слабое отношение, чем композиция, поэтому и ромбик в случае агрегации не закрашен.

И ромбики рисуются около контейнера (то есть клуб содержит людей, а не наоборот) — можно понимать ромбик как оперение стрелы, указывающей от контейнера к содержащемуся в нём объекту.

## Шаблоны и перечисления

Параметры-типы перечисляются через запятую в пунктирном прямоугольнике в правом верхнем углу класса.

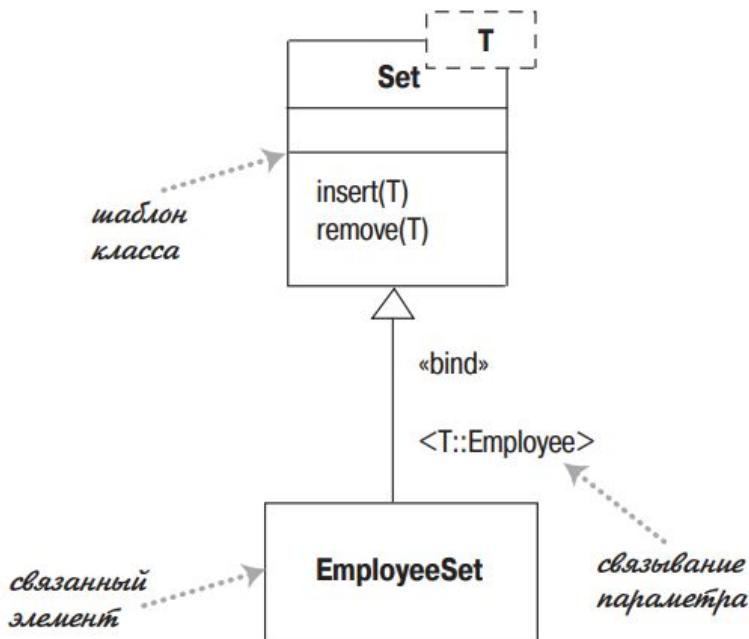


Инстанцирование шаблона (то есть подстановка фактических параметров-типов вместо формальных) рисуется:

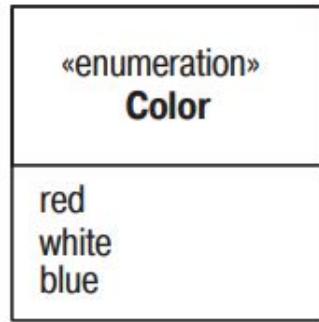
- либо как просто класс, у которого указано, что куда подставляется,



- либо как наследование со стереотипом <>bind>>



**Перечисления** используются для представления фиксированного набора значений, у которых нет других свойств, кроме их символических значений. Они изображаются в виде класса с ключевым словом <>enumeration>>.



## 11. Диаграммы объектов, диаграммы пакетов UML.

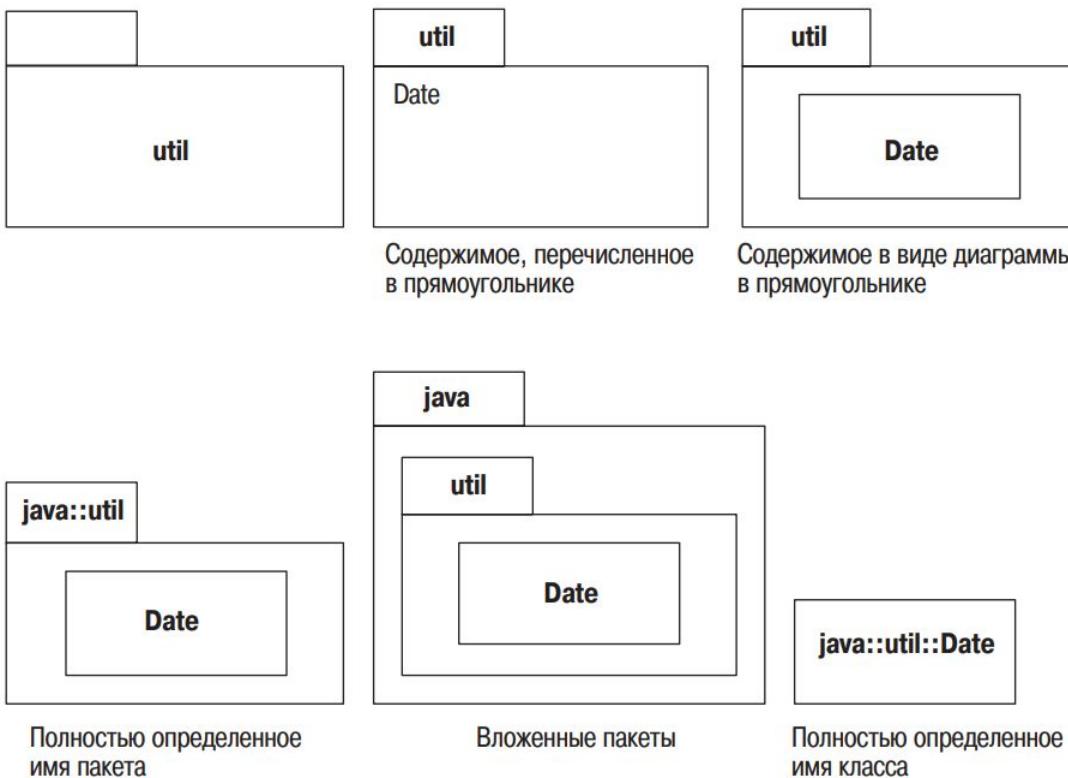
[конспект преза](#)

### Диаграммы пакетов

Это тоже **структурные** диаграммы, нужны они для того, чтобы показать разбиение кода по пакетам, либо сгруппировать по пакетам саму модель.

“Пакет” в UML означает более-менее то же, что пакет в Java или пространство имён в C++.

Синтаксис:



*Рис. 7.1. Способы изображения пакетов на диаграммах*

На диаграммах пакетов также могут рисоваться компоненты и классы, которые в этих пакетах находятся.

Диаграммы пакетов полезны ещё тем, что на них можно визуализировать зависимости между пакетами.

**Зависимость** - примерно то же самое, что в диаграмме классов (пунктирная стрелочка, над ней может быть <<какая-нибудь надпись>>)

Просто **зависимость** означает, что пакет как-то связан с другим пакетом, **стрелка** “реализация” означает, что в пакете-предке есть хотя бы один класс, являющийся предком хотя бы одного класса из пакета-потомка, либо интерфейс, который пакет-потомок реализует.

На самом деле, часто отношение реализации рисуют в том случае, если смысл пакета-предка в архитектуре — содержать интерфейсы или абстрактные классы, которые должны реализовывать другие пакеты.

## Диаграммы объектов

Визуализируют объекты в памяти во время работы программы и отношения между ними.

По смыслу относятся к поведенческим диаграммам, но их считают **структурными**, так как они описывают структуру времени выполнения, а не характер взаимодействия объектов.

Диаграммы объектов используют в двух случаях:

- чтобы визуализировать и проиллюстрировать диаграммы классов
- чтобы разобраться во взаимосвязях реально существующих объектов предметной области, ещё до того, как создана первая диаграмма классов (и тогда диаграмма классов будет на самом деле наведением классификации на объекты, выделенные на диаграмме объектов).

Литвинов обращает внимание на этот случай использования: в методологии DDD используются для анализа предметной области

## Синтаксис

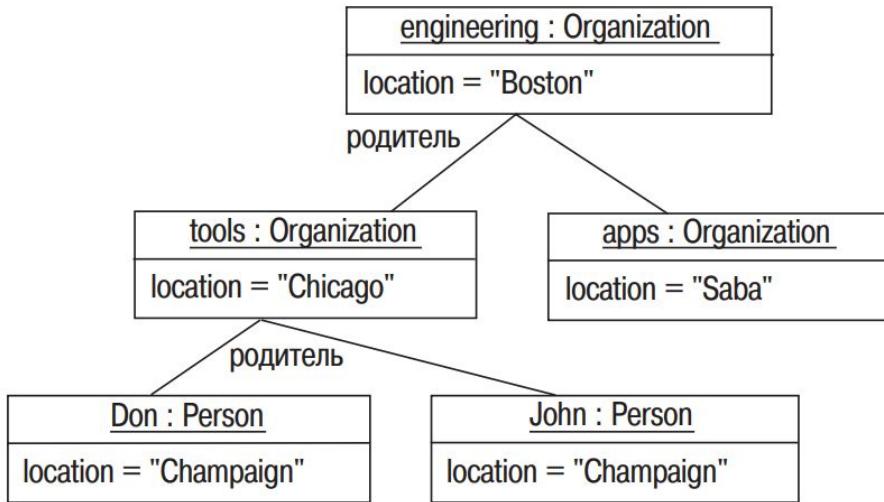


Диаграмма объектов с примером экземпляра класса Party

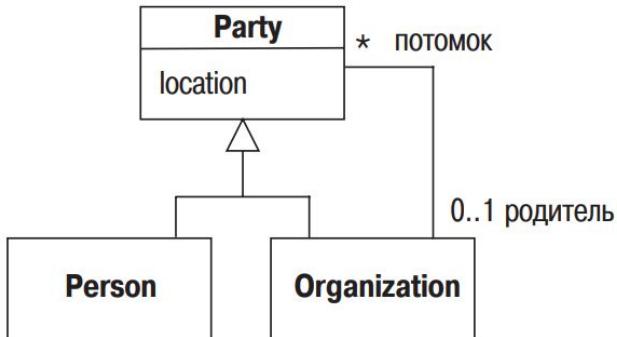


Диаграмма классов, показывающая структуру класса Party (вечеринка)

## 12. Диаграммы компонентов, диаграммы развёртывания UML.

### Диаграммы компонентов

[конспект преза](#)

На них изображаются компоненты, из которых состоит система или подсистема, и взаимосвязи между ними.

Относятся к **структурным** диаграммам.

**Компонент** - нечто структурно связанное и большее, чем класс (пакеты, пространства имен, сборки, .dll/.so-файлы, отдельные веб-сервисы и т.д.). Именно то, из чего состоит высокоуровневая архитектура приложения.

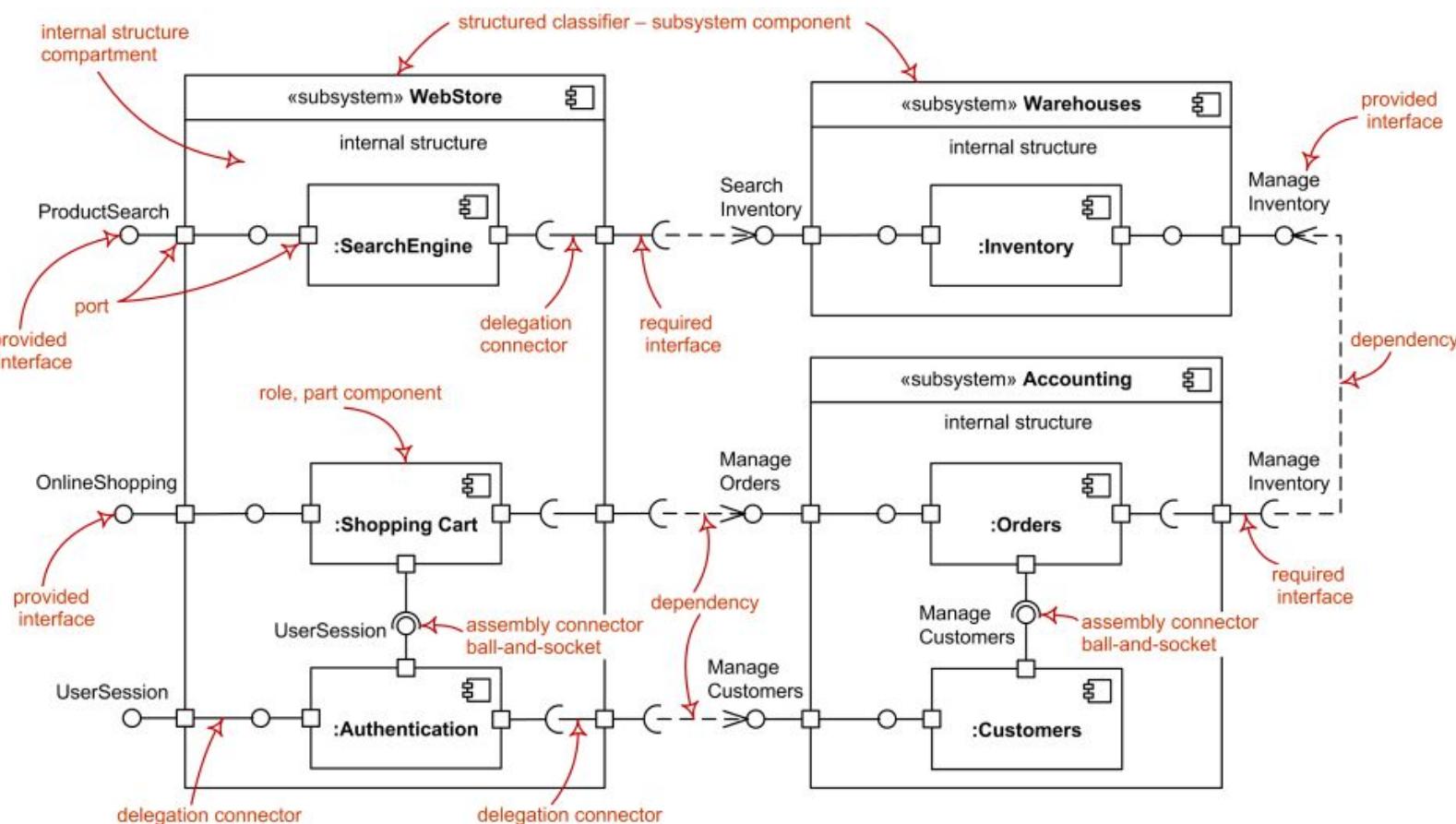
Важнее и полезнее диаграмм классов

- неизменны до глобальной переработки архитектуры
- позволяют посмотреть на высокоуровневую архитектуру создаваемой системы

## Синтаксис



*нотация UML 2*



## Диаграммы развертывания

что-то у Литвинова вообще не нашла, читала в Фаулере

**Диаграммы развертывания** представляют физическое расположение системы, показывая, на каком физическом оборудовании запускается та или иная составляющая программного обеспечения.

Относятся к **структурным** диаграммам.

## Синтаксис

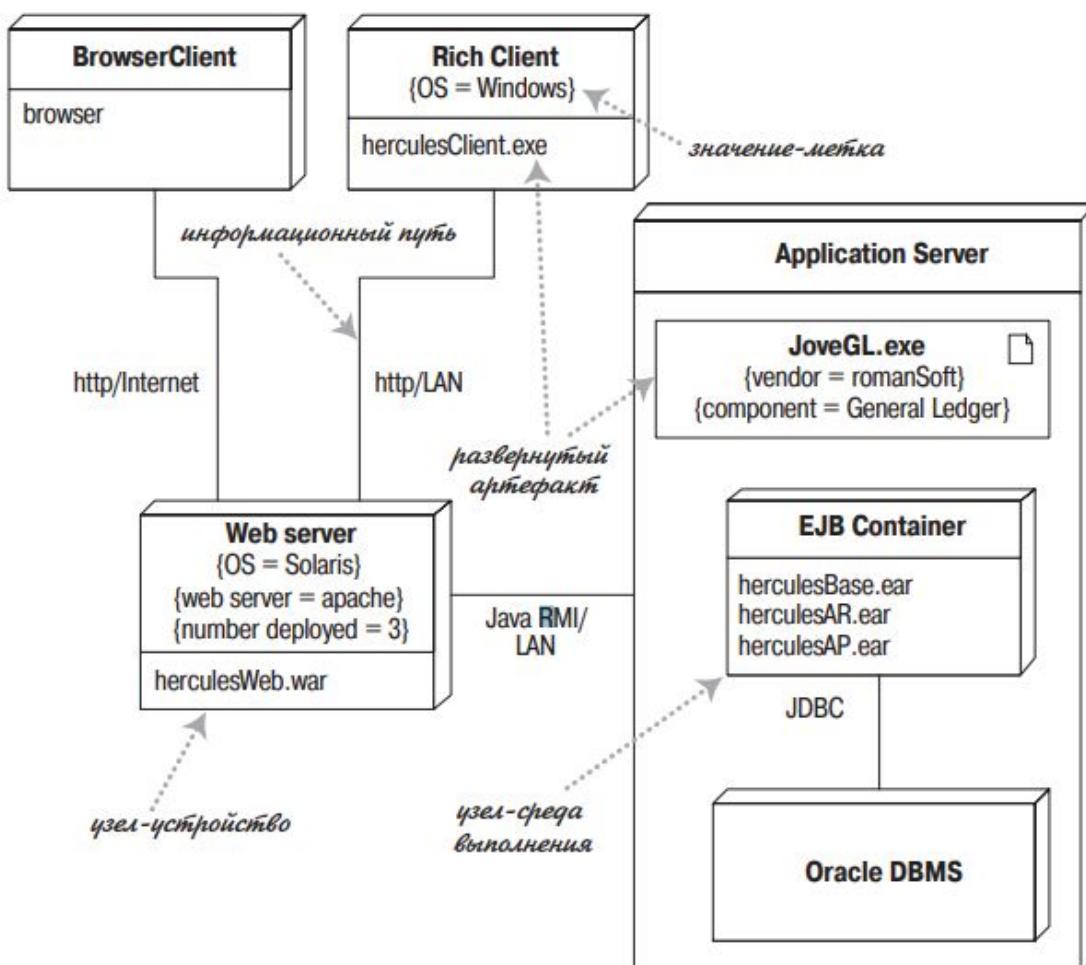
Главными элементами диаграммы являются узлы.

**Узел (node)** – это то, что может содержать программное обеспечение.

Бывает двух типов:

- **Устройство (device)** – это физическое оборудование.
- **Среда выполнения (execution environment)** – это программное обеспечение, которое само может включать другое программное обеспечение, например операционную систему или процесс-контейнер.

Узлы могут содержать **артефакты (artifacts)**, которые являются физическим олицетворением программного обеспечения; обычно это файлы.



## 13. Диаграмма случаев использования UML.

[конспект преза](#)

Ивар Якобсон, 1992

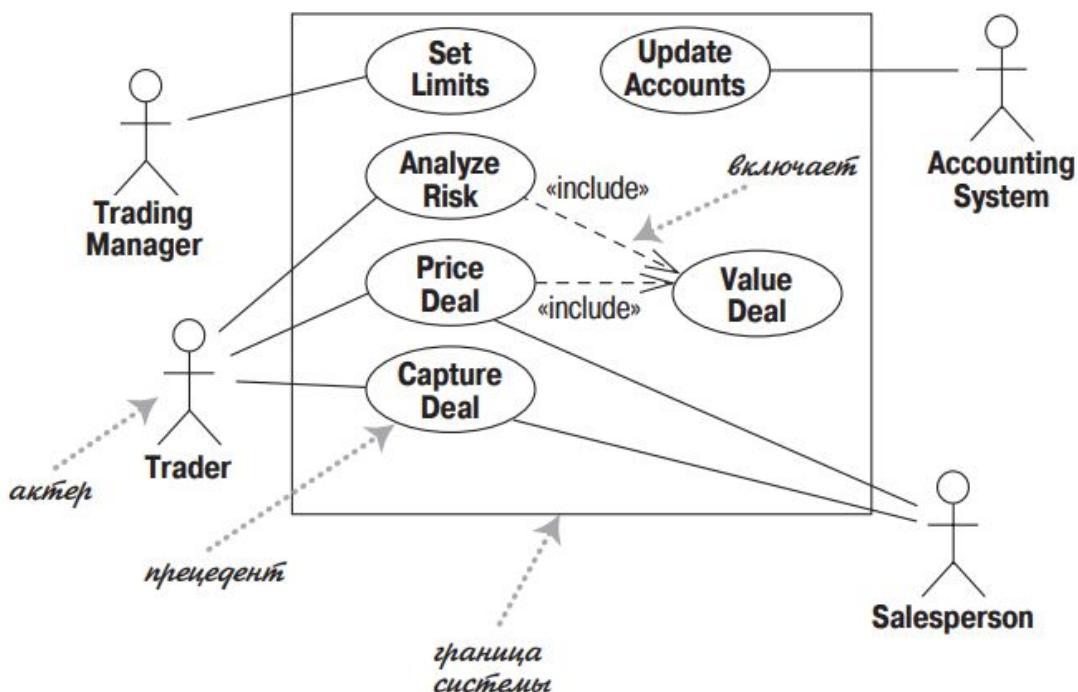
Также называют **диаграммой прецедентов** или **use case-диаграммой**.

Относится к **поведенческим** диаграммам.

Используется для анализа требований

## Синтаксис

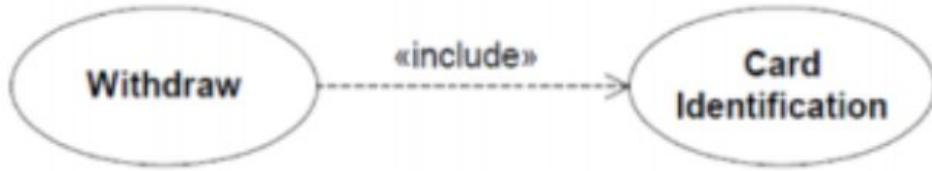
Состоит всего из двух типов сущностей и границы системы:



- **Акторы/актеры/роли** — внешние сущности, использующие систему.
  - Это могут быть группы пользователей, объединённые общей целью использования системы.
  - Это могут быть также внешние программные системы, которые пользуются нашей.
- **Случаи использования (прецеденты)** — цель использования системы актором.
  - Это 1-3 слова, описывающие, что пользователь хочет в итоге получить («Проанализировать риски», «Оценить сделку»)
- Каждый случай использования должен раскрываться в один или несколько **сценариев использования системы**.
  - Говорят, что и в каком порядке делает пользователь, чтобы достичь цели (например, логинится -> выбирает нужную сделку -> анализирует риски -> описывает результаты анализа в соответствующую форму).
  - Чаще всего описываются просто неформальным текстом, можно диаграммами активностей UML

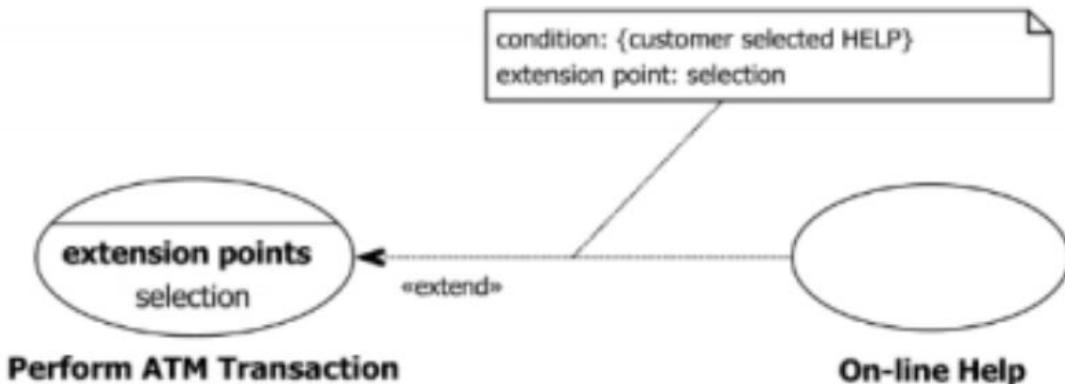
## Отношения между случаями использования

**Include** - один случай использования включает в себя другой



Например, при снятии налички пользователь хочет, чтобы система идентифицировала его банковскую карту.

**Extend** - один случай использования имеет одну или несколько точек расширения, в которых может использоваться функциональность, связанная с другим случаем использования.



Например, «Проведение транзакции с помощью банкомата» имеет точки расширения, где пользователь может получить справку по использованию системы.

Это может выражаться в связанных сценариях использования, например, словами «*А теперь пользователь хочет посмотреть баланс, но если он не знает, как это сделать, он может нажать на кнопку “показать справку”*».

## Подробнее про сценарии использования

Каждый случай использования раскрывается в один или несколько сценариев использования.

Типичная структура сценария использования такова:

- **Заголовок** — цель основного актора, собственно, случай использования.
- **Заинтересованные лица, акторы, основной актор** — сюда выписываются все роли, участвующие в сценарии, среди них выделяется одна — которая инициирует исполнение сценария.
- **Предусловия** — какие логические условия должны быть истинны, чтобы сценарий в принципе мог быть выполнен.

- **Триггеры (активаторы)** — что должно произойти, чтобы сценарий начал выполняться, при условии, что все предусловия выполнены.
- **Основной порядок событий** — что происходит, если всё идёт по плану. Сюда пишется наиболее типичная последовательность действий.
- **Альтернативные пути и расширения** — сюда пишется, что делать в нетипичной ситуации, тут же описываются точки расширения и реакция на ошибки.
- **Постусловия** — какие логические условия должны быть истинны после исполнения сценария.

## 14. Диаграмма активностей UML.

[конспект преза](#)

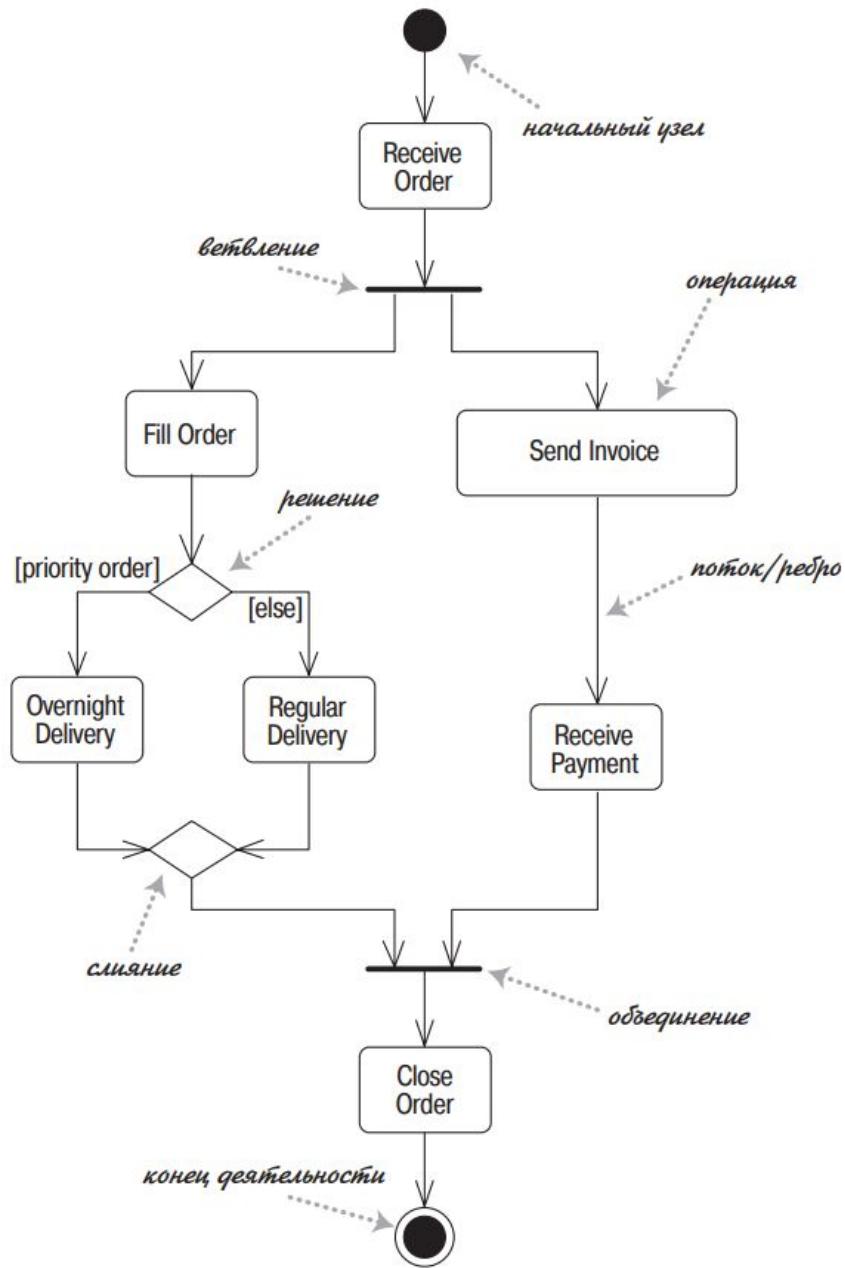
Также называют **диаграмма деятельности**.

Относится к **поведенческим** диаграммам.

Используется для моделирования поведения бизнес-процесса.

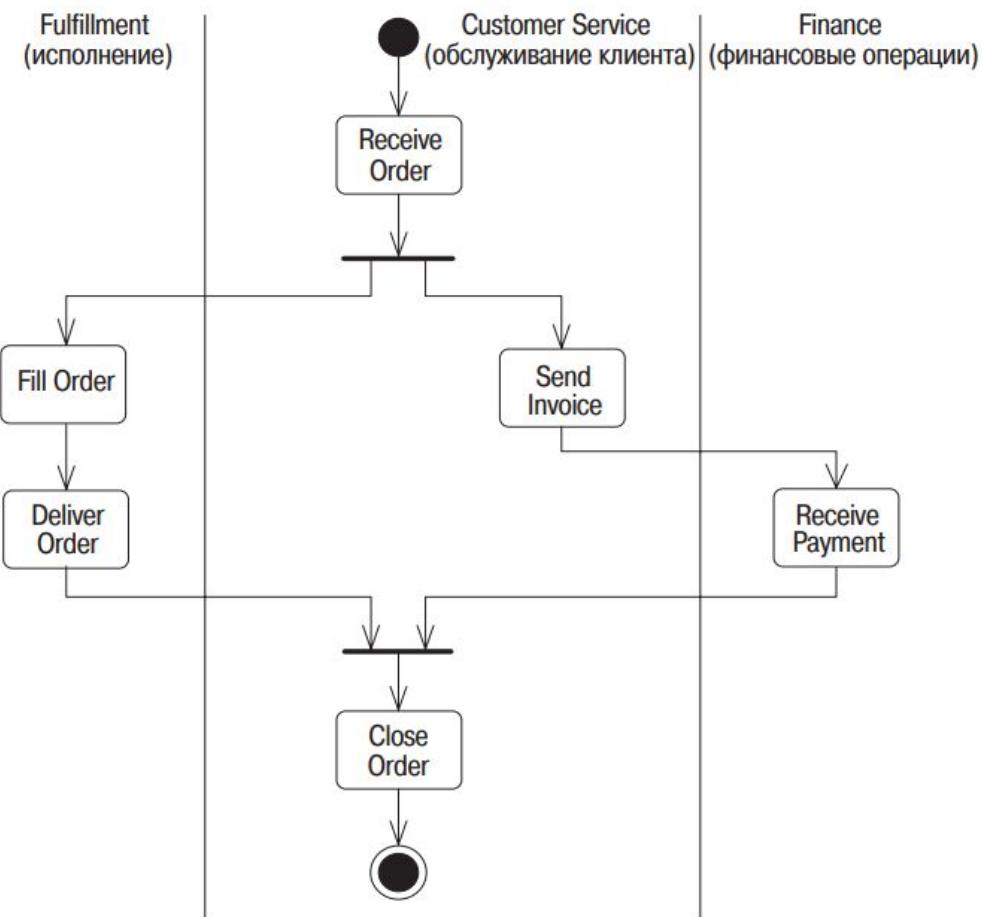
- для того, чтобы понять, как в существующий процесс вписывается разрабатываемая система,
- это хороший способ визуализации сценария использования системы с точки зрения пользователя.

## Синтаксис



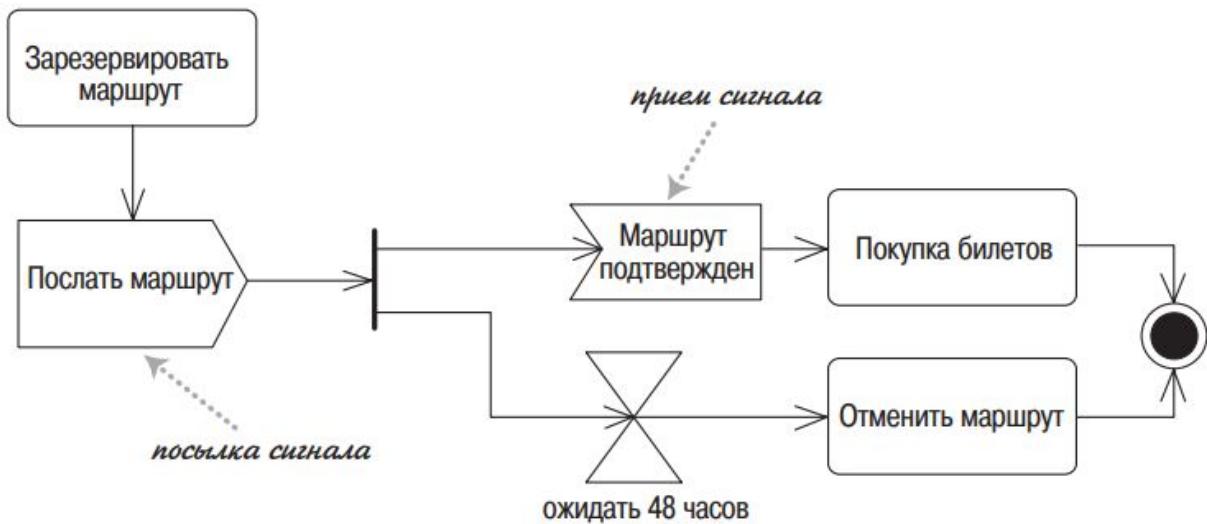
- Похожа на блок-схему, но поддерживает параллельные процессы
- «Ветвление»/«объединение» работает как fork/join для параллельных потоков
- Каждый блок «решение» может иметь несколько веток, но все ветки должны сходиться на блоке «слияние».

Еще можно показать разделение по отделам организации или частям системы:



И еще есть возможность показать асинхронное взаимодействие с помощью сигналов





### Различия с диаграммой конечных автоматов

- На диаграмме активностей рисуются активности
  - система в них не задерживается, а сразу переходит дальше
- На диаграмме конечных автоматов рисуются состояния
  - стабильные отрезки жизненного цикла объекта, в которых он находится большую часть времени и может из них выйти только если что-то произойдёт;
- полезная работа на диаграммах активностей производится в активностях, на диаграммах автоматов — как правило, при переходе;
- диаграммы активностей моделируют один метод объекта, диаграммы конечных автоматов — целый объект (состоиния моделируются полями объекта).

## 15. Диаграммы конечных автоматов UML.

[конспект преза](#)

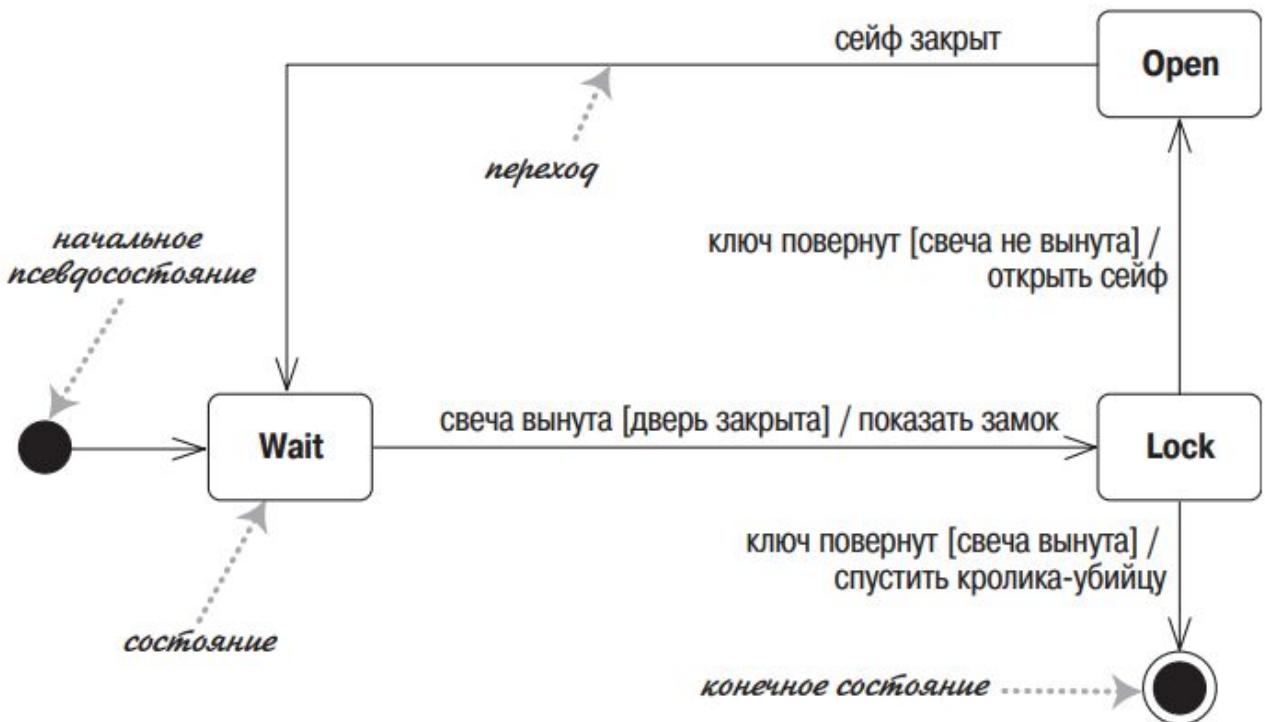
Также известны как **диаграммы состояний**.

Относятся к **поведенческим** диаграммам.

Предназначены для моделирования поведения «реактивных» систем (или частей системы), то есть систем, которые находятся в некоторых чётко определённых состояниях, от которых зависит их поведение, и могут реагировать на события, переходя из состояния в состояние и, возможно, делая при переходах полезную работу.

(например, сетевое соединение или торговый автомат)

## Синтаксис



- **Состояния** рисуются прямоугольниками со скруглёнными углами
  - У состояния есть имя и (опционально) действия, выполняемые в состоянии
 

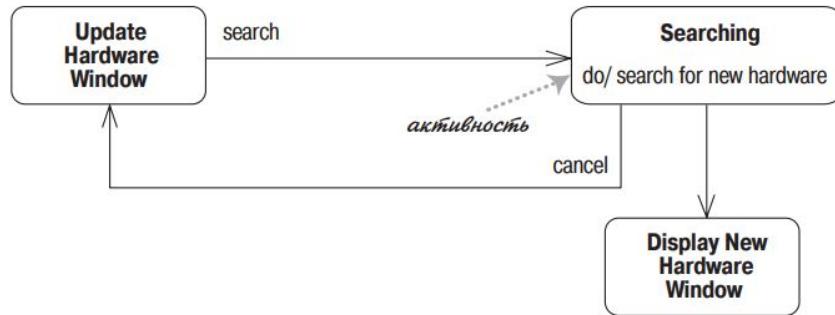
**Typing**

entry/ выделить все  
exit/ обновить поле  
character/ обработка символа  
help [verbose]/ открыть страницу помощи  
help [quiet]/ обновить панель статуса
- Состояния связаны **переходами** (стрелочками)
  - Над переходом пишется событие, которое инициирует переход, и действие, выполняемое при переходе
  - И, опционально, **стражник (guard)** - логическое условие, которое должно быть истинно, чтобы переход состоялся
  - События со стражниками должны быть взаимно исключающими
  - Синтаксис надписи на переходе:  
 $[<\text{trigger}> [, <\text{trigger}>]* [<\text{guard}>]] [/ <\text{behavior-expression}>]]$ 

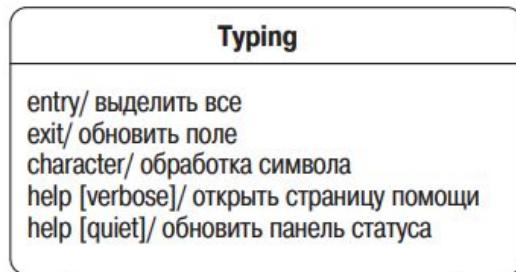
пример: свеча вынута [дверь закрыта] / показать замок
- Есть псевдосостояния начала и конца
  - Переход из псевдосостояния начала происходит мгновенно
  - Переход в состояние конца заканчивает исполнение

## Более подробно про синтаксис

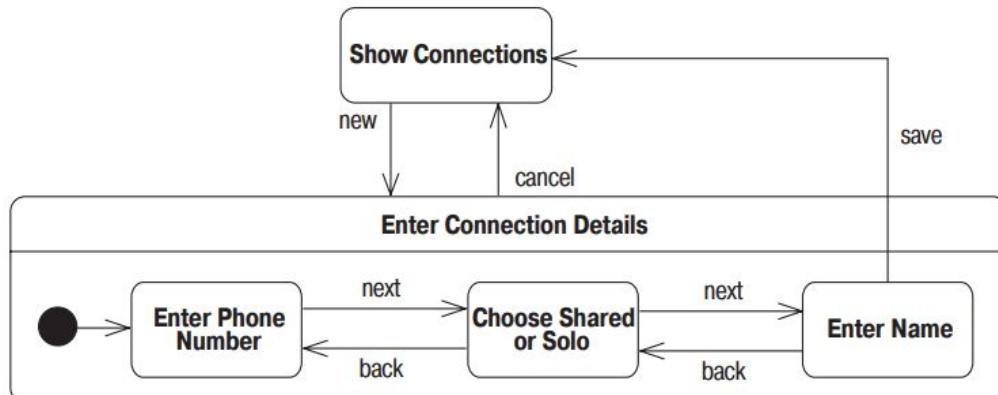
- Внутри состояния могут быть:
  - **entry activity** — то, что делается при входе в состояние по любому из переходов;
  - **exit activity** — то, что делается при выходе из состояния по любому исходящему переходу
  - **do activity** — деятельность, выполняющаяся всегда, когда система находится в таком-то состоянии



- **внутренний переход** — переход по событию, который ведёт в то же состояние и не приводит к срабатыванию entry и exit activity. Переход вполне может быть полноценным переходом в то же состояние (рисуется как петля в графе), тогда entry и exit activity работают как обычно, хоть состояние и не меняется.

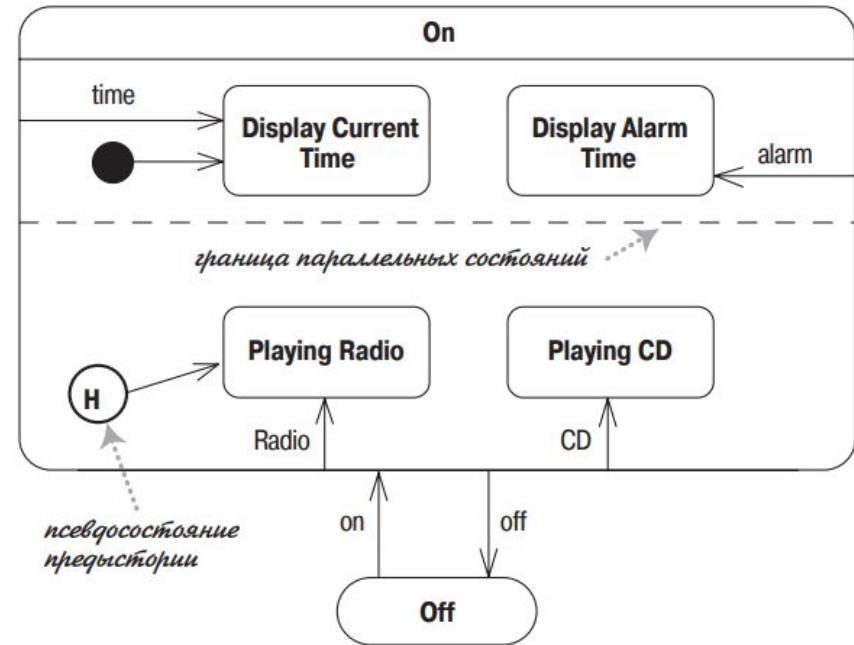


- Есть вложенные состояния с переходами сразу из всех внутренних состояний



Тут состояние «Enter connection details» содержит внутри свой конечный автомат, который начинает работать со стартового псевдосостояния, когда выполняется переход «new».

- Есть параллельные состояния и псевдосостояние истории



- 
- По сути это два автомата, работающих параллельно (что и показывает горизонтальная прерывистая линия, разделяющая параллельные подавтоматы).
- Стрелки от объемлющего состояния к вложенным означают, что система, находясь в объемлющем состоянии, реагирует на такие-то события и изменяет внутреннее состояние
- Псевдосостояние истории запоминает последнее вложенное состояние, в котором находился автомат, и возвращает автомат в него.
- Пример: часы с радио и будильником, проигрывание звука и время работают независимо

## Различия с диаграммой активностей

- На диаграмме активностей рисуются активности
- система в них не задерживается, а сразу переходит дальше
- На диаграмме конечных автоматов рисуются состояния
- стабильные отрезки жизненного цикла объекта, в которых он находится большую часть времени и может из них выйти только если что-то произойдёт;
- полезная работа на диаграммах активностей производится в активностях, на диаграммах автоматов — как правило, при переходе;
- диаграммы активностей моделируют один метод объекта, диаграммы конечных автоматов — целый объект (состояния моделируются полями объекта).

## Генерация кода

По конечным автоматам легко генерировать код.

- Самый простой способ: **гигантский switch**, внутри которого switch-и поменьше
  - внешний switch по текущему состоянию автомата

- внутренние switch-и по события, на которые автомат может реагировать в данном состоянии
- внутри - проверка условий стражников, выполнение действия по переходу и переход в следующее состояние.
- состояние моделируется enum-ом, хранящимся как поле объекта
- Второй способ: **таблица состояний** + универсальный интерпретатор, который ищет в таблице текущее состояние и событие
  - Обычно либо хранится как файл данных рядом с программой, либо вкомпилируется в исходный код.
  - Очень популярен в лексическом и синтаксическом анализе, но отлаживать или просто понимать такие программы очень тяжело.
  - Пример:

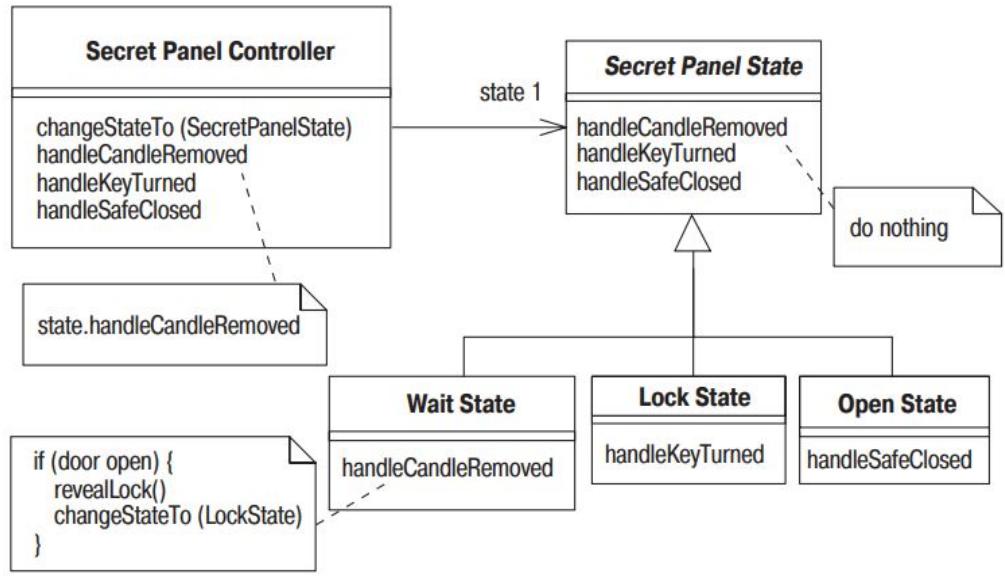
Исходное состояние	Целевое состояние	Событие	Защита	Процедура
Wait	Lock	Candle removed (свеча удалена)	Door open (дверца открыта)	Reveal lock (показать замок)
Lock	Open	Key turned (ключ повернут)	Candle in (свеча на месте)	Open safe (открыть сейф)
Lock	Final	Key turned (ключ повернут)	Candle out (свеча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

- Третий способ: **паттер “Состояние”**
  - Автомат представляется в виде класса, который имеет в качестве поля ссылку на интерфейс «текущее состояние».

Этот интерфейс имеет столько методов, сколько всего разных событий может обрабатывать автомат. Интерфейс реализуют конкретные классы, отвечающие за конкретные состояния, они определяют те методы интерфейса, на которые могут реагировать, там уже проверяют условия стражников и выполняют действие при переходе.

Каждый такой метод возвращает объект-состояние, в которое должен перейти автомат дальше.

- То есть, по сути, это тот же switch, где самый большой switch (по состояниям) спрятан в таблицу виртуальных методов.
- Такой подход делает реализацию автомата более читаемой, ограничивает ответственность каждого класса только одним состоянием и позволяет очень легко добавить новые состояния.
- Пример:



## 16. Диаграммы последовательностей UML.

[конспект преза](#)

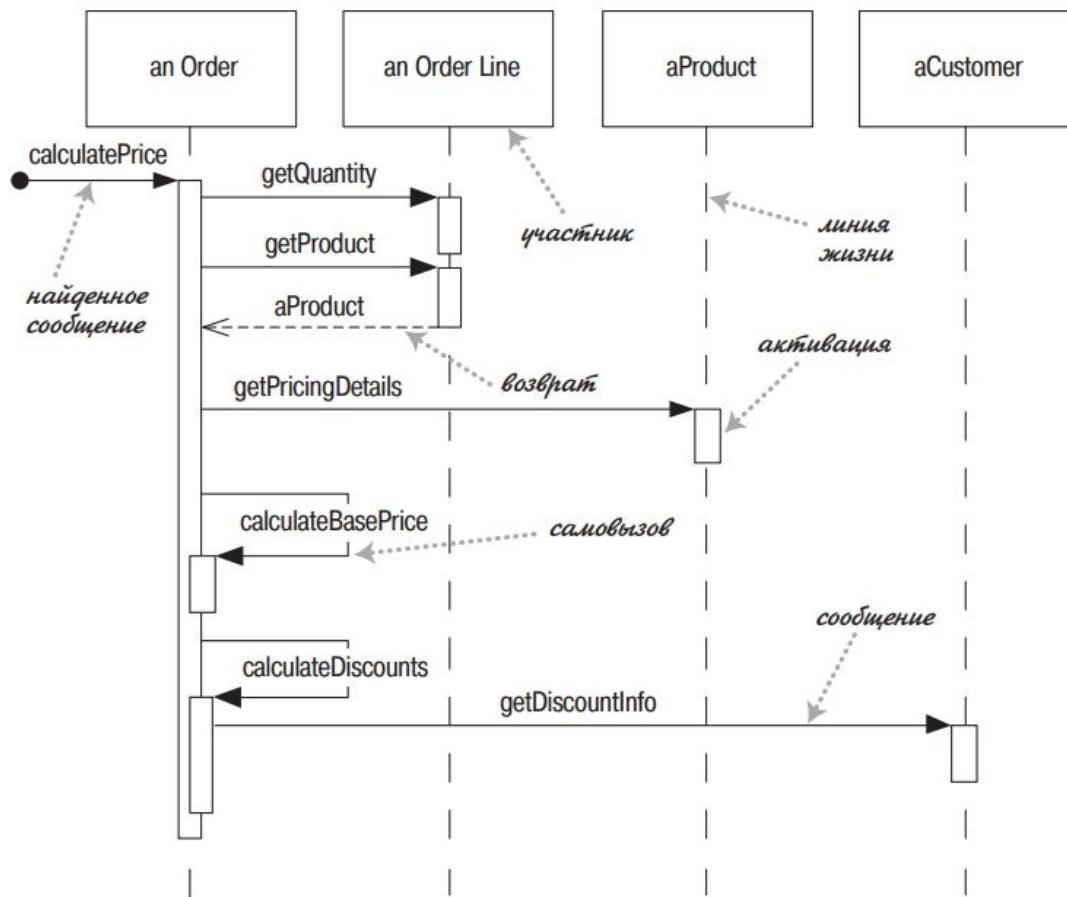
Относятся к **поведенческим** диаграммам.

Применяются для визуализации взаимодействия между объектами, подходят только для визуализации одного сценария взаимодействия (есть фреймы, но они неудобные) и показывают последовательность обмена сообщениями:

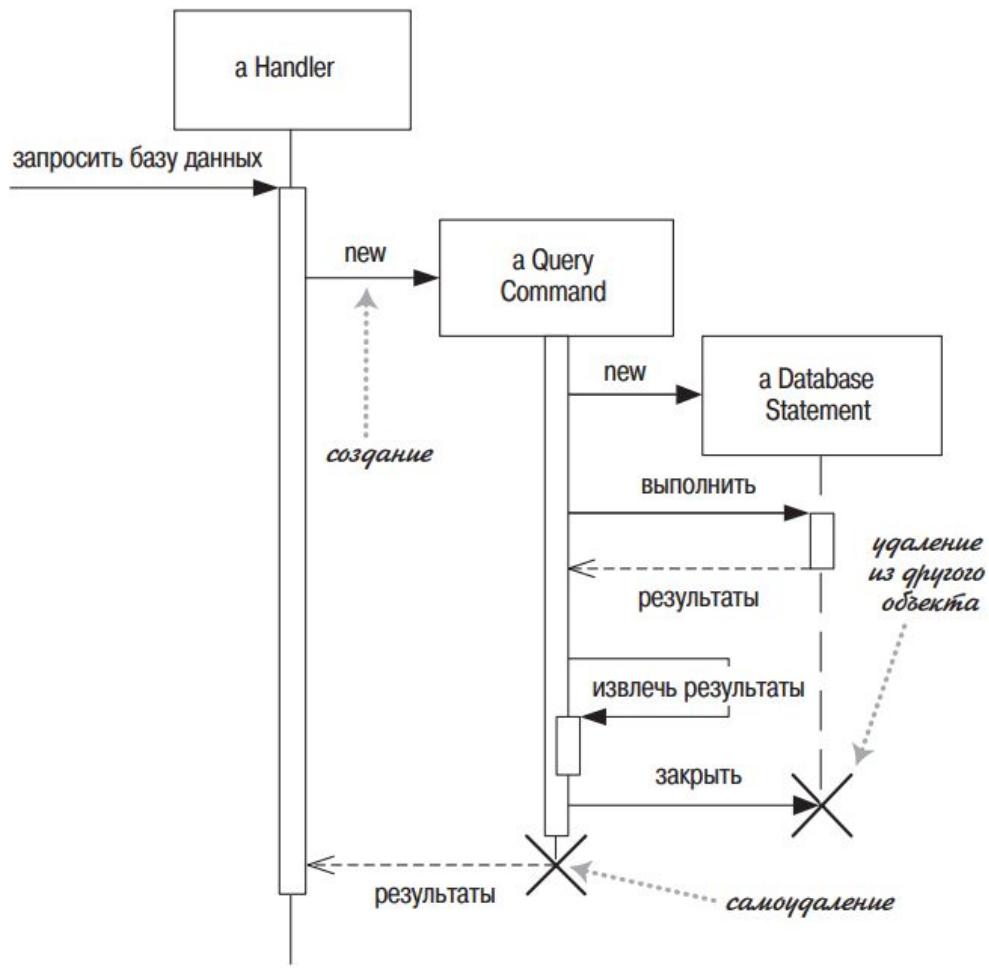
Примеры:

- при многопоточном и асинхронном программировании, чтобы визуализировать общение между потоками/асинхронные вызовы
- при описании телекоммуникационных протоколов
- на этапе анализа предметной области для визуализации последовательности коммуникаций между участниками взаимодействия
- на этапе проектирования для составления плана тестирования — кто в каком порядке должен дёргать и кто когда чем должен отвечать
- на этапе отладки, для визуализации логов работающей системы

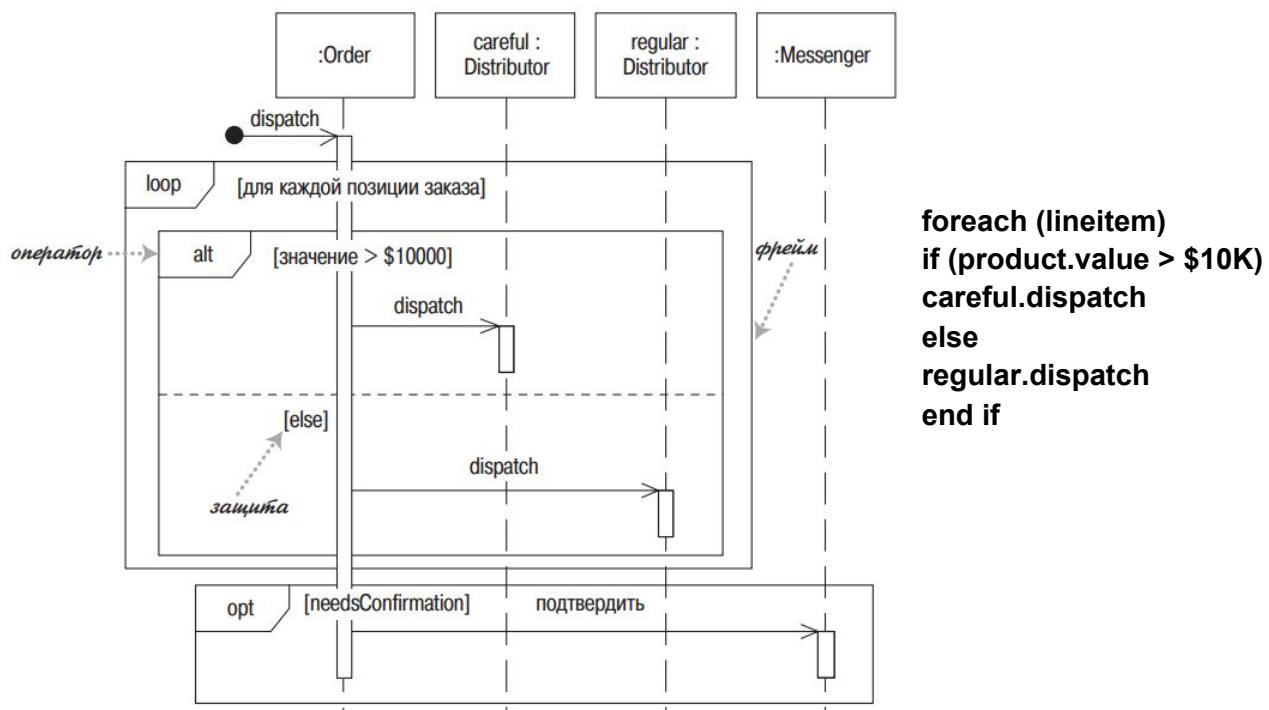
## Синтаксис



- На диаграмме рисуются **объекты**, из каждого объекта выходит **линия жизни** (пунктирная линия), на которой расположены **линии активации** (длинный белый прямоугольник).
  - Линия жизни** показывает, когда объект существует в памяти
  - Линия активации** показывает, когда объект занят какой-то работой
    - одновременно может быть несколько
- Стрелки между линиями активации обозначают **сообщения**
  - как правило, это вызовы методов
  - в сообщении можно передавать **параметры**
    - пишется `getPrice(quantity: number)`
  - можно **возвращать** значения из вызовов
- Можно показать **создание и удаление** объекта
  - Создание означает, как правило, вызов конструктора
  - Удаление — либо вызов деструктора, либо то место, где объект становится больше не нужен и его может собрать сборщик мусора.



- Можно показать целый алгоритм с ветвлениями и циклами с помощью фреймов



- Сильно ухудшают читабельность диаграммы
- Очень ситуационная штука

## 17. Диаграммы коммуникации UML.

[конспект преза](#)

Относятся к **поведенческим** диаграммам.

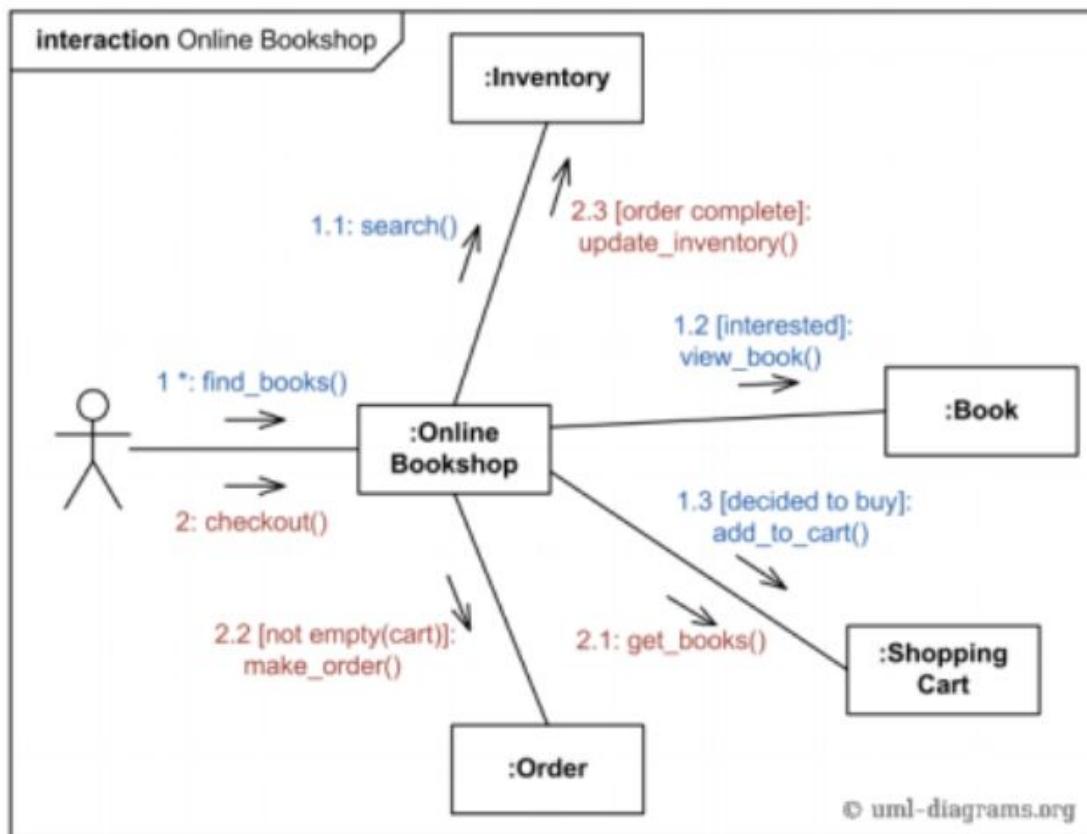
По сути те же диаграммы последовательностей, только «с высоты птичьего полёта».

Применяются для визуализации взаимодействия между объектами, подходят только для визуализации одного сценария взаимодействия и показывают последовательность обмена сообщениями:

Отличия от диаграмм последовательностей

- На диаграммах последовательностей поведение объектов рисуется снизу вверх, то есть, условно, по оси X откладываются объекты, по оси Y — время.
- На коммуникационных диаграммах объекты размещаются на двумерной плоскости, а порядок взаимодействия определяется числовыми индексами на стрелках.
  - они гораздо компактнее, но порядок действий во времени на них не очевиден.

Синтаксис



Сначала пользователь делает запрос `find_books()`, который обрабатывается объектом типа **Online Bookshop** (до двоеточия пишется имя объекта, если оно важно, после — имя типа, если оно важно).

Вызов `find_books()` приводит к вызову `search()`, `view_book()` и `add_to_cart()` в именно такой последовательности.

**Обратите внимание на нумерацию запросов: 1.1 означает, что это первый вызов внутри вызова 1, 1.3 — третий вызов внутри вызова 1.**

Дальше пользователь выполняет запрос `checkout()`, который обрабатывается с помощью ещё двух вызовов.

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

[конспект преза](#)

### Диаграммы составных структур

Относятся к **структурным** диаграммам.

Предназначены больше для проектирования аппаратного обеспечения, которое состоит из стандартизованных блоков, соединённых стандартными интерфейсами.

Демонстрируют внутреннюю структуру классов и, по возможности, взаимодействие элементов (частей) внутренней структуры класса.

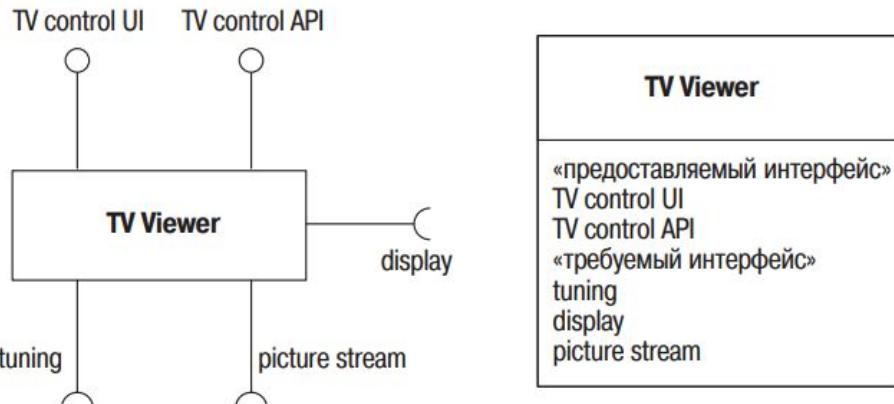
По сути, представляют собой продвинутые диаграммы компонентов.

#### Отличия от диаграмм компонентов

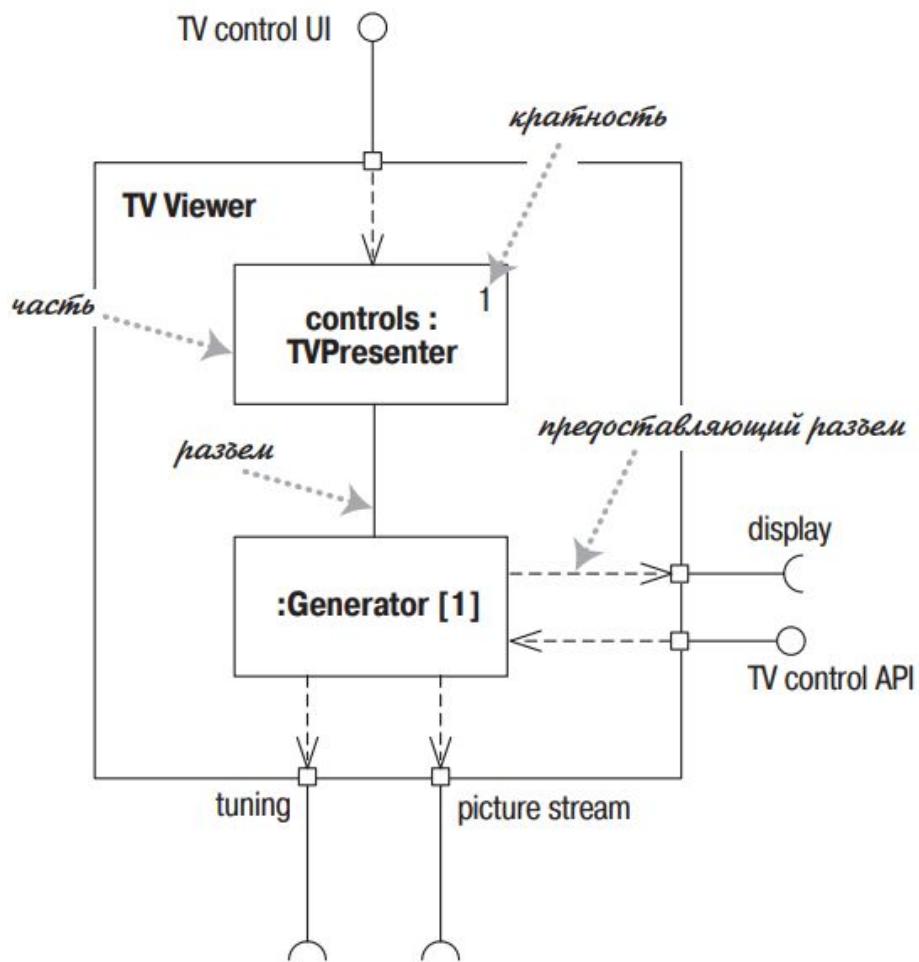
- диаграммы компонентов — это диаграммы, показывающие структуру времени компиляции
- диаграммы составных структур показывают структуру времени выполнения
- на диаграммах составных структур внутри объемлющей компоненты не другие компоненты, а **роли**
  - **роль** — это что-то вроде объекта, на диаграмме может быть несколько ролей одного типа, которые по-разному связаны друг с другом и делают разную работу

#### Синтаксис

Интерфейсы компонентов можно описывать как в шарово-гнездовой нотации, так и внутри символа компонента:



Еще небольшой пример:



Тут controls и :Generator — это роли, в качестве controls может выступать любой компонент, реализующий интерфейс TVPresenter.

[1] — это кратность компонента (и TVPresenter, и Generator в системе только один).

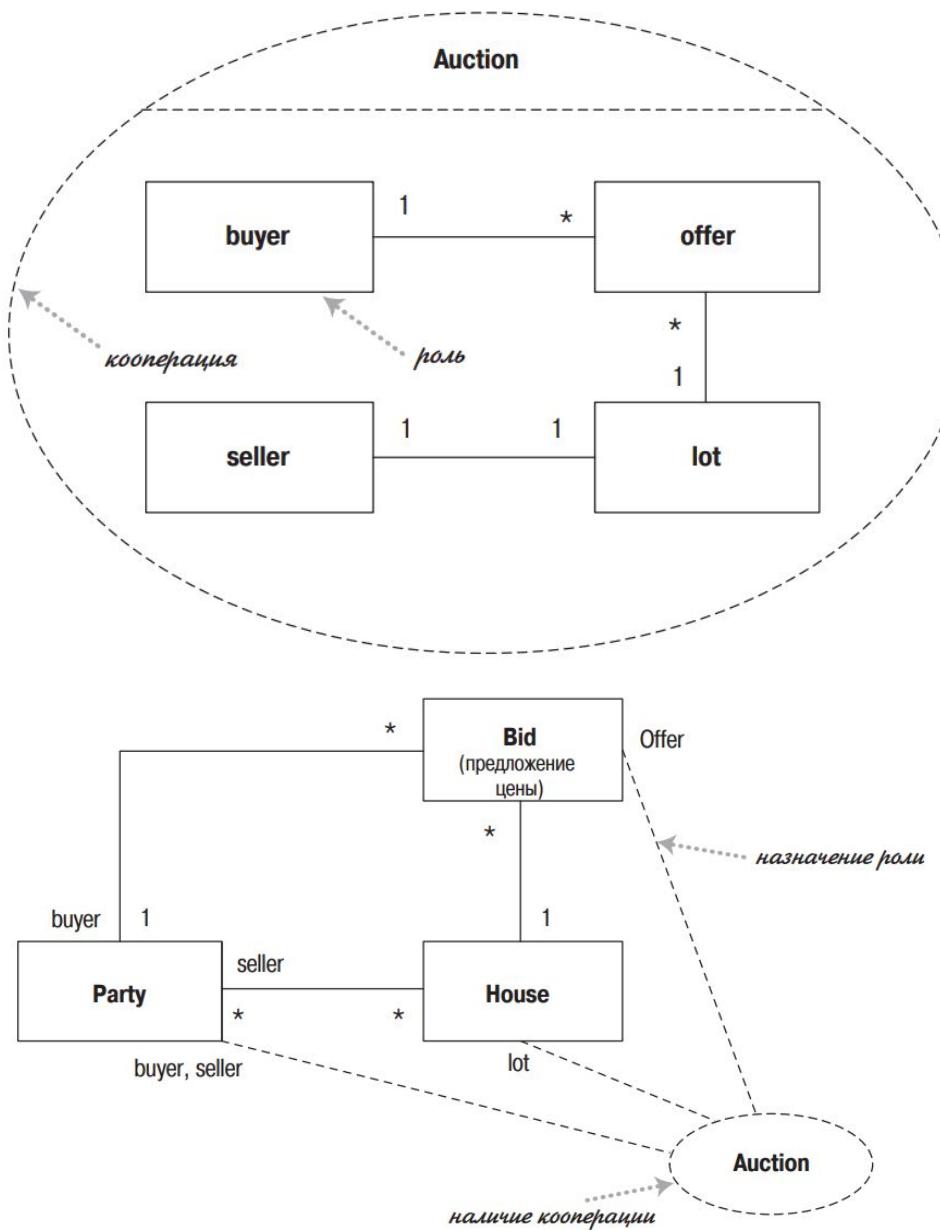
Порты, интерфейсы и связи тут в целом такие же, как на диаграмме компонентов (только связи называются “разъёмами”).

## Диаграммы коопераций

Относятся к **поведенческим** диаграммам.

Показывают взаимодействие между объектами (ролями) в рамках одного сценария использования. Редко встречаются на практике.

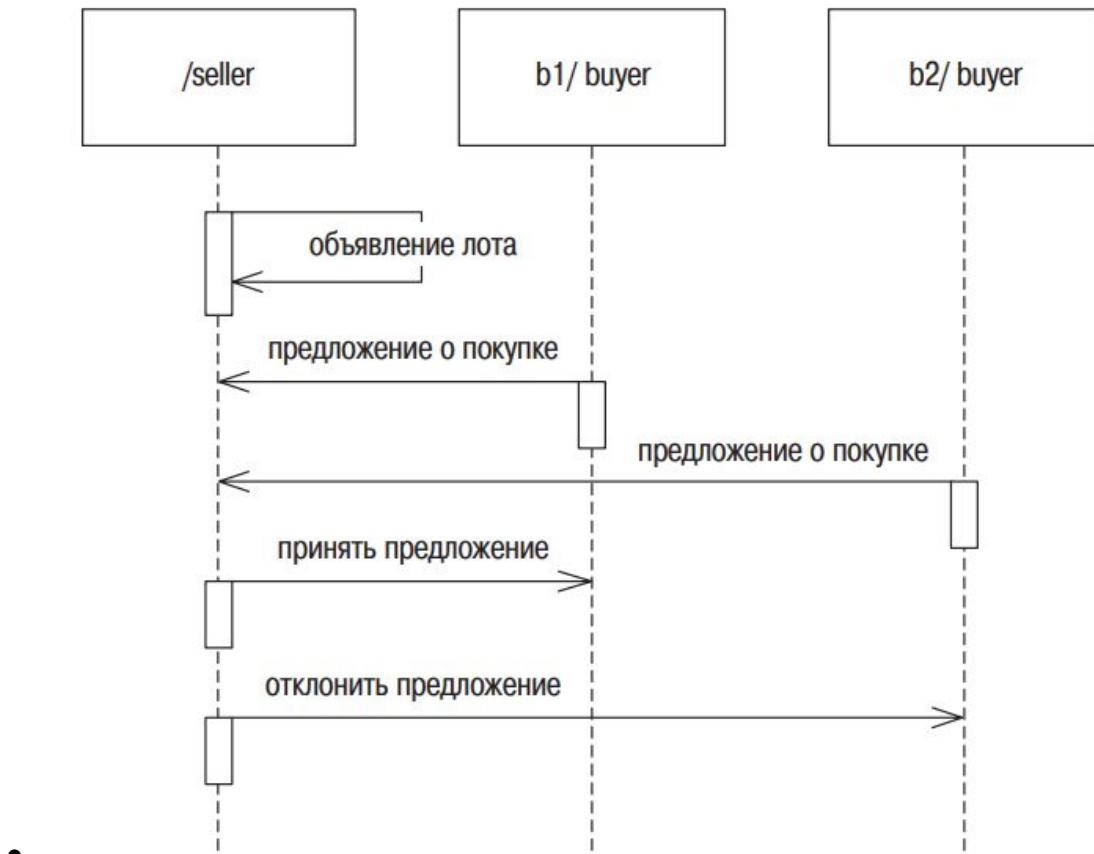
- Что-то среднее между диаграммами объектов и диаграммами классов.
- Вместо классов и объектов на них рисуются **роли** — сущности, на место которых может быть подставлен настоящий объект.
- Пример:



(это одно и то же в разной нотации)

Здесь показана кооперация в рамках сценария использования «Аукцион». Есть роли «покупатель», «продавец», которые могут взаимодействовать посредством лотов и предложений.

- Представляют простую структурную точку зрения на объекты, и могут дополняться диаграммами последовательностей для иллюстрации ещё и временных аспектов:



## Временные диаграммы

Относятся к **поведенческим** диаграммам.

Еще одна форма диаграмм взаимодействия, которая акцентирована на временных ограничениях: либо для одиночного объекта, либо для группы объектов.

Полезны для обозначения временных интервалов между изменениями состояний различных объектов. Кроме того, эти диаграммы знакомы инженерам по оборудованию.

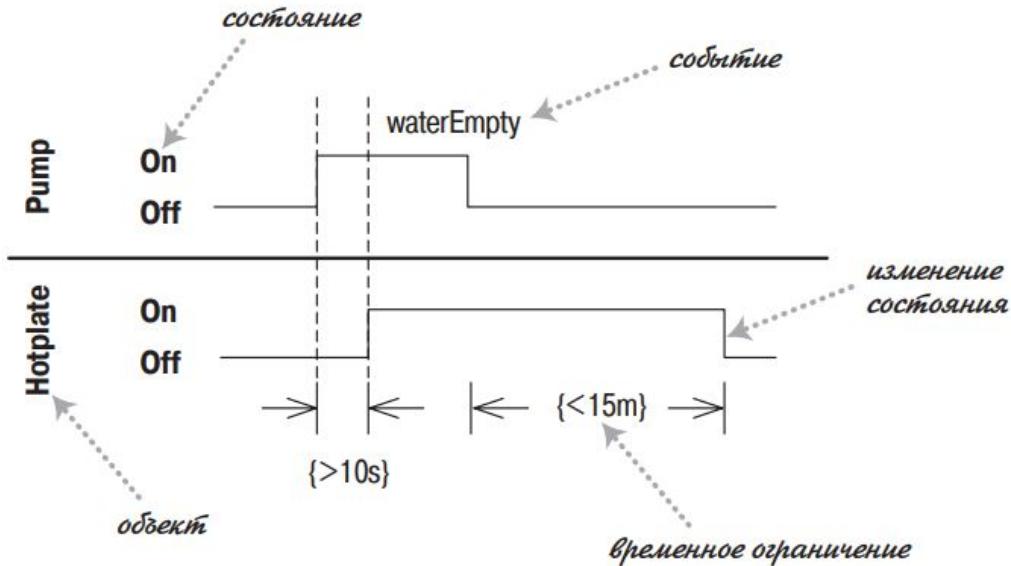
## Синтаксис

В обоих случаях рисуется временная шкала (время идёт слева направо), на ней вертикально располагаются объекты, которые могут находиться в некоторых состояниях.

Помимо состояний указываются и временные ограничения, в фигурных скобках, как принято в UML для ограничений.

### Вариант 1

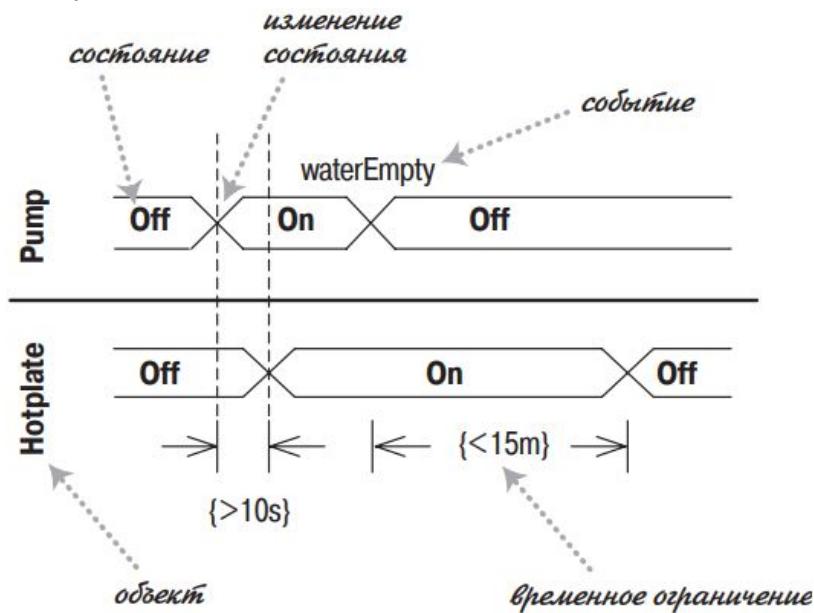
- показывает переключение состояния как скачок линии
- нагляднее



**Рис. 17.1.** Временная диаграмма, на которой состояния представлены в виде линий

### Вариант 2

- показывает переключение состояния как пересечение линий с именем состояния внутри
- удобнее, если состояний много



**Рис. 17.2.** Временная диаграмма, на которой состояния представлены в виде областей

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

[конспект преза](#)

Диаграммы обзора взаимодействия

Относятся к **поведенческим** диаграммам.

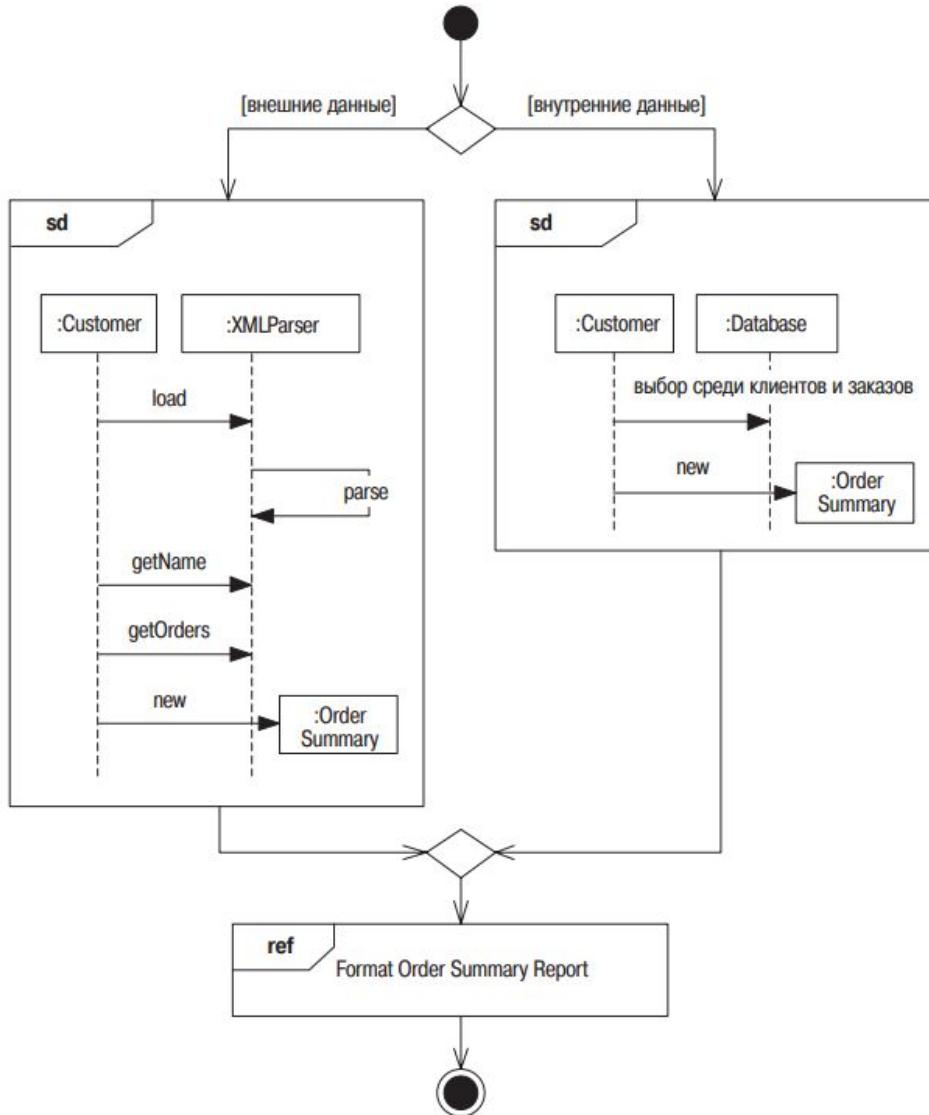
Появились в UML 2.

Представляют собой совмещённые на одной диаграмме элементы диаграммы последовательностей и диаграммы активностей.

Применяются такие диаграммы, когда есть сложный алгоритм и нужна диаграмма последовательностей, которая бы его визуализировала, но имеется куча ветвлений и циклов.

На диаграммах последовательностей есть фреймы, но для сколько-нибудь сложных алгоритмов они быстро сделают диаграмму нечитаемой.

## Синтаксис (на примере)



## Диаграммы ПОТОКОВ данных

НЕ ИЗ UML

Показывают обмен данными в системе.

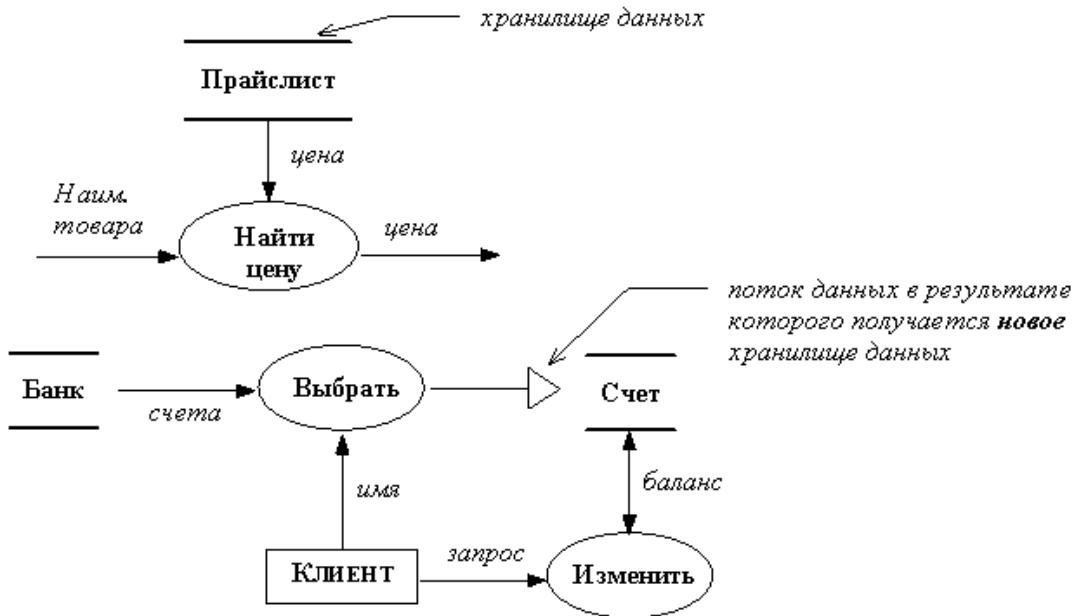
Полезны как первый набросок архитектуры системы.

## Синтаксис

Есть один вид стрелок, который показывает поток данных, и три вида сущностей, между которыми, собственно, ходят данные:

- **процесс** — то, что может как-то преобразовывать данные внутри проектируемой системы
  - рисуется кружком;
- **внешняя сущность** — то, что поставляет или потребляет данные
  - рисуется прямоугольником;

- **хранилище** — то, где данные могут лежать, куда их можно поместить и забрать при необходимости
  - рисуется двумя горизонтальными линиями.



## 20. Диаграммы IDEF0, характеристики. Feature tree.

[конспект преза](#) (лекция 4, почти все)

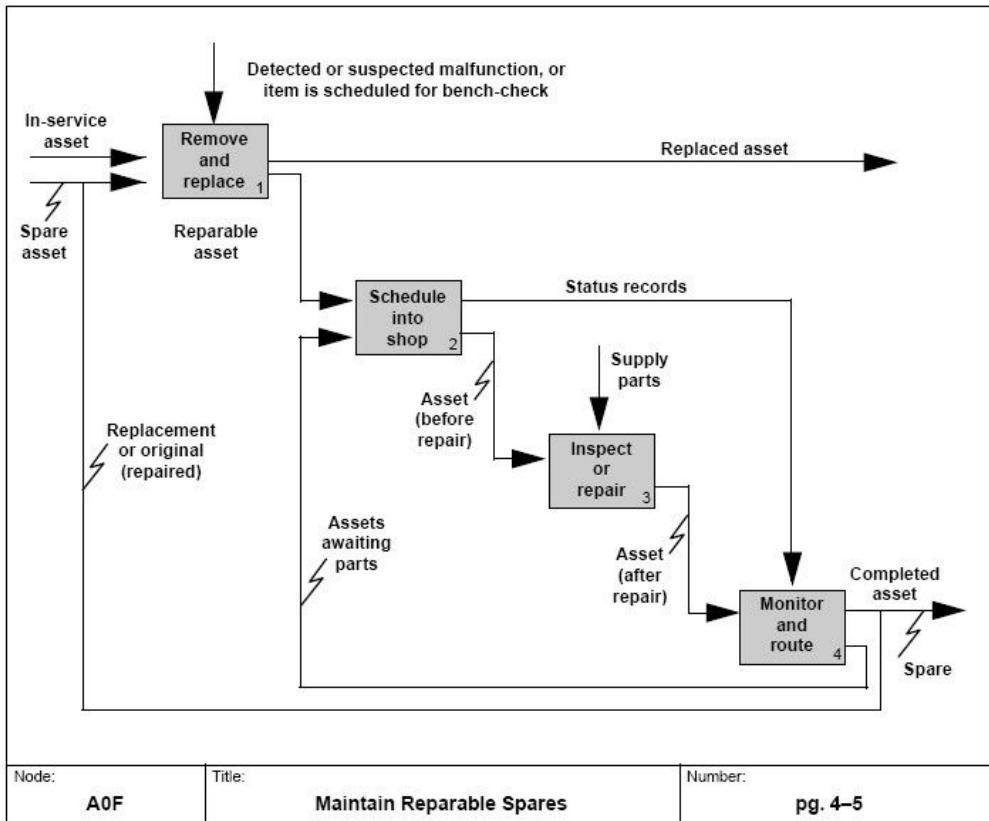
[конспект преза](#) (лекция 5, совсем мало про IDEF0)

### Диаграммы IDEF0

Нотация IDEF0 используется для структурной декомпозиции системы (или бизнес-процесса, в который система встраивается).

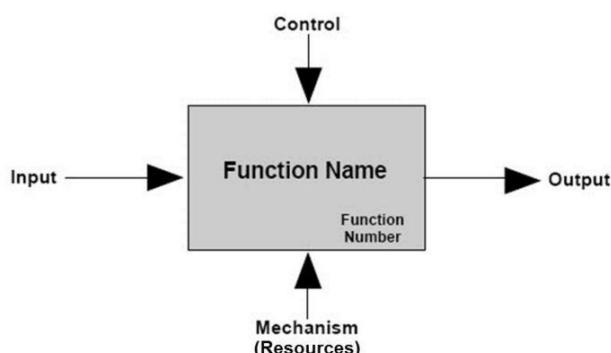
Чаще используются в анализе бизнес-процессов.

Процесс представляется в виде набора шагов, соединённых входами и выходами друг с другом.



Каждая сторона блока с шагом имеет свой смысл:

- Слева — входные данные или материалы
- Сверху — управление
  - то, что регламентирует работу блока и как-то управляет им
- Снизу — механизмы
  - то, что блок непосредственно не перерабатывает, но необходимо блоку для работы.
- Справа — выходные данные или продукты



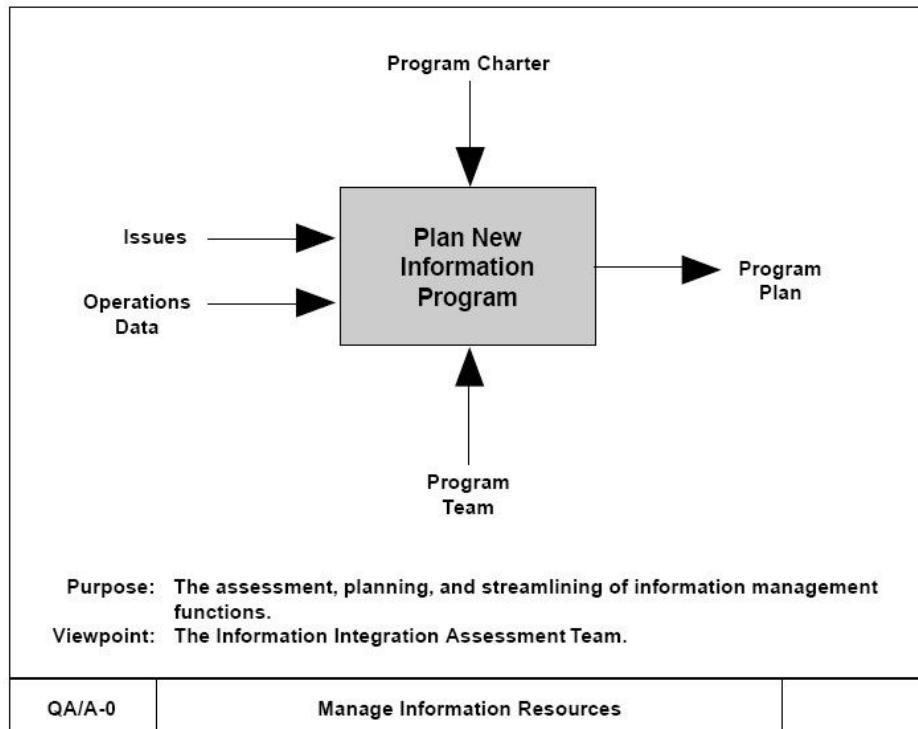
Каждый блок может быть далее раскрыт своей IDEF0-диаграммой,

- при этом все входы на исходной диаграмме должны быть входами и на диаграмме, раскрывающей блок

- аналогично с выходами

В корне иерархии диаграмм лежит **контекстная диаграмма IDEF0**, которая интересна в плане работы с требованиями.

- она позволяет чётко определить границу системы, её входы и выходы, так, чтобы это помещалось на одном экране.
- на ней рисуется всего один блок, означающий вообще всю систему.

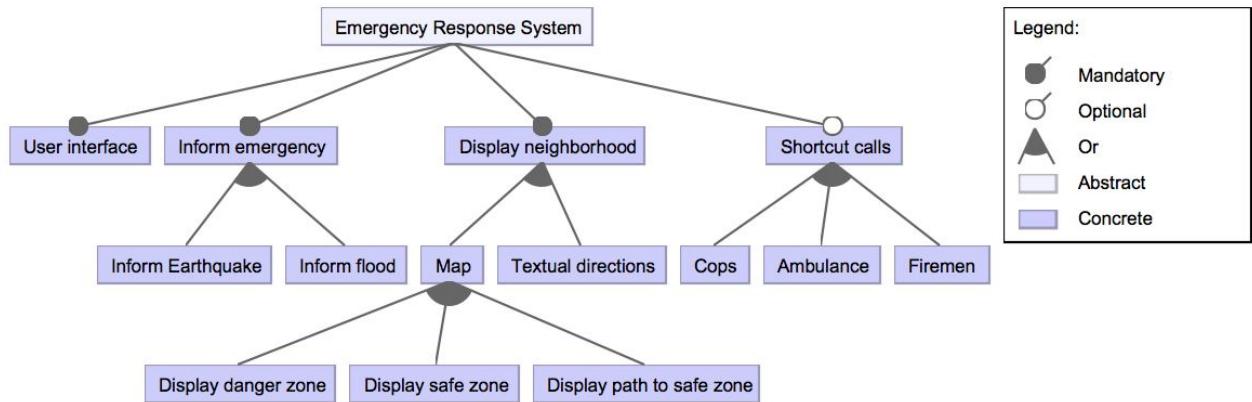


## Диаграмма характеристик

Требования обычно иерархичны, поэтому их удобнее всего в виде дерева. Чаще всего это делают текстом, в виде списков, но есть и визуальная нотация: **диаграмма характеристик (feature diagram)**.

- Для “повседневного” анализа требований диаграммы характеристик не очень удобны
- Они активно используются в разработке линеек программных продуктов (например, если у вас есть Professional и Ultimate-версии, то можно отметить на общей диаграмме характеристик, какие характеристики куда попадают)

## Синтаксис



### Характеристики бывают:

- абстрактными
  - надо ещё декомпозировать перед тем как реализовать
- конкретными
  - пригодными к реализации

### Отношения между характеристиками бывают:

- “или” (включающее и исключающее),
- обязательные
- необязательные
  - без необязательной характеристики продукт вполне может существовать

Feature tree / дерево требований / fishbone diagram

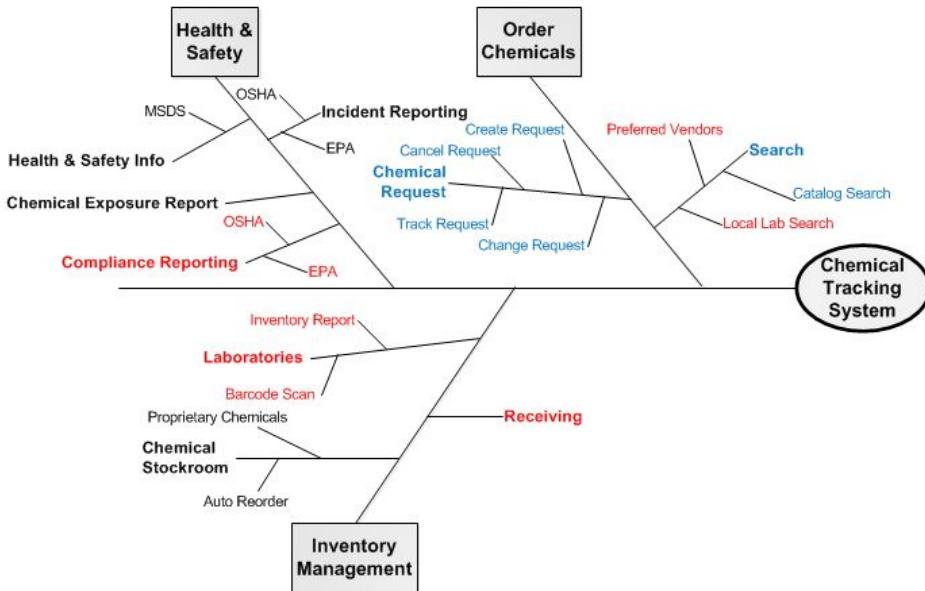
Более простая нотация, чем диаграмма характеристик.

На такой диаграмме никаких взаимосвязей не показать, но как инструмент первоначального анализа она может быть очень полезна.

### Синтаксис

“Голова рыбы” — это система в целом, от неё отходит “хребет”, от которого отходят основные фичи.

Они дальше детализируются более мелкими фичами, рисуемыми как ответвления от основных.



## 21. Моделирование требований в SysML.

[конспект преза](#)

**SysML** - графический язык моделирования.

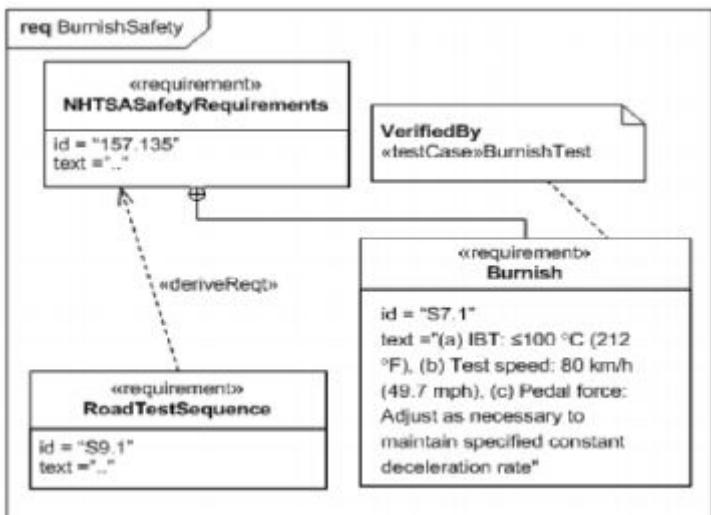
Изначально создавался как профиль (т.е. стандартное расширение) языка UML для моделирования больших систем, включавших в себя как программные, так и аппаратные компоненты, а также людей и другие системы.

Потом этот язык стал отдельным языком, туда добавили несколько новых видов диаграмм, в частности, **диаграмма требований** - более формальная нотация дерева требований.

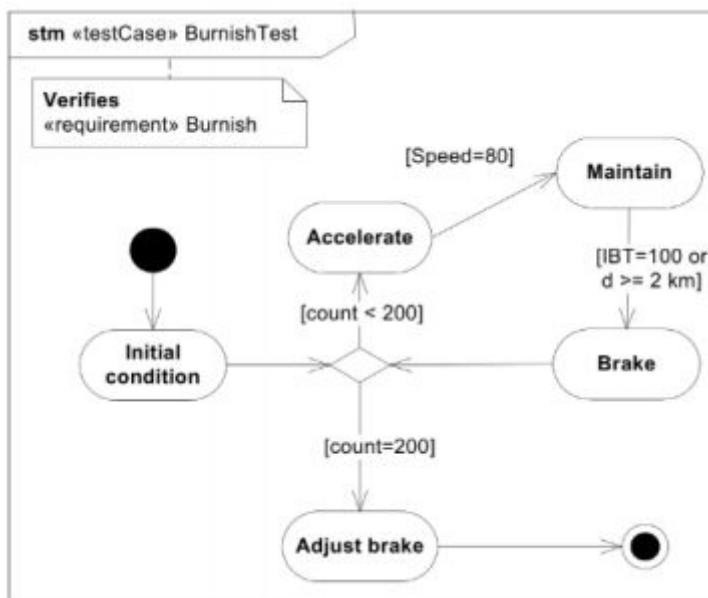
### Синтаксис

**Требования** рисуются похоже на классы, со стереотипом `<<requirement>>`, именем, идентификатором требования и текстовым описанием.

Требования могут находиться в разных **отношениях** (например, `<<deriveReqt>>`, означающее, что одно требование является уточнением другого, или `satisfy`, означающее, что какой-то элемент системы, например, класс, удовлетворяет или реализует то или иное требование).



Требования могут быть также уточнены **сценарием тестирования** (в SysML даже есть стандартные отношения <<verifies>>/<<verifiedBy>>). Сценарий тестирования может быть представлен в виде диаграммы активностей.



## 22. Язык BPMN.

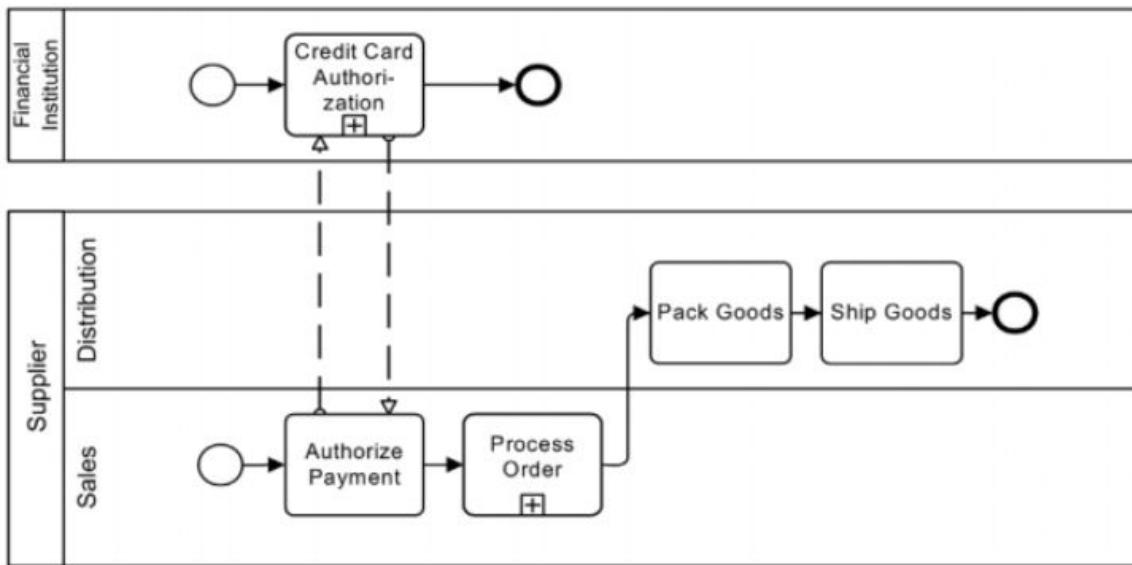
[конспект преза википедия](#)

**BPMN = Business Process Model and Notation.**

- Версия 1.0 в 2004 году, текущая (2.0) — в 2011
- Используется скорее аналитиками, а не архитекторами
- Для более продвинутого описания бизнес-процессов
  - Сильно продвинутые диаграммы активностей
  - Позволяют описывать группы взаимодействующих процессов
  - Исполнимая семантика
  - Правила генерации в BPEL

- Business Process Execution Language - настоящий исполнимый язык описания бизнес-процессов, который поддерживают многие движки-исполнители

## Диаграмма процессов



Тут изображены два бизнес-процесса двух независимых организаций, общающихся посылкой и приёмом сообщений.

Бизнес-процесс одной из организаций поделен на два раздела, но тем не менее, это один бизнес-процесс, на что указывает единый поток управления.

Почитаем подробнее про синтаксис на [википедии](#).

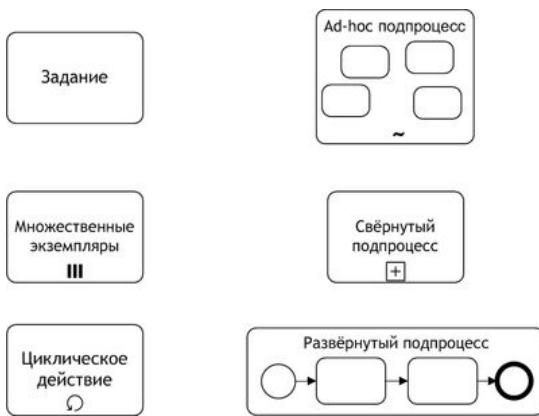
## Объекты потока управления

**События** изображаются окружностью и означают какое-либо произошествие в мире.

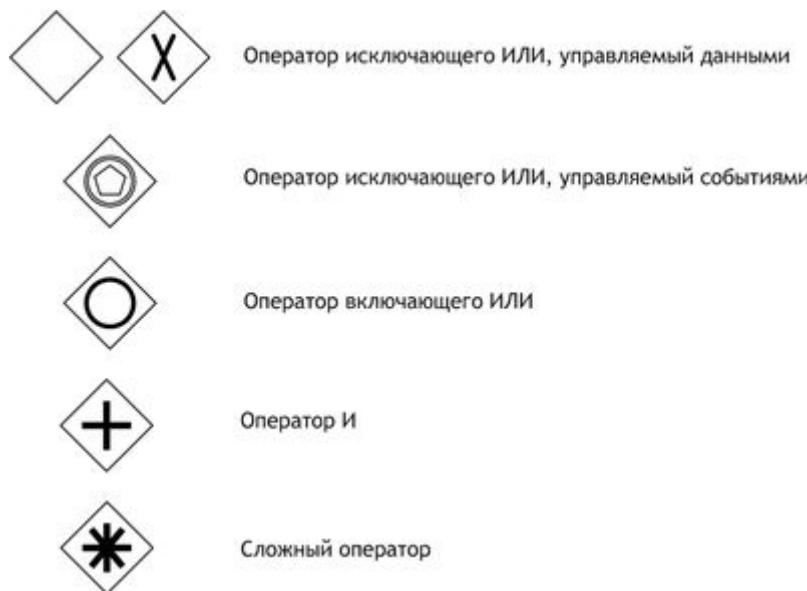
По расположению в процессе события могут быть классифицированы на начальные (англ. *start*), промежуточные (*intermediate*) и завершающие (*end*).

	Начальные	Промежуточные	Завершающие
	Обработка	Генерация	
Простое			
Сообщение			
Таймер			
Ошибка			
Отмена			
Компенсация			
Условие			
Сигнал			
Составное			
Ссылка			
Останов			

**Действия** изображаются прямоугольниками со скругленными углами. Среди действий различают задания и подпроцессы. Графическое изображение свёрнутого подпроцесса снабжено знаком плюс у нижней границы прямоугольника.



**Логические операторы (развилки)** изображаются ромбами и представляют точки принятия решений в процессе.



### Соединяющие объекты

Поток управления

Условный поток

Поток по умолчанию

Поток сообщений

Ненаправленная ассоциация

Направленная ассоциация

Двунаправленная ассоциация

### Роли

**Роли** — визуальный механизм организации различных действий в категории со сходной функциональностью. Существует два типа ролей:

- **Пулы** изображаются прямоугольником, который содержит несколько объектов потока управления, соединяющих объектов и артефактов.
- **Дорожки** представляют собой часть пула. Дорожки позволяют организовать объекты потока управления, связывающие объекты и артефакты.



Свёрнутый пул

## Артефакты

**Артефакты** позволяют разработчикам отображать дополнительную информацию в диаграмме. Это делает диаграмму более удобочитаемой и насыщенной информацией. Существуют три предопределённых вида артефактов:



Данные



Группа



Текстовая аннотация

## Диаграмма хореографии (choreography diagram)

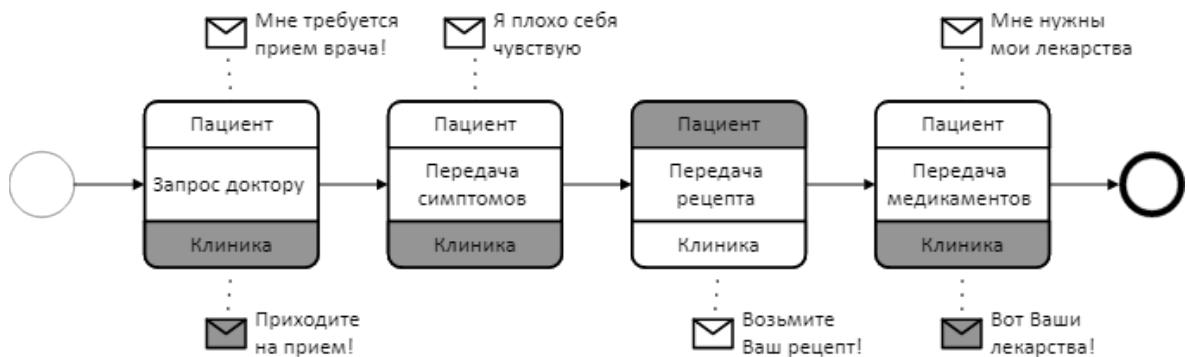
Описывает исключительно взаимодействие между бизнес-процессами.

Такая диаграмма нужна, если попытка изобразить процессы в дорожках на диаграмме процессов привела бы к хаосу из стрелочек.

### Синтаксис

Прямоугольники со скруглёнными углами — это **точки общения между процессами**, белым цветом выделен инициатор взаимодействия, серым — тот, кто должен ему ответить, конвертиком — сообщение или документ.

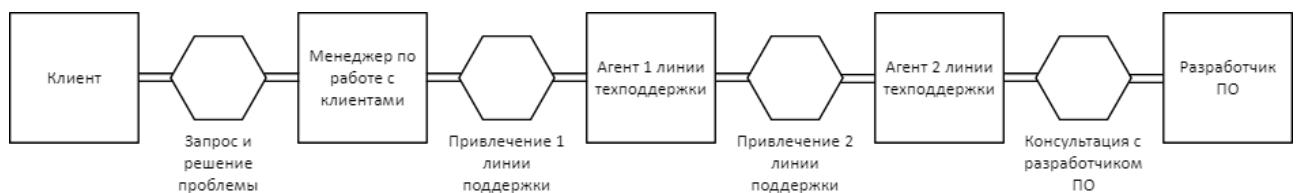
Ветвлениия и события тут рисуются, а активности — нет.



### Диаграмма диалогов (conversation diagram)

Показывает схему общения бизнес-процессов, не в смысле когда кто с кем общается, как диаграмма хореографии, а в смысле кто с кем в принципе может общаться.

### Синтаксис



Прямоугольниками показаны участники взаимодействия, двойными линиями с шестиугольниками — общение между участниками.

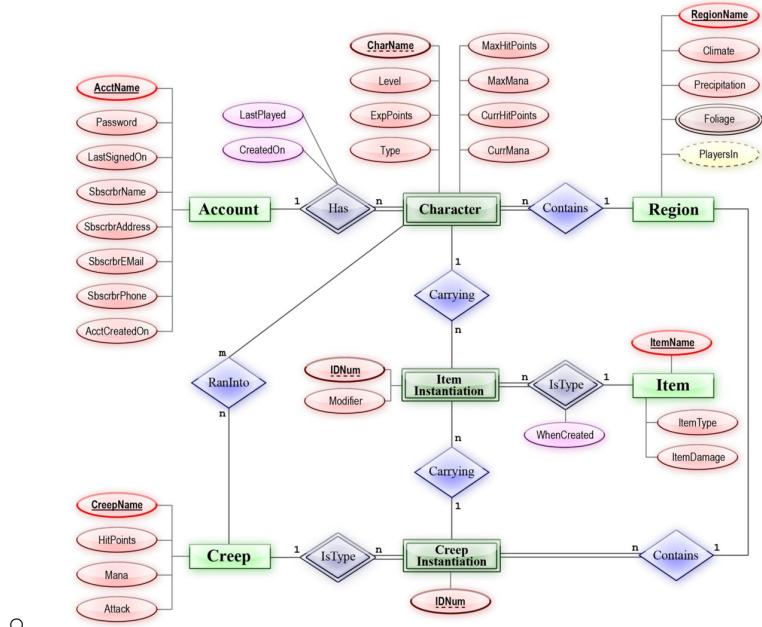
Плюсик внутри шестиугольника показывает, что взаимодействий между участниками много и они могут быть уточнены диаграммами хореографий или диаграммами процессов без деталей.

Три вертикальные черты внутри участника взаимодействия означают, что участников на самом деле может быть много.

## 23. Моделирование данных: диаграммы «Сущность-связь».

### конспект преза

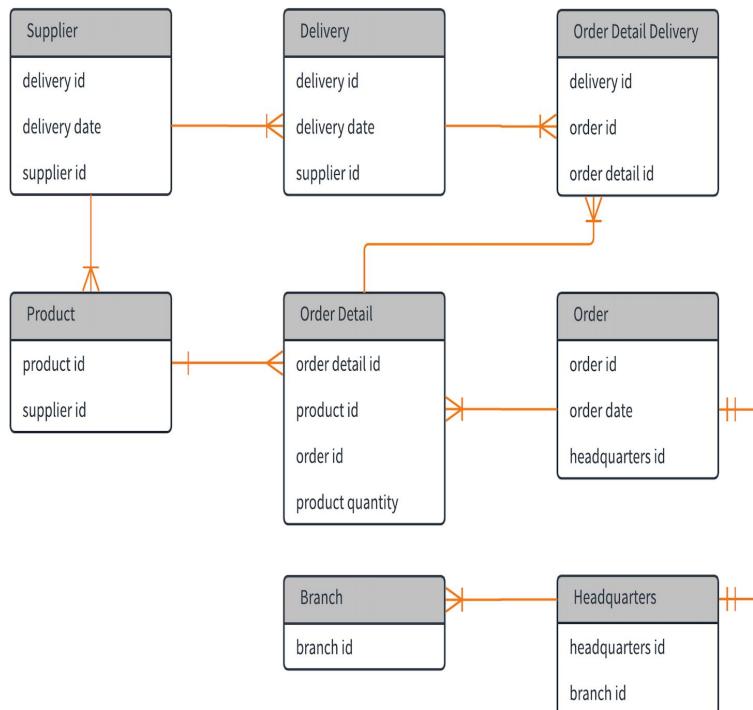
- Описывают концептуальную модель предметной области
- Идеальны для моделирования схем реляционных баз данных
- Есть несколько нотаций
- 1976 год, Питер Чен



- Сущности - прямоугольники
- Свойства - овалы
- Связи между сущностями - ромбы

- Могут иметь свойства
- Могут иметь кратность

- Разные варианты нотации «вороньей лапки»:



- Свойства рисуются прямо внутри сущности
- Название за обозначение множественности связи (три разветвляющиеся линии на конце связи, похожие на птичью ногу)
- Детали нотации различаются от инструмента к инструменту

## 24. Концептуальное моделирование, диаграммы ORM.

[конспект преза](#)

**Object-Role Modeling (ORM)** — это ещё одна нотация для построения концептуальной модели предметной области и, в конечном итоге, схемы базы данных проектируемой системы.

Используются реже ER-диаграмм

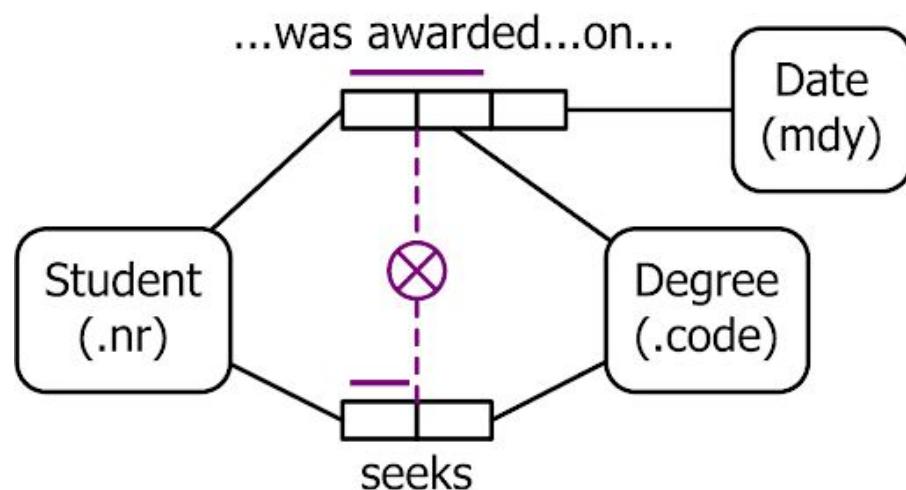
- существенно больше по размеру (каждый атрибут всё-таки надо моделировать как сущность)

Считываются удобнее на первых этапах анализа

- более устойчивы к изменениям в предметной области или нашего её понимания

ORM в программировании чаще используется в другом смысле — Object-Relational Mapping. Это библиотеки, которые представляют содержимое реляционной базы как набор настоящих объектно-ориентированных объектов, и наоборот. СЕЙЧАС НЕ ОБ ЭТОМ

Синтаксис



- Нет свойств, есть только сущности и связи
- Сущность имеет схему идентификации (её единственный атрибут — признак, по которому её можно отличить от других), это то, что пишется в скобках
- Связи играют роль таблиц в реляционной модели — каждая связь представляет собой кортеж из нескольких сущностей
  - Связи часто п-арные и могут быть соединены ограничениями
  - Сущности играют в связи роли, обозначаемые прямоугольниками посреди связи

- Над связью пишется вариант прочтения, например, «Студент такой-то получил степень такую-то»
- Горизонтальные и вертикальные полосы над ролями показывают, какие сочетания сущностей в данной связи должны быть уникальны.

## 25. Понятие и примеры CASE-систем.

[конспект преза](#)

**CASE-системы (Computer-Aided Software Engineering)** - инструментальные средства, которые позволяют пользоваться визуальными языками и моделировать.

- В 80-е годы термином CASE называли всё, что помогает разрабатывать ПО с помощью компьютера
  - Даже текстовые редакторы
- Теперь — прежде всего средства для визуального моделирования (UML-диаграммы, ER-диаграммы и т.д.)
- Отличаются от графических редакторов тем, что “понимают”, что в них рисуют
  - они знают, что класс — это класс, у него есть имя, поля и методы (а не просто текст на фигуре)
  - они знают синтаксис объявления полей и методов
  - они в целом знают синтаксис диаграмм и будут ругаться, если вы делаете что-то противоестественное

### Типичная функциональность CASE-инструментов

- Набор визуальных редакторов
  - чтобы рисовать диаграммы
- Репозиторий
  - централизованное хранилище информации о моделях
- Набор генераторов
  - генерируют код по моделям (как правило, не исполнимый, а заглушки)
- Текстовый редактор
  - для редактирования сгенерированного кода или документации
- Редактор форм
- Средства обратного проектирования (reverse engineering)
  - парсеры, которые могут преобразовывать код на разных языках программирования (или байт-коды) в диаграммы
- Средства верификации и анализа моделей
- Средства эмуляции и отладки
- Средства обеспечения командной разработки
  - git или svn тут не подходят, потому что они не могут адекватно показывать различия

- API для интеграции с другими инструментами
- Библиотеки шаблонов и примеров

## Примеры CASE-инструментов

- “Рисовалки”

Используются прежде всего как графические редакторы, специально заточенные под рисование диаграмм.

- Microsoft Visio
- Dia
- SmartDraw
- LucidChart
- Creately

- Полноценные CASE-системы

То самое, про что шла речь в предыдущем разделе.

- Enterprise Architect
- Rational Software Architect
- MagicDraw
- Visual Paradigm
- GenMyModel

- Забавные штуки

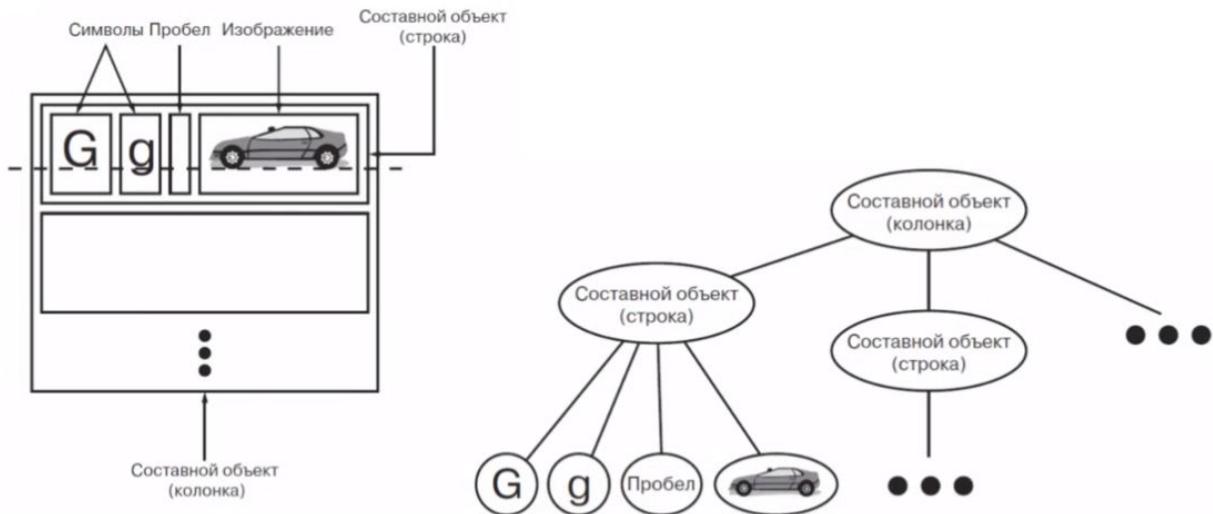
Направлены в основном на быстрое иллюстрирование документации или веб-страниц чем-то, похожим на UML-диаграммы, позволяют текстом описать, что надо нарисовать.

- <https://www.websequencediagrams.com/>
- <http://yuml.me/>
- <http://plantuml.com/>

## 26. Паттерн «Компоновщик».

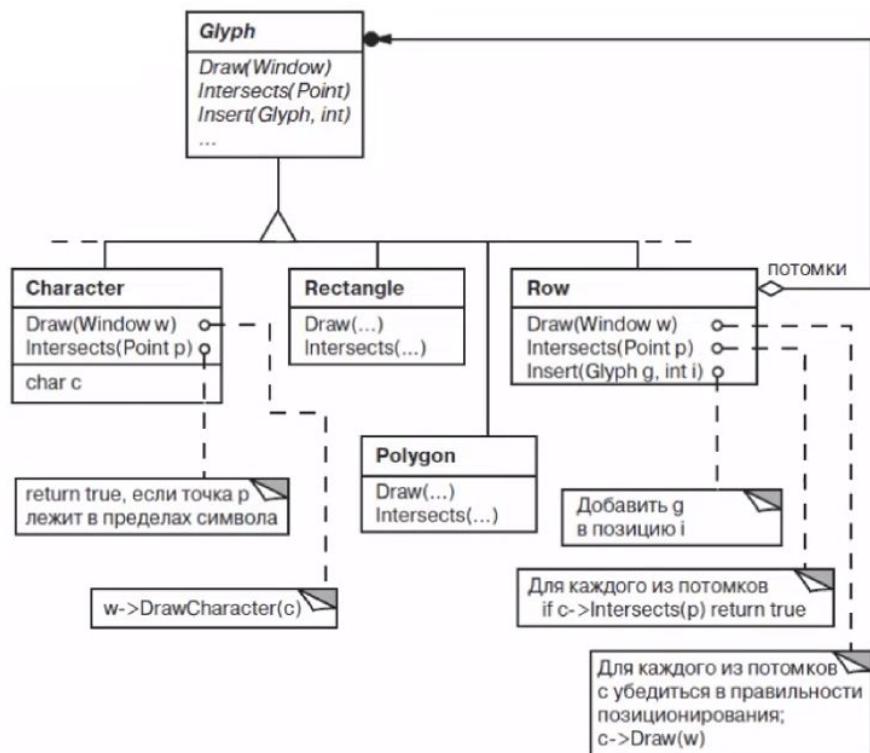
Паттерн “Компоновщик”

### Рекурсивная композиция



Предлагается использовать древовидное представление. Выглядит это примерно так, как показано на слайде выше.

### Диаграмма классов: глифы

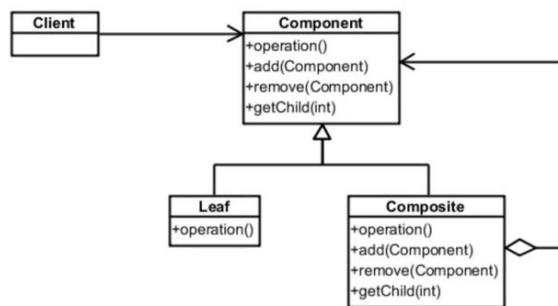


В диаграмме классов это могло бы выглядеть подобным образом. Глиф - это все, что можно нарисовать на экране. От него наследуются всякие базовые штуки (символ, прямоугольник, многоугольник...), а также составной глиф строки.

## Паттерн “Компоновщик”

Composite

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



Обобщается это все до такого вот состояния, как показано на слайде выше. Паттерн, который позволяет представить иерархию объектов в виде часть-целое. Позволяет обрабатывать простые составные объекты единообразно.

Существует Component (базовый класс) от которого наследуются либо элементарные объекты, либо Component`ы, которые состоят из одного или более Component. Паттерн на практике используется повсеместно.

Например, все синтаксические деревья в компиляторах устроены именно так. Обычно есть базовый класс нод (node???), в котором хранится привязка элемента, информация о типе и т.д., а от него наследуются конкретные Statement`ы, операции, операторы, и все остальное, что нужно для написания синтаксического анализатора. Файловые системы устроены подобным образом.

Не очень хорошо выражается в объектно-ориентированных языках.

## “Компоновщик” (Composite), детали реализации

- ▶ Ссылка на родителя
  - ▶ Может быть полезна для простоты обхода
  - ▶ “Цепочка обязанностей”
  - ▶ Но дополнительный инвариант
  - ▶ Обычно реализуется в Component
- ▶ Разделяемые поддеревья 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 и в Composite, тогда придётся делать cast
    - ▶ Иначе надо бросать исключения в add и remove

Тонкости внутреннего устройства:

Ссылка на родителя. Необходимо подумать нужна ли она, и так необходимо подумать в каждом конкретном случае. В большинстве случаев ссылка на родителя оказывается ненужной, т.к. большая часть операций над компоновщиком выполняется рекурсивно, и если нам всё-таки нужен родитель, мы можем передавать родителя в качестве параметра рекурсивного вызова. Ссылка на родителя может быть полезна для простоты обхода дерева. Для того чтобы уметь реализовать паттерн “цепочка обязанностей”. Например, так устроены графические библиотеки. Экранная форма состоит из контроллов. События происходящие с контроллом могут образоваться по цепочке от потомка к предку по иерархии компоновщика. Таким образом если каждый контрол знает ссылку на своего родителя, он может легко понять, что с ним что-то случилось. Например, по нему кликнули, истек таймер. Там это реально удобно. Тем не менее это дополнительный вариант, который необходимо поддерживать. Иногда его поддерживать довольно хлопотно, особенно, если эта иерархия часто меняется, поэтому ссылка родителя это весьма ситуационное решение.

Разделяемые деревья и листья. Позволяют сильно экономить память, особенно, если структура достаточно сложная, состоит из повторяющихся элементов. Например, с помощью разделяемых деревьев часто реализуются иммутабельные (немутабельные???) структуры данных. Операции удаления, добавления в таких иммутабельных списках не приводят к квадратичному росту памяти в зависимости от числа операций, а позволяет переиспользовать уже существующую. Тоже самое с компиляторами, там иногда бывает полезно иметь разделяемые куски синтаксического дерева для того, чтобы их можно было переиспользовать в разных фрагментах программы с целью экономии памяти.

При реализации всех схем с разделяющими деревьями надо помнить о том, что у вас в таком случае не может быть разделяемого состояния, вам не получится хранить ссылку на родителя, так как вы не знаете откуда вы пришли. С разделяемыми состояниями может помочь паттерн “приспособленец”. Все состояния, которое не относится к разделяемому поддереву можно просто вынести и хранить его вне, а общее состояние оставить в

поддереве. Также есть проблемы связанные с удалением памяти. Литвинов сказал, что на практике не заморачивался с разделением поддеревьев.

Концептуальная и идеологическая проблема паттерна компоновщик состоит в том, что операции для работы с потомками(добавить, удалить потомка...) они не имеют смысла для листа. Предок (компонент) навязывает потомку (листу) операции, которые лист выполнить не может. Из-за этого это плохой объектно-ориентированный дизайн. В данном случае с этим ничего не поделаешь так как мы хотим чтобы все операции выполнялись единообразно, в том числе операция, которая требует добавления потомков, можно считать, что лист это компонент у которого всегда 0 потомков.

Можно не объявлять операции с потомками в компоненте, если кому надо он может отcast`ить композит к тому, что нужно и уже выполнить операции. Так часто делают в синтаксических анализаторах. Обычно единообразная работа с каждым листом не требуется. Дерево обходится с помощью визитора, который в каждый момент знает с каким конкретно типом он имеет дело, поэтому не требуется оценивать в ходе операции добавления, удаления потому что для каждого узла они могут быть свои и это не составляет никакой проблемы. Для текстового редактора проблема есть, так как мы не хотим специально разбираться с каждым элементом, какого он типа, мы хотим просто иметь возможность быстро его обходить, быстро добавлять элементы, быстро удалять. Так что в текстовом редакторе лучше реализовывать add remove в компоненте и бросать исключения из их реализации в листе.

## “Компоновщик”, детали реализации (2)

- ▶ Операция getComposite() – более аккуратный аналог cast-а
- ▶ Где определять список потомков
  - ▶ В Composite, экономия памяти
  - ▶ В Component, единообразие операций
  - ▶ “Список” вполне может быть хеш-таблицей, деревом или чем угодно
- ▶ Порядок потомков может быть важен, может нет
- ▶ Кеширование информации для обхода или поиска
  - ▶ Например, кеширование ограничивающих прямоугольников для фрагментов картинки
  - ▶ Инвалидация кеша
- ▶ Удаление потомков
  - ▶ Если нет сборки мусора, то лучше в Composite
  - ▶ Следует опасаться разделяемых листьев/поддеревьев

Можно сделать операцию GetComposite(), которая у листа возвращает Null, а у Composite`а будет возвращать отcast`анный Composite. Это более аккуратный аналог cast`а, который позволяет меньше боли испытывать при работе с такой структурой.

Проблема: где определять список потомков. В Component для единообразия операций либо в Composite, чтобы листы не наследовали пустые списки (экономия памяти). В Composite предпочтительнее. Обратите внимание, что потомки могут быть устроены довольно хитро, вполне может быть хеш-таблица, дерево и т.д.. Обычно это особого смысла не имеет, но есть хорошие примеры, такие как windows presentation foundation библиотека для разработки графических интерфейсов, очень известна .net библиотека. Она хранит некоторых из потомков в виде хеш-таблицы.

Еще над чем стоит задуматься на этапе проектирования, это порядок потомков. Он может быть важен, а может быть и не важен. Это важно учитывать при операциях сравнения композитов, возможно, что для одного типа конкретных узлов порядок важен, а для другого нет. Например, в синтаксических деревьях имеет смысл помнить порядок, в котором принимаются параметры функции, но может не иметь смысла, в каком порядке применяются модификаторы при объявлении функции(`public static` или `static public`). Если порядок потомков вдруг оказывается неважным, это существенно запутывает работу с таким деревом.

Полезным приемом является кэширование информации. Например, для нашего текстового редактора можно использовать кэширование ограничивающих прямоугольников вокруг каждого глифа, для того чтобы иметь возможность быстро проверять попали ли мы мышко в тот глиф или не попали. Спрашивать потомков о том куда именно попали, только в том случае когда мы попали в родительский глиф. Обычная с кэширование проблема это инвалидация кэша. Нужно нотифицировать всех родителей каждого конкретного элемента дерева о том, в это время что-то изменилось, чтобы они сбросили кэш, а потом перекэшировали себя.

Удаление потомков. Если есть сборщик мусора то, об этом обычно даже не надо думать. Если сборщика мусора нет, то приходится реализовывать его вручную в Composite, также следует помнить о разделяемые листья или поддеревья. Если гарантируете, что ваши деревья всегда принадлежат конкретному композиту, то можете смело их удалять, если не можете гарантировать, то вам следует использовать что-то типа умных указателей с подсчетом ссылок.

## 27. Паттерн «Декоратор».

Паттерн “Декоратор”

### Усовершенствование UI

- ▶ Хотим сделать рамку вокруг текста и полосы прокрутки, отключаемые по опции
- ▶ Желательно убирать и добавлять элементы обрамления так, чтобы другие объекты даже не знали, что они есть
- ▶ Хотим менять во время выполнения — наследование не подойдёт
  - ▶ Наш выбор — композиция
  - ▶ Прозрачное обрамление

В нашем примере мы хотим использовать паттерн “Декоратор” для создания красивого пользовательского интерфейса. Например, мы хотим сделать рамочку вокруг текста и полосу прокрутки, которая позволяет текст просматривать, но так чтобы они отличались по опции. Полосы прокрутки могут исчезать, появляться в зависимости от размера документа.

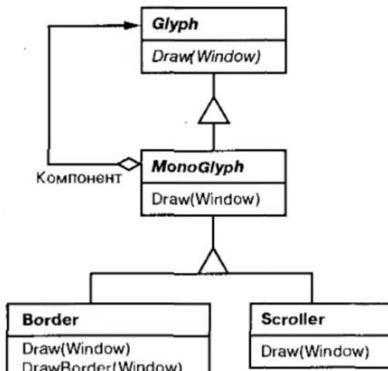
Мы хотим чтобы это было прозрачно по отношению ко всем остальным элементам текстового редактора, например, мы можем добавлять рамку вокруг каждого конкретного глифа, как отдельного так и составного, например, выделить строку и сказать обвести ее в рамочку. Возможно, мы хотим иметь полосу прокрутки внутри глифа, например, если у нас есть большая таблица, внутри которой необходимо что-то скролить, возможно, было бы круто это сделать. Хотим рассматривать весь интерфейс нашего пользовательского редактора, как набор глифов. Идея такого дизайна проистекает из языка sno tek. На примерах из этого языка была написана первое издание книжки про паттерны. Язык был очень популярным в начале 90-х, является предшественником javascript. Язык динамической типизации, который был интересен тем, что был настолько интерпретируемым, что даже сама среда разработки была написана на нём, в нее можно было влезть программно даже во время написания программы и что-то с ней сделать. Можно было переиспользовать среду разработки для создания интерфейса своих программ. Язык очень быстро вышел из моды, но некоторые идеи, как мы видим, из него остались. Хотим сделать такой редактор, который позволял бы свой пользовательский интерфейс использовать внутри документов и наоборот. В общем случае это плохая идея, так как вы можете случайно что-то сломать.

Хотим менять элементы оформления прямо во время выполнения, при этом использовать что у нас есть текст и у нас есть текст в рамочке, и это два разных класса у нас не получится. Из-за этого вместо наследования мы будем использовать композицию. Они являются взаимозаменяемыми вещами. Композиция в нашем случае хороша, так как позволяет переконфигурировать систему. Сделаем композицию таким образом, чтобы она была незаметна для окружающего, то есть чтобы объект, который внутри себя содержит другой объект, мог прикидываться обычным глифом.

Паттерн “Декоратор”

## Моноглиф

- ▶ Абстрактный класс с ровно одним сыном
  - ▶ Вырожденный случай компоновщика
- ▶ “Обрамляет” сына, добавляя новую функциональность



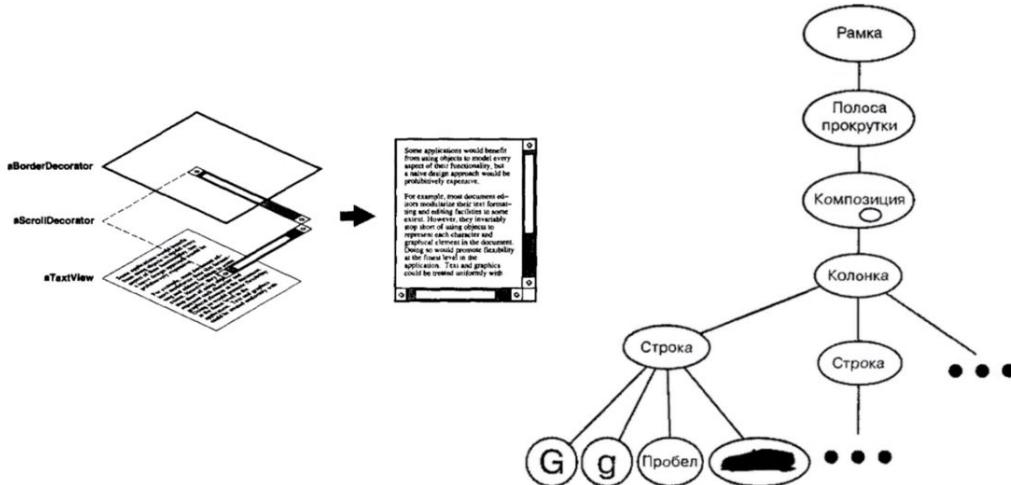
© Э. Гамма и др., Приемы  
объектно-ориентированного  
проектирования

Именно для этого заведен класс моноглиф, который будет хранить в себе класс глиф. Это будет абстрактный класс, который в чем-то похож на композит из паттерна компоновщик, но у него всего один сын. Паттерн декоратор можно рассматривать как вырожденный случай компоновщика, где есть ограничение, что у каждого композита может быть не более одного

сына. Моноглиф в отличие от соответствующих классов паттерна компоновщик, служит не для того чтобы собрать их воедино, а служит для того чтобы добавить новую функциональность тому объекту, который внутри него находится. Например, у нас есть глиф от него наследуется моноглиф, от него наследуется border и scroller. Border и scroller реализуют метод Draw так чтобы сначала нарисовать себя, а потом то, что внутри них. На самом деле в общем случае можно и наоборот, но это вопрос, который решается в каждом конкретном случае отдельно.

Паттерн "Декоратор"

## Структура глифов

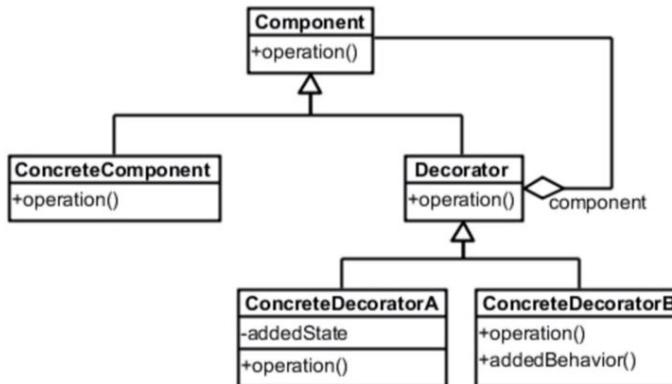


© Э. Гамма и др., Приемы объектно-ориентированного проектирования

Наш документ может выглядеть следующим образом. Наш обычный текстовый документ, на него поверх можно надеть декоратор, который будет рисовать полосы прокрутки, и декоратор, который будет рисовать рамочку. Обратите внимание, что в таком случае у нас будет рамочка внутри полосы прокрутки, а если поменяем местами у нас будет полоса прокрутки, а внутри нее документ в рамке. В нашем случае это хорошо, так как обеспечивает некоторую гибкость.

# Паттерн “Декоратор”

Decorator



Обобщается эта конструкция до паттерна декоратор, который устроен вот таким вот образом. У нас есть базовый класс Компонент, у которого есть некоторый набор операций. Компонент реализует какие-то конкретные компоненты, их может быть много. Есть класс декоратор, который внутри себя содержит поле типа компонент и есть конкретные декораторы, которые обязаны реализовывать тот же самый интерфейс, что и компонент, но при этом могут добавлять либо новое поведение, либо новое состояние. При этом прорасывая запросы тому настоящему компоненту, который внутри них содержится, это позволяет декоратору надеваться не только на конкретный компонент, но и на другие декораторы, выстраивая такие вот цепочки из декораций.

## Декоратор, особенности

- ▶ Динамическое добавление (и удаление) обязанностей объектов
  - ▶ Большая гибкость, чем у наследования
- ▶ Позволяет избежать перегруженных функциональностью базовых классов
- ▶ Много мелких объектов

Чем хороши декораторы?

Можно прям в рантайме добавлять новые обязанности объектам, удалять новые и старые обязанности, обеспечивая тем самым существенно большую гибкость чем у наследования. Декоратор позволяет не писать много кода в обычных классах, мы могли бы поддержать в классе, который отвечает за отображение текста еще и рисование рамочек, полос прокрутки, но получится монструозный класс, в котором ничего не понятно.

Декоратор позволяет разбить функциональность на много разных классов и как-то эту функциональность инкапсулировать так чтобы она была более менее самодостаточной.

Чем плохи декораторы?

Большое количество мелких объектов, не очень эффективное использование памяти. Если вы увлечетесь декораторами то, у вас вместо одного нормального объекта будет 150 разных объектов, которые декорируют друг друга хитрым образом.

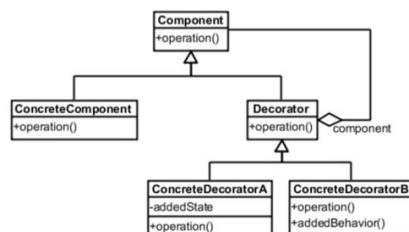
Во-первых, будет не очень понятно кто кого декорирует, кто за кого отвечает.

Во-вторых, все объекты имеют оверхед на хранение разного рода метаинформации, типа таблицы виртуальных методов и так далее, поэтому это занимает память.

В-третьих, и наверное, самое главное, у вас будет оверхед на вызов почти любой функции, потому что каждая функция декоратора должна быть виртуальной, в силу того, что он не знает декорирует он настоящий объект или это и есть настоящий объект, а также вызывать их надо будет по цепочке, получим стек вызовов из 150 элементов, где-то в глубине которого находится объект, который по настоящему делает работу.

## “Декоратор” (Decorator), детали реализации

- ▶ 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 неё



Во-первых, декоратор должен реализовывать интерфейс декорируемого объекта. Есть схемы, которые позволяют выводить запросы к объектам с другим интерфейсом, но это другой паттерн, паттерн “Адаптер”. **На экзамене будут вопросы о том, чем один паттерн отличается от другого.** Замечание, которое касается не только паттерн Декоратор, но и всех паттернов, это то, что не надо увлекаться с абстракциями, если у вас всего один конкретный класс декоратор, то абстрактного декоратора можно не делать, тоже самое касается и компоновщика и всех остальных паттернов. Не делайте классы только для того чтобы они были, или потому что в книжке так написано. Абстрактный класс нужен, если вы хотите воспользоваться полиморфизмом.

Декоратор не очень удобно реализовывать, если компонент имеет развитый интерфейс. Если у компонента слишком много методов, то в декораторе вы будете вынуждены отдельно реализовывать эти методы, вручную вызывая методы компонента. Желательно чтобы компонент был интерфейсом, а не каким-то конкретным классом. Хотя это не так важно, как то что у него должно быть не очень много методов. Даже если их около 10, стоит задуматься не использовать ли какую-то другую схему расширения. Например, паттерн стратегия. Стратегия это в каком-то смысле декоратор наоборот. Вы не надеваете новую функциональность на объект, а вставляете новую функциональность в объект и он ей пользуется. В качестве самодельного аналога паттерна стратегия, который тоже хорошо подходит вместо декоратора, если у вас слишком развитый интерфейс в классах компонент. Это список расширений, которые вызываются вручную компонентом перед операцией либо

после операции. Так делают многие оконные библиотеки. Эти расширения обычно называются хуками или зацепками. Представляют из себя обычно виртуальные функции с пустыми реализациями в базовом классе, и при необходимости можно эти реализации переопределить, добавив какие-то свои действия либо перед операцией, либо после операции. Например нарисовать рамочку можно таким образом. В каком-нибудь классе текста вы описываете before draw либо after draw. Draw делает свое дело, а потом в after draw вы рисуете рамочку вокруг того что получили. С одной стороны это хорошо, и вы даете пользователям вашей системы реализовать какие-то свои операции, при этом вы определяете конкретные места куда они могут встраиваться. Декоратор в этом плане хуже, так как дает возможно делать что угодно, любую операцию переопределить, любую операцию как угодно задекорировать, а такую неопределенность не очень любят разработчики. Они начинают думать как это можно сделать лучше, а если это можно сделать несколькими разными способами, то это вообще беда. С другой стороны подход с расширениями плох тем, что вам вам заранее придется подумать обо всех возможных точках расширения вашей системы.

Паттерн декоратор можно использовать в синтаксических анализаторах. Например, декораторами могут быть узлы, которые хранят в себе информацию о типе. У вас есть синтаксическое дерево, состоящее просто из узлов, а потом проход занимающийся семантическим анализом надевает на это дерево декораторы и получается размеченное типами дерево. Тогда встречая следующие проходы, которые не хотят ничего знать про типы, они могут спокойно пользоваться деревом как и раньше, потому что декоратор реализует интерфейс декорируемого объекта, все хорошо, они даже ничего не подозревают об этом, а те проходы, которым информация о типе нужна, могут откастать каждый узел к конкретному типу декоратора и спрашивать у него дополнительное состояние. Такой подход часто используется для разметки графов. Есть граф без разметки, его можно с помощью декораторов дополнить дополнительной информацией.

## 28. Паттерн «Стратегия».

Паттерн “Стратегия”

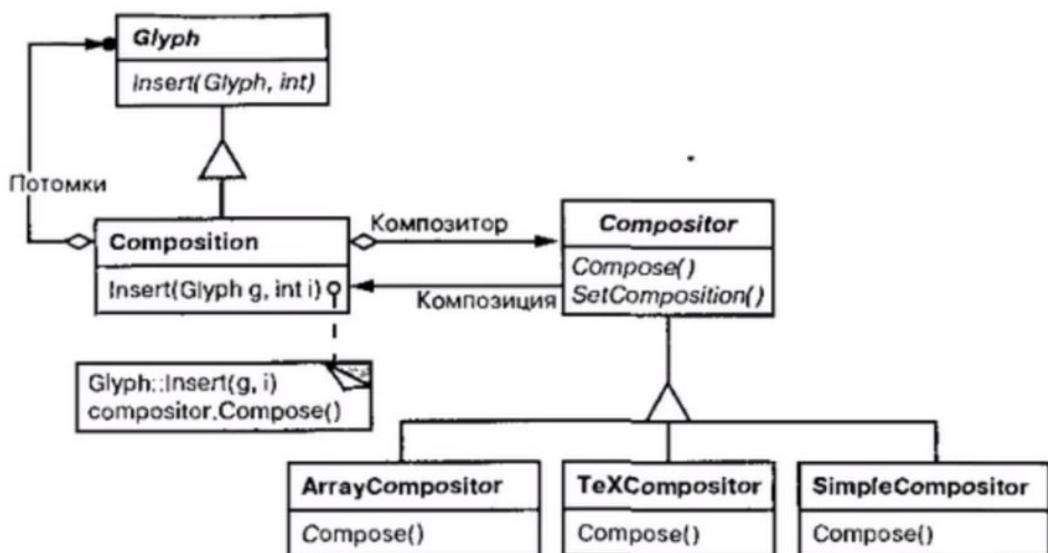
### Форматирование текста

- ▶ Задача — разбиение текста на строки, колонки и т.д.
- ▶ Высокоуровневые параметры форматирования
  - ▶ Ширина полей, размер отступа, межстрочный интервал и т.д.
- ▶ Компромисс между качеством и скоростью работы
- ▶ Инкапсуляция алгоритма

Очень популярный паттерн. В нашем случае мы хотим его использовать для решения задач, связанных с аналитических операций, форматированием текста. Форматирование текста - разбиение его на строки, колонки и так далее. Пользователь задает только

высокоуровневые параметры(ширина отступа, ширина полей), а текстовый редактор пытается так разместить слова в документе, чтобы они соответствовали этим параметрам. Проблема: разные алгоритмы форматирования могут использовать разные сведения о документе и давать совершенно разные результаты. Пример: в тхе используется такой странный критерий, как цвет документа это соотношение пробельных и непробельных символов на странице. Тех умеет избегать ситуаций с визуальным разрывом документа, когда пробелы случайным образом выстраиваются в колонну, и визуально текст делится на колонки. Из-за того что тхе необходимо учесть все такие тонкости, компиляция документа занимает какое-то время. В ворде наоборот, он делает это в реальном времени, но немного хуже. В нашем текстовом редакторе хочется уметь выбирать между быстрым и некачественным и медленным, но качественным режимами форматирования. Это нужно для того, чтобы когда мы редактируем документ, мы можем это делать в реальном времени, а когда уже все сделали и готовы выпустить окончательную версию, мы переключаемся в “качественный” режим и компилируем красивый ответ. И для этого мы хотим спрятать алгоритм форматирования от самого редактора, чтобы он об этом ничего не знал. Для этого мы создадим классы Compositor и Composition.

## Compositor и Composition

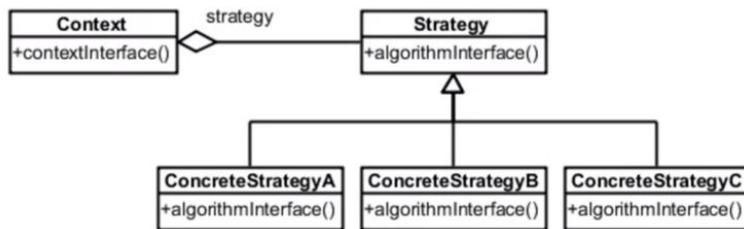


**Composition** это глиф, который может сам располагать свои элементы в документе. Для того чтобы это делать он использует интерфейс **Compositor**, который определяет методы `Compose` и `SetComposition`. `Compose` берет **Composition**, который заранее был выставлен и вставляет в него дополнительные пробельные символы, для того, чтобы обеспечить нужное форматирование. И вот этот **Compositor** реализуется уже конкретными классами, которые обеспечивают уже разные алгоритмы. На слайде это простой алгоритм, теховский алгоритм и механизм на массивах. Обобщается это все до паттерна стратегия.

# Паттерн “Стратегия”

## Strategy

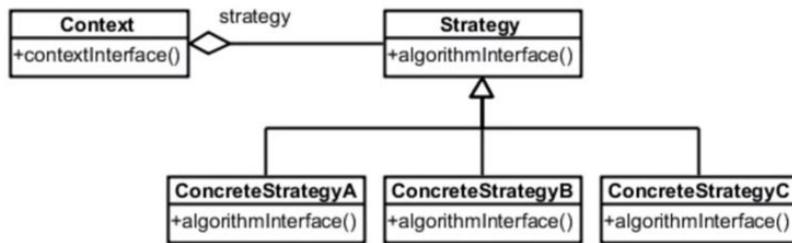
- ▶ Назначение — инкапсуляция алгоритма в объект
- ▶ Самое важное — спроектировать интерфейсы стратегии и контекста
  - ▶ Так, чтобы не менять их для каждой стратегии
- ▶ Применяется, если
  - ▶ Имеется много родственных классов с разным поведением
  - ▶ Нужно иметь несколько вариантов алгоритма
  - ▶ В алгоритме есть данные, про которые клиенту знать не надо
  - ▶ В коде много условных операторов



Стратегия устроена примерно таким же образом. Это Context в нашем примере это класс Composition. Интерфейс (или класс) Strategy инкапсулирует алгоритм. Стратегия реализуется разными конкретными стратегиями. Задача паттерна это инкапсуляция алгоритма в объект так, чтобы с алгоритмом можно было работать как с конкретными данными. Прежде всего для того чтобы уметь использовать полиморфизм и уметь легко переключаться между разными алгоритмами. Применяется чаще всего когда у нас есть родственные классы с несколько различающимся поведением, это поведение можно инкапсулировать в объект стратегию и сделать один класс, который параметризуется стратегии(либо принимает его в конструкторе, либо как значение какого-нибудь свойства). Иногда просто в предметной области у нас есть несколько вариантов одного алгоритма, алгоритм достаточно сложный и нам хочется иметь возможность выбирать из вариантов. Поэтому мы алгоритм прячем в объекте, и разные варианты реализуем, как разные варианты этого алгоритма. Иногда это применяется и когда один алгоритм, если он достаточно сложный его стоит инкапсулировать. Иногда это применяется когда алгоритм использует в своей работе временные данные (разного рода кэши), они могут быть достаточно сложны и из-за этого стоит вынести алгоритм в отдельный класс, чтобы все эти данные стали его private частями, а контекст про них ничего не знал. При проектировании стоит учитывать, чтобы любая реализация могла пользоваться этим интерфейсом. Стратегии очень часто нужны данные из контекста. Передавать данные можно двумя разными способами. Можно отдать стратегии весь контекст целиком и тогда каждая конкретная реализация сможет достать из контекста то, что ей нужно. Тогда контексту необходимо иметь паблик методы, которые дают доступ к тому что нужно стратегиям, это плохо так как контекст может использовать кто-то еще, если этот кто-то еще сможет нарушить инкапсуляцию контекста просто из-за того что стратегии могут им пользоваться то это плохая идея. Решение: стратегии можно реализовать, как вложенные классы внутри контекста. Второй способ: Интерфейс стратегии отдавать те данные из контекста, которые

им нужны. Минус - может не удастся предсказать какие данные нужны каждой конкретной стратегии, появится новая стратегия, ей не хватит данных, придется все переписывать. Зато не нарушена инкапсуляция контекста.

## “Стратегия” (Strategy), детали реализации



- ▶ 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

Как раз тот самый вопрос про передачу данных из контекста в стратегию. Передача самого контекста образует неизбежную циклическую зависимость между стратегией и контектом. То есть стратегии и контексту придется существовать вместе. Если передавать как параметры, то стратегия может вполне существовать отдельно от контекста, но здесь проблема с выбором передаваемых параметров либо мы передаем лишние, либо про какие-то можем забыть.

## “Стратегия” (Strategy), детали реализации (2)

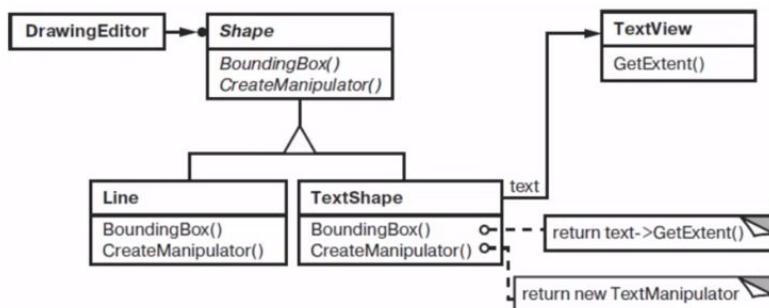
- ▶ Стратегия может быть параметром шаблона
  - ▶ Если не надо её менять на лету
  - ▶ Не надо абстрактного класса и нет оверхеда на вызов виртуальных методов
- ▶ Стратегия по умолчанию
  - ▶ Или просто поведение по умолчанию, если стратегия не установлена
- ▶ Объект-стратегия может быть приспособленцем

Стратегия может быть параметром шаблона, особенно в C++, где шаблоны и системы очень развиты. В таком случае ее не получится уже поменять во время выполнения (шаблоны генерируются во время компиляции). Зато нет абстрактного класса, нет оверхеда на вызов виртуальных методов. Именно так устроена вся стандартная библиотека C++. Класс строк это массив чаров снабженный стратегией выделения памяти, снабженный трейдами и так далее. Это просто обычные стратегии, которые передаются в класс как параметры шаблона. Если стратегия выставляется извне, то ее могут забыть выставить, для этого выставляют стратегию по умолчанию. Стратегии чаще всего не имеют собственного состояния и поэтому они могут быть разделяемыми. Иногда они могут иметь состояние и тут может применяться паттерн приспособленец, когда все состояние необходимое для работы хранится в контексте, извлекается из контекста при вызове метода из стратегии и помещается обратно в контекст по окончании времени работы. Это позволяет объект стратегии переиспользовать в нескольких разных контекстах, что хорошо, если у вас контекстов миллион а стратегий по сути две. В современном мире паттерн стратегия не очень полезен, так как есть лямбда функции. У лямбда функции есть одна маленькая проблема - может быть только один метод в реализации интерфейса. Если стратегия предполагает несколько разных методов, то лямбда функции не подойдут.

## 29. Паттерн «Адаптер».

### Проблема неподходящих интерфейсов

- ▶ Графический редактор
  - ▶ Shape, Line, Polygon, ...
- ▶ Сторонний класс TextView
  - ▶ Хотим его реализацию
  - ▶ Другой интерфейс



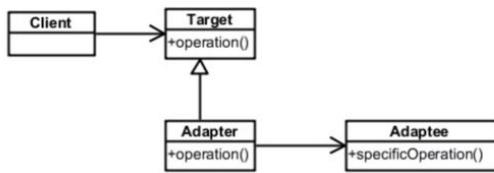
В примере с текстовым редактором у нас допустим есть какая-нибудь сторонняя библиотека, которая умеет здорово рисовать всякие графические элементы и есть наш код, который умеет работать с текстом. Или наоборот мы пишем графический редактор, мы не хотим делать под нее рисование текста, так как нам пришлось бы разбираться со многими типографскими вещами. Мы просто берем стороннюю библиотеку и пытаемся встроить ее в наш графический редактор чтобы она нам текстовые фигуры рисовала по заказу. Однако

есть следующая проблема, у нас есть Shape у него есть свой интерфейс, который мы сами спроектировали, есть сторонняя библиотека TextView, которая ничего не знает про наш shape и это проблема. Поэтому мы хотим применить паттерн адаптер. Для этого мы создаем свою реализацию shape - TextViewShape, который сам ничего делать не умеет, но хранит в себе поле TextView и все запросы по интерфейсу shape передает TextView.

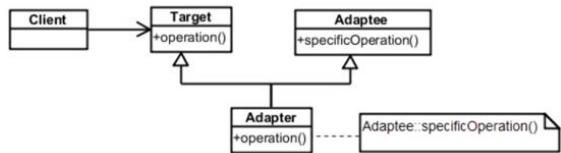
## Паттерн “Адаптер”

### Adapter

#### ► Адаптер объекта:



#### ► Адаптер класса:



#### ► Нужно множественное наследование

##### ► private-наследование в C++

Обобщается это в два разных варианта паттерна адаптер. На предыдущем слайде был адаптер объекта. Когда у нас есть целевой интерфейс, который хотим адаптировать, у нас есть адаптируемый объект, у которого интерфейс совершенно другой. Адаптер наследуется от целевого интерфейса имея у себя в качестве поля адаптируемый объект, ну и operation реализуется как вызов specificOperation. Клиент пользуется нашим адаптируемым объектом по интерфейсу таргет. С адаптером класса немного иначе. У него есть тот же самый интерфейс таргет и адаптер наследуется и от таргета и от адаптируемого объекта. Это позволяет не писать в адаптере лишнего поля и позволяет при вызове операции адапти не писать конкретно поле точка имя операции. В любом нормальном языке можно написать specificOperation и будет вызвана правильная операция у предка. Если таргет это не интерфейс, а абстрактный класс то требуется наследование, большинство современных языков его не поддерживают. К счастью в таком случае таргет просто делают интерфейсом. Еще проблема в том что адаптер не является потомком адапти по сути, так как он использует его только для реализации своих операций, в C++ это выражается с помощью приват наследования. По скольку правила наследования никем кроме C++ не поддерживаются то это очень специфичный для плюсов паттерн. Адаптер класса используется очень редко, может быть полезен для двунаправленных адаптеров, которые пытаются адаптировать и то и другое (литвинов на практике таких не встречал).

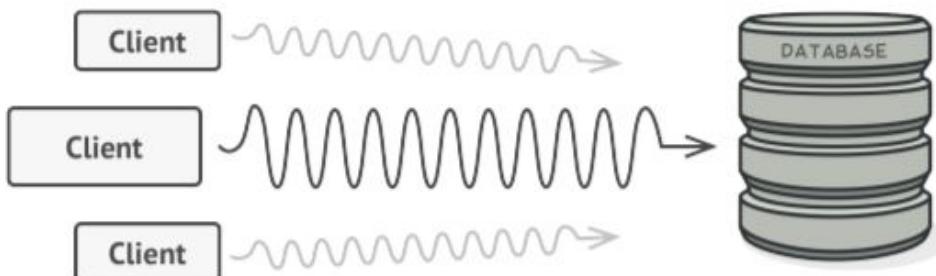
## 30. Паттерн «Заместитель».

Заместитель (Proxy) — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты

перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

### Проблема:

Для чего вообще контролировать доступ к объектам? Рассмотрим такой пример: у вас есть внешний ресурсоёмкий объект, который нужен не все время, а изредка.

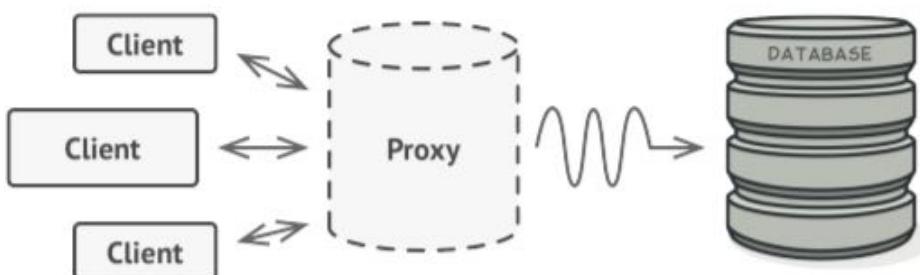


*Запросы к базе данных могут быть очень медленными.*

Мы могли бы создавать этот объект не в самом начале программы, а только тогда, когда он кому-то реально понадобится. Каждый клиент объекта получил бы некий код отложенной инициализации. Но, вероятно, это привело бы к множественному дублированию кода. В идеале, этот код хотелось бы поместить прямо в служебный класс, но это не всегда возможно. Например, код класса может находиться в закрытой сторонней библиотеке.

### Решение:

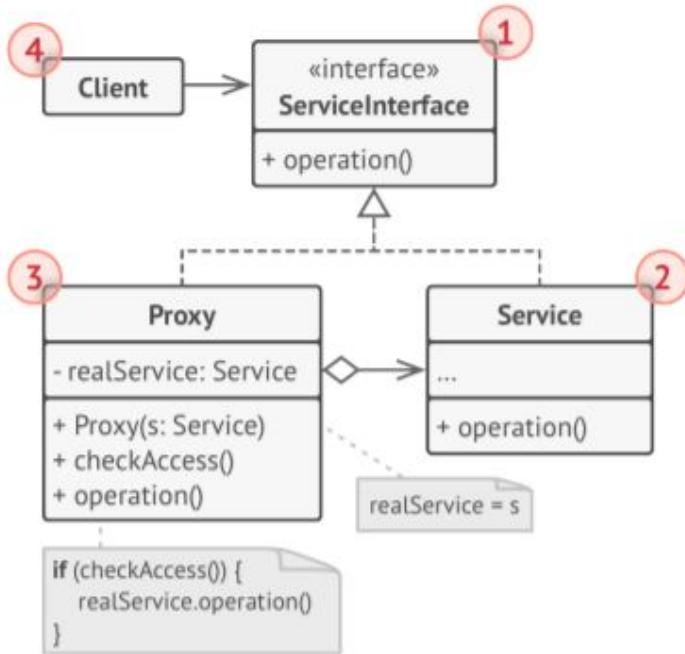
Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.



*Заместитель «притворяется» базой данных, ускоряя работу за счёт ленивой инициализации и кеширования повторяющихся запросов.*

Но в чём же здесь польза? Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одноковому интерфейсу, объект-заместитель можно передать в любой код, ожидающий сервисный объект.

### Структура:



- **Интерфейс сервиса** определяет общий интерфейс для сервиса и заместителя. Благодаря этому, объект заместителя можно использовать там, где ожидается объект сервиса.
- **Сервис** содержит полезную бизнес-логику
- **Заместитель** хранит ссылку на объект сервиса. После того как заместитель заканчивает свою работу (например, инициализацию, логирование, защиту или другое), он передаёт вызовы вложенному сервису. Заместитель может сам отвечать за создание и удаление объекта сервиса.
- **Клиент** работает с объектами через интерфейс сервиса. Благодаря этому, его можно «одурачить», подменив объект сервиса объектом заместителя.

#### Применимость:

**Ленивая инициализация (виртуальный прокси).** Когда у вас есть тяжёлый объект, грузящий данные из файловой системы или базы данных.

Вместо того, чтобы грузить данные сразу после старта программы, можно сэкономить ресурсы и создать объект тогда, когда он действительно понадобится

**Защита доступа (защищающий прокси).** Когда в программе есть разные типы пользователей, и вам хочется защищать объект от неавторизованного доступа. Например, если ваши объекты — это важная часть операционной системы, а пользователи — сторонние программы (хорошие или вредоносные).

Прокси может проверять доступ при каждом вызове и передавать выполнение служебному объекту, если доступ разрешён.

**Локальный запуск сервиса (удалённый прокси).** Когда настоящий сервисный объект находится на удалённом сервере.

В этом случае заместитель транслирует запросы клиента в вызовы по сети в протоколе, понятном удалённому сервису.

**Логирование запросов (логирующий прокси).** Когда требуется хранить историю обращений к сервисному объекту.

Заместитель может сохранять историю обращения клиента к сервисному объекту.

**Кеширование объектов («умная» ссылка).** Когда нужно кешировать результаты запросов клиентов и управлять их жизненным циклом.

Заместитель может подсчитывать количество ссылок на сервисный объект, которые были отданы клиенту и остаются активными. Когда все ссылки освобождаются, можно будет освободить и сам сервисный объект (например, закрыть подключение к базе данных). Кроме того, Заместитель может отслеживать, не менял ли клиент сервисный объект. Это позволит использовать объекты повторно и здороно экономить ресурсы, особенно если речь идёт о больших прожорливых сервисах.

#### **Преимущества:**

Позволяет контролировать сервисный объект незаметно для клиента.

Может работать, даже если сервисный объект ещё не создан.

Может контролировать жизненный цикл служебного объекта.

#### **Недостатки:**

Усложняет код программы из-за введения дополнительных классов.

Увеличивает время отклика от сервиса.

#### **Сравнение с другими паттернами:**

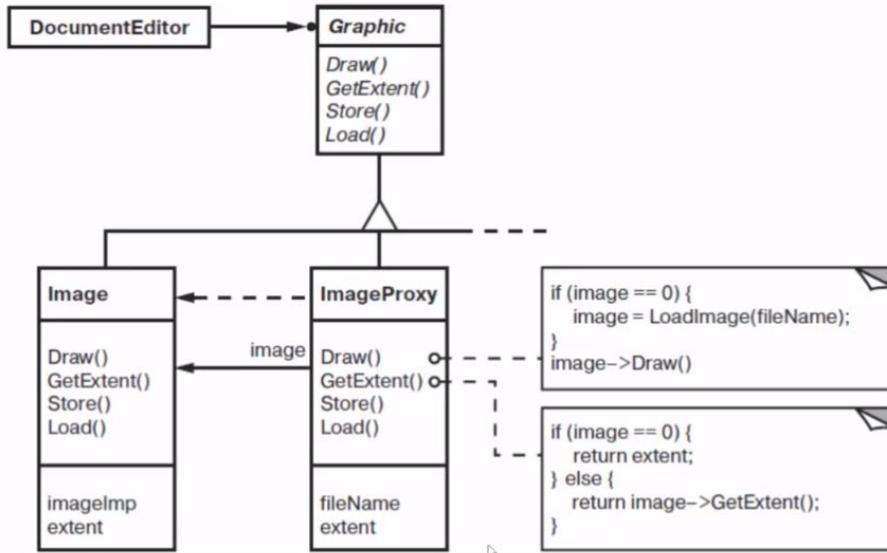
- **Адаптер** предоставляет классу альтернативный интерфейс. **Декоратор** предоставляет расширенный интерфейс. **Заместитель** предоставляет тот же интерфейс.
- **Фасад** похож на **Заместитель** тем, что замещает сложную подсистему и может сам её инициализировать. Но в отличие от Фасада, Заместитель имеет тот же интерфейс, что его служебный объект, благодаря чему их можно взаимозаменять.
- **Декоратор** и **Заместитель** имеют схожие структуры, но разные назначения. Они похожи тем, что оба построены на композиции и делегируют работу другим объектам. Паттерны отличаются тем, что Заместитель сам управляет жизнью сервисного объекта, а обёртывание Декораторов контролируется клиентом.

#### **Комментарии Литвинова, которые, как мне кажется, могут быть полезными:**

Пример из текстового редактора. Пусть у нас на какой-то странице документа есть большое изображение, и мы не хотим подгружать его сразу, а подгрузить его только тогда, когда пользователь долистает до него. Для этого мы создаем объект прокси, который ведет себя точно так же как обычное изображение, но не грузит изображение с диска, единственное что он делает читает файл, узнает размеры изображения, заголовок. Теперь когда документ знает размеры изображения он может себя спокойно отрендерить, сделав разбивку на страницы, а само изображение прокси честно подгрузит только когда пользователь до него

долистает. Очень напоминает паттерн декоратор. Как пример описанного выше прикрепляю слайд.

## Отложенная загрузка изображения



Заместитель отличается от декоратора класса назначением. Декоратор можно рассматривать как частный случай паттерна Заместитель.

Прежде всего этот паттерн в реальной жизни используется для проксирования объектов, которые работают на другой машине (WSF?). Стремление сделать так чтобы данные вызовы были очень похожи на локальные вызовы.

Прокси используется для механизмов отложенной загрузки (как это было выше).

Прокси для контроля доступа. Проверяет права на выполнение операции перед тем как реально выполнить операцию.

Умные указатели это прокси.

При проектировании следует следить за тем что у проксируемого объекта может не быть памяти.

## 31. Паттерн «Фасад».

**Фасад** — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

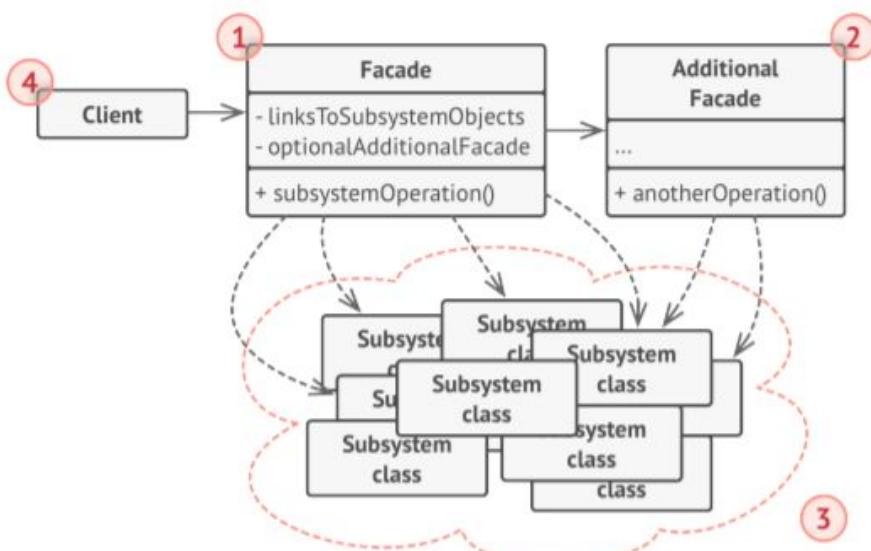
### Проблема:

Вашему коду приходится работать с большим количеством объектов некой сложной библиотеки или фреймворка. Вы должны самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее. В результате бизнес-логика ваших классов тесно переплетается с деталями реализации сторонних классов. Такой код довольно сложно понимать и поддерживать.

### Решение:

Фасад — это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100% функциональности, которой можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные. Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей. К примеру, программа, заливающая видео котиков в социальные сети, может использовать профессиональную библиотеку сжатия видео. Но все, что нужно клиентскому коду этой программы — простой метод `encode(filename, format)`. Создав класс с таким методом, вы реализуете свой первый фасад.

### Структура:



1. **Фасад** предоставляет быстрый доступ к определённой функциональности подсистемы. Он «знает», каким классам нужно переадресовать запрос, и какие данные для этого нужны.
2. **Дополнительный фасад** можно ввести, чтобы не «захламлять» единственный фасад разнородной функциональностью. Он может использоваться как клиентом, так и другими фасадами.
3. **Сложная подсистема** состоит из множества разнообразных классов. Для того, чтобы заставить их что-то делать, нужно знать подробности устройства подсистемы, порядок инициализации объектов и так далее. Классы подсистемы не знают о существовании фасада и работают друг с другом напрямую.
4. **Клиент** использует фасад вместо прямой работы с объектами сложной подсистемы.

### Применимость:

## **Когда вам нужно представить простой или урезанный интерфейс к сложной подсистеме.**

Часто подсистемы усложняются по мере развития программы. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать, настраивая её каждый раз под конкретные нужды, но вместе с тем, применять подсистему без настройки становится труднее. Фасад предлагает определённый вид системы по умолчанию, устраивающий большинство клиентов.

## **Когда вы хотите разложить подсистему на отдельные слои.**

Используйте фасады для определения точек входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Например, возьмём ту же сложную систему видеоконвертации. Вы хотите разбить её на слои работы с аудио и видео. Для каждой из этих частей можно попытаться создать фасад и заставить классы аудио и видео обработки общаться друг с другом через эти фасады, а не напрямую.

### **Преимущества:**

Изолирует клиентов от компонентов сложной подсистемы.

### **Недостатки:**

Фасад рискует стать **божественным объектом**, привязанным ко всем классам программы.

### **Отношения с другими паттернами:**

- **Фасад** задаёт новый интерфейс, тогда как **Адаптер** повторно использует старый. Адаптер оборачивает только один класс, а Фасад оборачивает целую подсистему. Кроме того, Адаптер позволяет двум существующим интерфейсам работать сообща, вместо того, чтобы задать полностью новый.
- **Абстрактная фабрика** может быть использована вместо **Фасада** для того, чтобы скрыть платформо-зависимые классы.
- **Легковес** показывает, как создавать много мелких объектов, а **Фасад** показывает, как создать один объект, который отображает целую подсистему.
- **Посредник** и **Фасад** похожи тем, что пытаются организовать работу множества существующих классов.
  - Фасад создаёт упрощённый интерфейс к подсистеме, не внося в неё никакой добавочной функциональности. Сама подсистема не знает о существовании Фасада. Классы подсистемы общаются друг с другом напрямую.
  - Посредник централизует общение между компонентами системы. Компоненты системы знают только о существовании Посредника, у них нет прямого доступа к другим компонентам.
- **Фасад** можно сделать **Одиночкой**, так как обычно нужен только один объект-фасад.
- **Фасад** похож на **Заместитель** тем, что замещает сложную подсистему и может сам её инициализировать. Но в отличие от Фасада, Заместитель имеет тот же интерфейс, что его служебный объект, благодаря чему их можно взаимозаменять.

## **Комментарии Литвинова, которые, как мне кажется, могут быть полезными:**

Паттерн фасад применяется когда у нас есть сложная система и мы хотим дать возможность ей легко пользоваться. Фасад хорошо применим, если у нас есть разные подсистемы, которые мы хотим инкапсулировать. Фасад подходит для разделения между уровнями в многоуровневых архитектурах, это позволяет хорошо прятать сложность. Фасад можно понимать как интерфейс для систем из нескольких классов. На практике фасады используются, если вам нужно инкапсулировать систему целиком.

## **32. Паттерн «Приспособленец».**

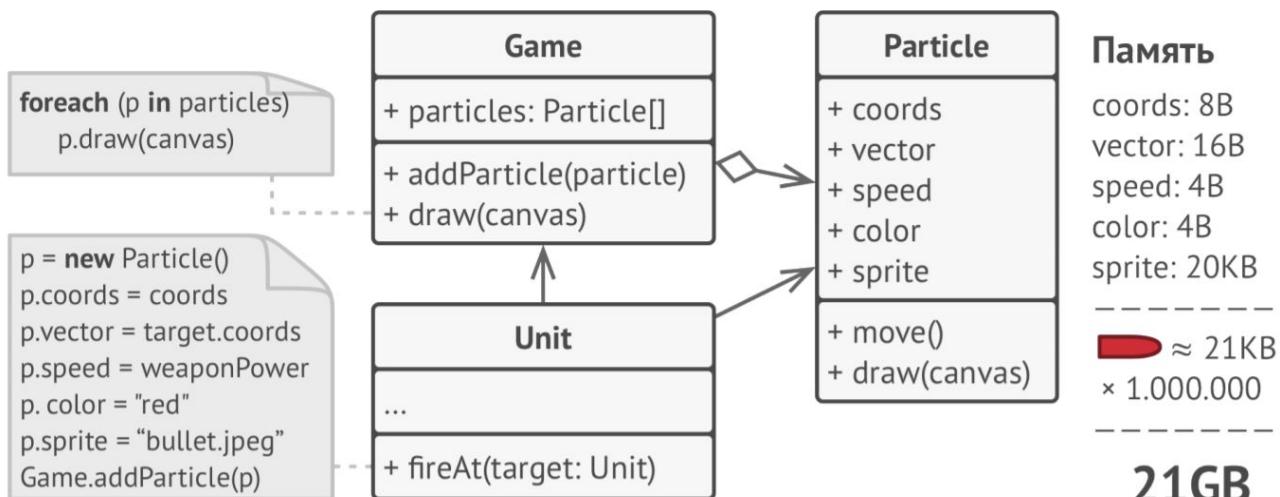
Легковес — это структурный паттерн проектирования, который позволяет вместить большее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

### **Проблема:**

На досуге вы решили написать небольшую игру, в которой игроки перемещаются по карте и стреляют друг в друга. Фишкой игры должна была стать реалистичная система частиц. Пули, снаряды, осколки от взрывов — всё это должно красиво летать и радовать взгляд.

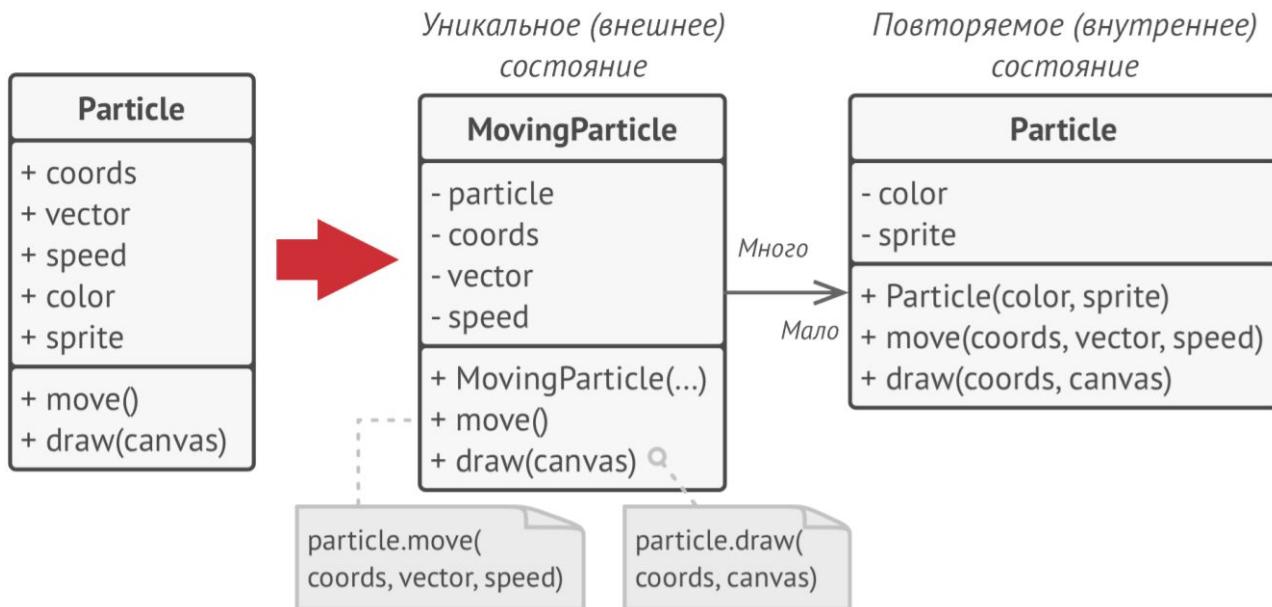
Игра отлично работала на вашем мощном компьютере. Однако ваш друг сообщил, что игра начинает тормозить и вылетает через несколько минут после запуска. Покопавшись в логах, вы обнаружили, что игра вылетает из-за недостатка оперативной памяти. У вашего друга компьютер значительно менее «прокачанный», поэтому проблема у него и проявляется так быстро.

И действительно, каждая частица представлена собственным объектом, имеющим множество данных. В определённый момент, когда побоище на экране достигает кульминации, новые объекты частиц уже не вмещаются в оперативную память компьютера, и программа вылетает.



### Решение:

Если внимательно посмотреть на класс частиц, то можно заметить, что цвет и спрайт занимают больше всего памяти. Более того, они хранятся в каждом объекте, хотя фактически их значения одинаковы для большинства частиц.

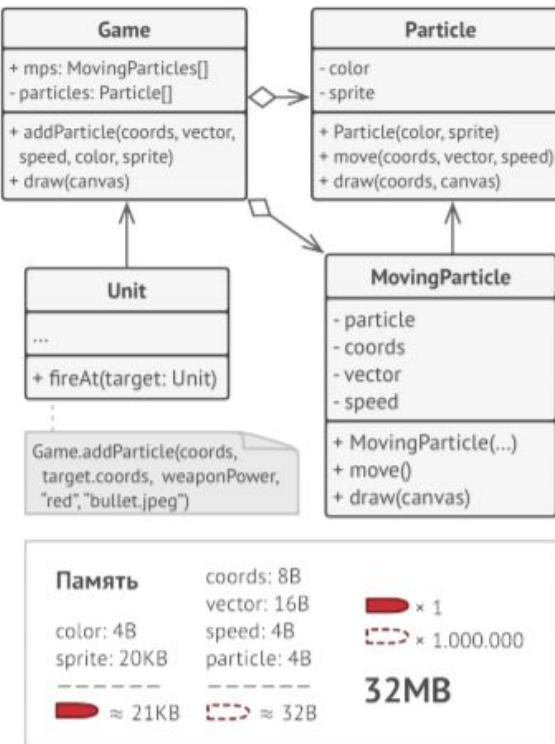


Остальное состояние объектов — координаты, вектор движения и скорость — отличаются для всех частиц. Таким образом, эти поля можно рассматривать как контекст, в котором частица используется. А цвет и спрайт — это данные, не изменяющиеся во времени.

Неизменяемые данные объекта принято называть «внутренним состоянием». Все остальные данные — это «внешнее состояние».

Паттерн Легковес предлагает не хранить в классе внешнее состояние, а передавать его в те или иные методы через параметры. Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах. Но главное — понадобится гораздо меньше

объектов, ведь теперь они будут отличаться только внутренним состоянием, а оно имеет не так много вариаций.



В нашем примере с частицами достаточно будет оставить всего три объекта с отличающимися спрайтами и цветом — для пуль, снарядов и осколков. Несложно догадаться, что такие облегчённые объекты называют легковесами.

«Но погодите-ка, нам потребуется столько же этих объектов, сколько было в самом начале!», — скажете вы и будете правы! Но дело в том, что объекты-контексты занимают намного меньше места, чем первоначальные. Ведь самые тяжёлые поля остались в легковесах (простите за каламбур), и сейчас мы будем ссылаться на эти объекты из контекстов, вместо того, чтобы повторно хранить дублирующееся состояние.

## Неизменяемость Легковесов

Так как объекты легковесов будут использованы в разных контекстах, вы должны быть уверены в том, что их состояние невозможно изменить после создания. Всё внутреннее состояние легковес должен получать через параметры конструктора. Он не должен иметь сеттеров и публичных полей.

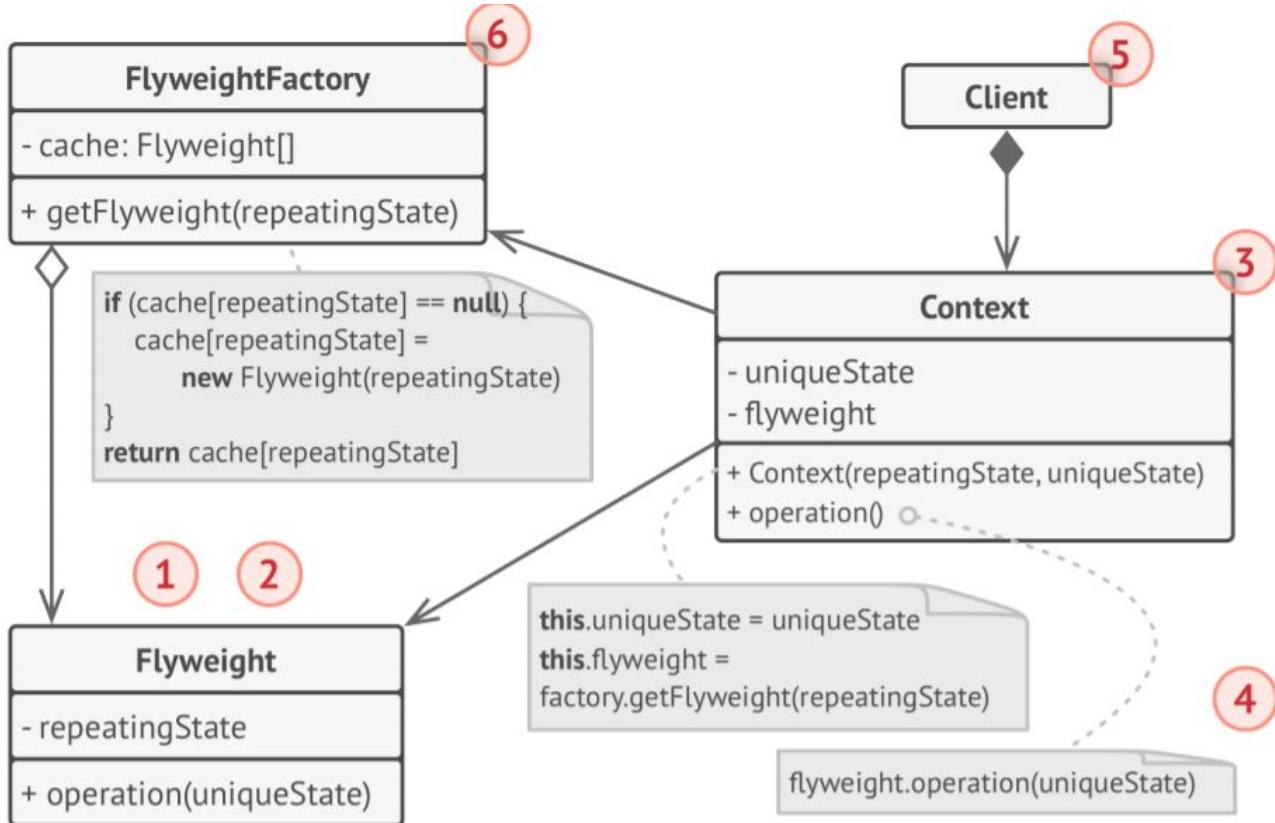
## Фабрика Легковесов

Для удобства работы с легковесами и контекстами можно создать фабричный метод, принимающий в параметрах всё внутреннее (а иногда и внешнее) состояние желаемого объекта.

Главная польза от этого метода в том, чтобы искать уже созданные легковесы с таким же внутренним состоянием, что и требуемое. Если легковес находится, его можно повторно

использовать. Если нет — просто создаём новый. Обычно этот метод добавляют в контейнер легковесов либо создают отдельный класс-фабрику. Его даже можно сделать статическим и поместить в класс легковесов.

### Структура:



1. Вы всегда должны помнить о том, что Легковес применяется в программе, имеющей громадное количество одинаковых объектов. Этих объектов должно быть так много, чтобы они не помещались в доступную оперативную память без ухищрений. Паттерн разделяет данные этих объектов на две части — легковесы и контексты.

2. **Легковес** содержит состояние, которое повторялось во множестве первоначальных объектов. Один и тот же легковес можно использовать в связке со множеством контекстов. Состояние, которое хранится здесь, называется внутренним, а то, которое он получает извне — внешним.

3. **Контекст** содержит «внешнюю» часть состояния, уникальную для каждого объекта. Контекст связан с одним из объектов легковесов, хранящих оставшееся состояние.

4. Поведение оригинального объекта чаще всего оставляют в Легковесе, передавая значения контекста через параметры методов. Тем не менее, поведение можно поместить и в контекст, используя легковес как объект данных.

5. **Клиент** вычисляет или хранит контекст, то есть внешнее состояние легковесов. Для клиента легковесы выглядят как шаблонные объекты, которые можно настроить во время использования, передав контекст через параметры.

6. **Фабрика легковесов** управляет созданием и повторным использованием легковесов. Фабрика получает запросы, в которых указано желаемое состояние легковеса. Если легковес с таким состоянием уже создан, фабрика сразу его возвращает, а если нет — создаёт новый объект.

#### **Применимость:**

**Когда не хватает оперативной памяти для поддержки всех нужных объектов.**

Эффективность паттерна **Легковес** во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены все перечисленные условия:

- в приложении используется большое число объектов;
- из-за этого высоки расходы оперативной памяти;
- большую часть состояния объектов можно вынести за пределы их классов;
- большие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено.

#### **Преимущества:**

Экономит оперативную память.

#### **Недостатки:**

Расходует процессорное время на поиск/вычисление контекста.

Усложняет код программы из-за введения множества дополнительных классов.

#### **Отношения с другими паттернами:**

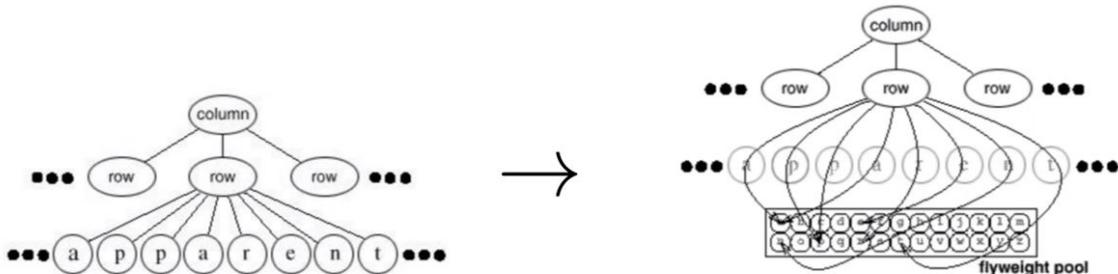
- **Компоновщик** часто совмещают с **Легковесом**, чтобы реализовать общие ветви дерева и сэкономить при этом память.
- **Легковес** показывает, как создавать много мелких объектов, а **Фасад** показывает, как создать один объект, который отображает целую подсистему.
- Паттерн **Легковес** может напоминать **Одиночку**, если для конкретной задачи у вас получилось свести количество объектов к одному. Но помните, что между паттернами есть два кардинальных отличия:
  1. В отличие от Одиночки, вы можете иметь множество объектов-легковесов.
  2. Объекты-легковесы должны быть неизменяемыми, тогда как объект-одиночка допускает изменение своего состояния.

#### **Комментарии Литвинова, которые, как мне кажется, могут быть полезными:**

Паттерн, который предназначается для поддержки большого количества мелких объектов. Например, в текстовом редакторе с каждым символом хочется работать, как с отдельным объектом, но у нас может быть миллион символов, и если это все отдельные объекты, то

это не очень хорошо с точки зрения используемой памяти. Для этого используется палитра символов как на слайде справа.

## “Приспособленец”, пример



И у нас есть ссылки на эти объекты из палитры, которые выглядят, как обычные символы, но они могут переиспользоваться.

Этот паттерн не очень популярный так как он очень ситуационный и требуется чтобы были выполнены все условия, написанные на слайде.

- ▶ Когда в приложении используется много мелких объектов
- ▶ Они допускают разделение состояния на внутреннее и внешнее
  - ▶ Желательно, чтобы внешнее состояние было вычислимым
- ▶ Идентичность объектов не важна
  - ▶ Используется семантика Value Type
- ▶ Главное, когда от такого разделения можно получить ощутимый выигрыш

## 33. Паттерн «Мост».

[преза](#) [конспект](#)

**Мост** — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга. Паттерн Мост предполагает, что основной код, необходимый для функционирования объекта, переносится в реализацию. Всё остальное, включая взаимодействие с клиентом, содержится в абстракции. Её методы, при необходимости, могут быть изменены или дополнены. Кроме того, она содержит экземпляр реализации и использует его для обработки поступающих от клиентов запросов. Под обработкой подразумевается как прямая переадресация запроса, так и вызов группы методов реализации для получения результата.

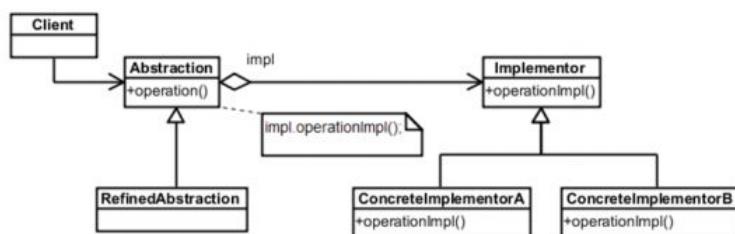
Пример:

Пример:

- ▶ Есть система, интерпретирующая программы для роботов
- ▶ Есть класс *Sensor*, от которого наследуются *SonarSensor*, *LightSensor*, ...
- ▶ Связь с роботом может выполняться по USB или Bluetooth, а может быть, программа и вовсе исполняется на симуляторе
- ▶ Интерпретатор хочет работать с сенсорами, не заморачиваясь реализацией механизма связи
- ▶ Рабоче-крестьянская реализация — *USBLightSensor*, *BluetoothLightSensor*, *USBSonarSensor*, *BluetoothSonarSensor*, ...
- ▶ Число классов — произведение количества сенсоров и типов связи

Схема:

### “Мост”, общая схема



- ▶ *Abstraction* — определяет интерфейс абстракции, хранит ссылку на реализацию
- ▶ *RefinedAbstraction* — расширяет интерфейс абстракции, делает полезную работу, используя реализацию
- ▶ *Implementor* — определяет интерфейс реализации, в котором абстракции предоставляются низкоуровневые операции
- ▶ *ConcreteImplementor* — предоставляет конкретную реализацию *Implementor*



Применимость:

- Когда хочется разделить абстракцию и реализацию, например, когда реализацию можно выбирать во время компиляции или во время выполнения (“Стратегия”, “Прокси”)
- Когда абстракция и реализация должны расширяться новыми подклассами
- Когда хочется разделить одну реализацию между несколькими объектами (как copy-on-write в строках)

## Реализация:

- Создание правильного Implementor-а: самой абстракцией в конструкторе, в зависимости от переданных параметров (как вариант — выбор реализации по умолчанию и замена её по ходу работы или если класс коллекции поддерживает несколько реализаций, то решение можно принять в зависимости от размера коллекции - для небольших коллекций применяется реализация связный список, для больших - в виде хешированных таблиц)
- Принимать реализацию извне (как параметр конструктора, или реже, как значение в сеттер)
- Фабрика/фабричный метод позволяет спрятать платформозависимые реализации, чтобы не зависеть от них всех при сборке. Преимущество - класс Abstraction напрямую не привязан ни к одному из классов Implementor.

## Отличия от других паттернов:

- Мост проектируют загодя, чтобы развивать большие части приложения отдельно друг от друга. **Адаптер** применяется постфактум, чтобы заставить несовместимые классы работать вместе.
- Мост, **Стратегия** и **Состояние** (а также слегка и **Адаптер**) имеют схожие структуры классов — все они построены на принципе «композиции», то есть делегирования работы другим объектам. Тем не менее, они отличаются тем, что решают разные проблемы. Помните, что паттерны — это не только рецепт построения кода определённым образом, но и описание проблем, которые привели к данному решению.
- **Абстрактная фабрика** может работать совместно с Мостом. Это особенно полезно, если у вас есть абстракции, которые могут работать только с некоторыми из реализаций. В этом случае фабрика будет определять типы создаваемых абстракций и реализаций.
- Паттерн **Строитель** может быть построен в виде Моста: директор будет играть роль абстракции, а строители — реализаций.

## Вырожденный мост (Pointer To Implementation (PImpl)):

Вырожденный мост для C++, когда “абстракция” имеет ровно одну реализацию, часто полностью дублирующую её интерфейс

Зачем: чтобы клиенты класса не зависели при сборке от его реализации

- Позитивно сказывается на времени компиляции программ на C++
- Позволяет менять реализацию независимо, сохраняя бинарную совместимость

Как: предварительное объявление класса-реализации, полное

определение — в .cpp-файле вместе с методами абстракции

Часто используется в реализации библиотек (например, Qt)

## 34. Паттерн «Фабричный метод».

[преза](#) [конспект](#)

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Пример:

### Пример, текстовый редактор

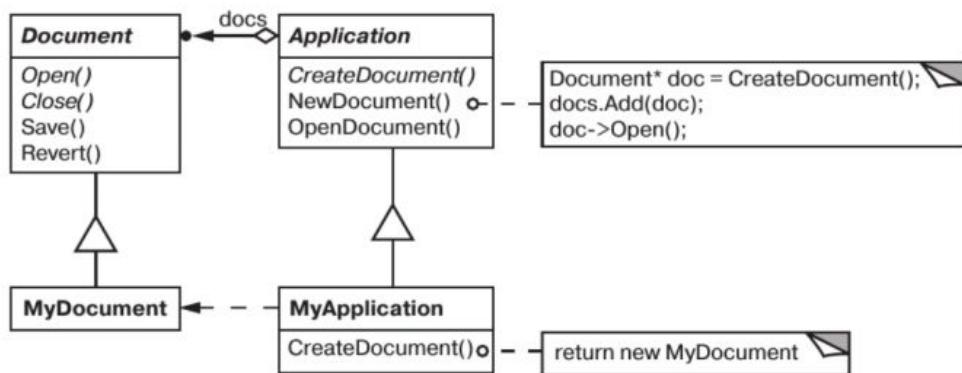
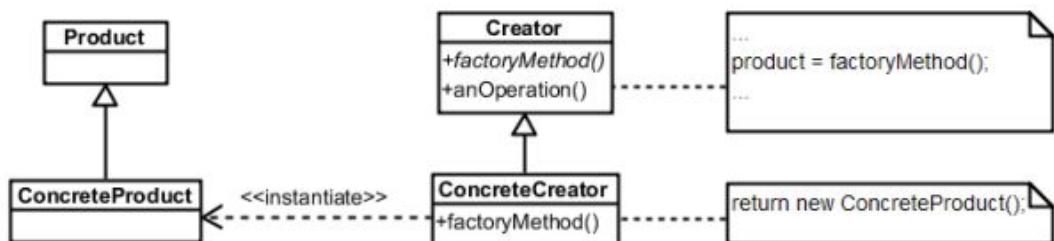


Схема:



- **Product** (продукт) - определяет интерфейс объектов, создаваемых фабричным методом
- **ConcreteProduct** (конкретный продукт) - реализует интерфейс **Product**
- **Creator** (создатель) - объявляет фабричный метод, возвращающий объект типа **Product**. **Creator** может также определять реализацию по умолчанию фабричного метода, который возвращает объект **ConcreteProduct**. Еще может вызывать фабричный метод для создания объекта **Product**.

Применимость:

- классу заранее неизвестно, объекты каких классов ему нужно создавать

- объекты, которые создает класс, специфицируются подклассами
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов

### Реализация:

- Две основных разновидности паттерна: абстрактный Creator (не содержит реализации объявленного в нем фабричного метода) или реализация по умолчанию (второй вариант может быть полезен для расширяемости). В первом случае для определения реализации необходимы подклассы, поскольку никакого разумного умолчания не существует; во втором случае конкретный класс Creator использует фабричный метод главным образом ради повышения гибкости.
- Параметризованные фабричные методы - еще один вариант паттерна, который позволяет фабричному методу создавать разные виды продуктов.
- Если язык поддерживает инстанциацию по прототипу (JavaScript, Smalltalk), можно хранить порождаемый объект
- Creator не может вызывать фабричный метод в конструкторе
- Можно сделать шаблонный Creator, чтобы не порождать подклассы

### Отличия от других паттернов:

- Многие архитектуры начинаются с применения Фабричного метода (более простого и расширяемого через подклассы) и эволюционируют в сторону **Абстрактной фабрики, Прототипа или Строителя** (более гибких, но и более сложных).
- Классы **Абстрактной фабрики** чаще всего реализуются с помощью Фабричного метода, хотя они могут быть построены и на основе **Прототипа**.
- Фабричный метод можно использовать вместе с **Итератором**, чтобы подклассы коллекций могли создавать подходящие им итераторы.
- **Прототип** не опирается на наследование, но ему нужна сложная операция инициализации. Фабричный метод, наоборот, построен на наследовании, но не требует сложной инициализации.
- Фабричный метод можно рассматривать как частный случай **Шаблонного метода**. Кроме того, Фабричный метод нередко бывает частью большого класса с Шаблонными методами.

## 35. Паттерн «Абстрактная фабрика».

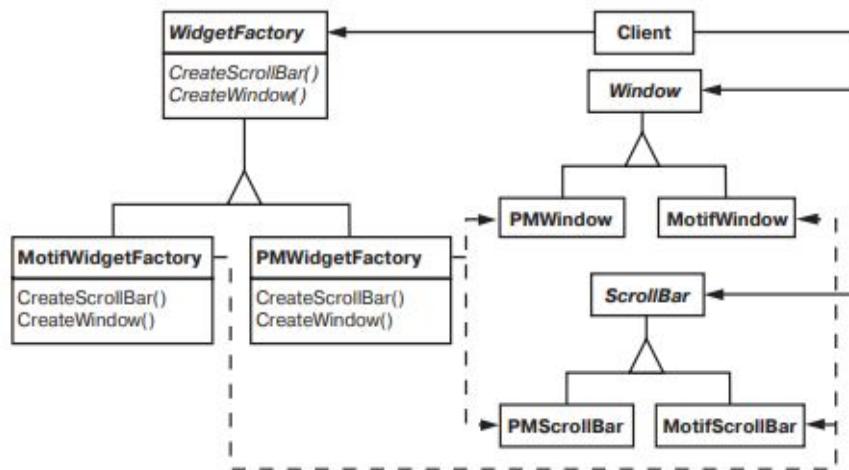
[преза](#) [конспект](#)

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

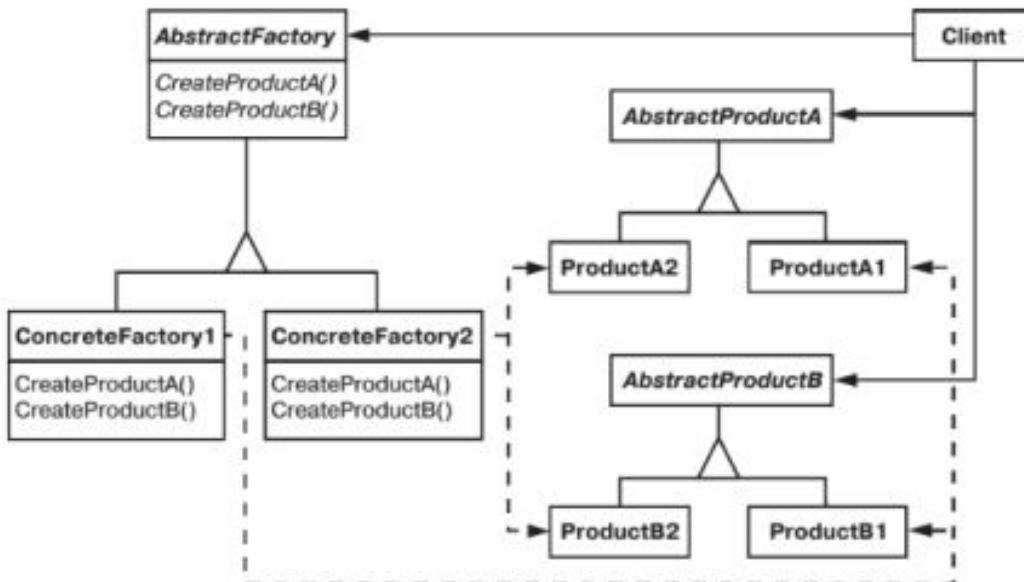
Пример:

Фабрика виджетов (значение каждого виджета в структуре ниже)

### Мотивация



Структура:



- **AbstractFactory** (**WidgetFactory**) - абстрактная фабрика. Объявляет интерфейс для операций, создающих абстрактные объекты-продукты.
- **ConcreteFactory** (**MotifWidgetFactory**, **PMWidgetFactory**) - конкретная фабрика. Реализует операции, создающие конкретные объекты-продукты.
- **AbstractProduct** (**Window**, **ScrollBar**) - абстрактный продукт. Объявляет интерфейс для типа объекта-продукта.

- **ConcreteProduct** (MotifWindow, MotifScrollBar) - конкретный продукт. Определяет объект-продукт, создаваемый соответствующей конкретной фабрикой. Реализует интерфейс Abstract-Product.
- **Client** - клиент. Пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct.

### Применимость:

- 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 и вам необходимо обеспечить выполнение этого ограничения
- Хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию

### Детали реализации:

- Хорошо комбинируются с паттерном “Одиночка”
  - Если семейств продуктов много, то фабрика может инициализироваться прототипами, тогда не надо создавать сотню подклассов
  - Прототип на самом деле может быть классом (например, Class в Java)
  - Если виды объектов часто меняются, может помочь параметризация метода создания. Может пострадать типобезопасность
- Известные применения:** В библиотеке InterViews для обозначения классов абстрактных фабрик используется суффикс Kit.

### Отличия от других паттернов:

- Многие архитектуры начинаются с применения **Фабричного метода** (более простого и расширяемого через подклассы) и эволюционируют в сторону Абстрактной фабрики, **Прототипа** или **Строителя** (более гибких, но и более сложных).
- **Строитель** концентрируется на построении сложных объектов шаг за шагом. Абстрактная фабрика специализируется на создании семейств связанных продуктов. Строитель возвращает продукт только после выполнения всех шагов, а Абстрактная фабрика возвращает продукт сразу же.
- Классы Абстрактной фабрики чаще всего реализуются с помощью **Фабричного метода**, хотя они могут быть построены и на основе **Прототипа**.
- Абстрактная фабрика может быть использована вместо **Фасада** для того, чтобы скрыть платформо-зависимые классы.
- Абстрактная фабрика может работать совместно с **Мостом**. Это особенно полезно, если у вас есть абстракции, которые могут работать только с некоторыми из

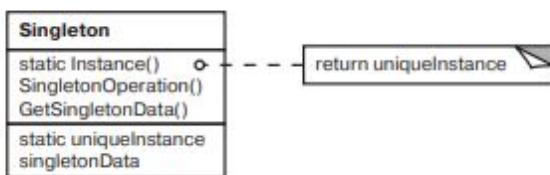
реализаций. В этом случае фабрика будет определять типы создаваемых абстракций и реализаций.

- Абстрактная фабрика, **Строитель** и **Прототип** могут быть реализованы при помощи **Одиночки**.

## 36. Паттерн «Одиночка».

[преза](#) [конспект](#)

Структура:



Singleton – одиночка:

- определяет операцию `Instance`, которая позволяет клиентам получать доступ к единственному экземпляру. `Instance` – это операция класса, то есть метод класса в терминологии Smalltalk и статическая функция- член в C++;
- может нести ответственность за создание собственного уникального экземпляра.

Мотивация:

Для некоторых классов важно, чтобы существовал только один экземпляр. Хотя в системе может быть много принтеров, но возможен лишь один спулер. Должны быть только одна файловая система и единственный оконный менеджер. В цифровом фильтре может находиться только один аналогоцифровой преобразователь (АЦП). Бухгалтерская система обслуживает только одну компанию. Как гарантировать, что у класса есть единственный экземпляр и что этот экземпляр легко доступен?

Глобальная переменная дает доступ к объекту, но не запрещает инстанцировать класс в нескольких экземплярах. Более удачное решение – сам класс контролирует то, что у него есть только один экземпляр, может запретить создание дополнительных экземпляров, перехватывая запросы на создание новых объектов, и он же способен предоставить доступ к своему экземпляру. Это и есть назначение паттерна одиночка.

Детали реализации:

- Гарантирует, что у класса есть только один экземпляр
- Предоставляет глобальный доступ к этому экземпляру
- Позволяет использовать подклассы без модификации клиентского кода

## Отличия от других паттернов:

- **Фасад** можно сделать Одиночкой, так как обычно нужен только один объект-фасад.
- Паттерн **Легковес** может напоминать Одиночку, если для конкретной задачи у вас получилось свести количество объектов к одному. Но помните, что между паттернами есть два кардинальных отличия:
  - В отличие от Одиночки, вы можете иметь **множество объектов-легковесов**. Объекты-легковесы должны быть неизменяемыми, тогда как объект-одиночка допускает изменение своего состояния.
  - **Абстрактная фабрика**, **Строитель** и **Прототип** могут быть реализованы при помощи Одиночки.

## Плюсы и минусы:

### Плюсы:

- Гарантирует наличие единственного экземпляра класса.
- Предоставляет к нему глобальную точку доступа.
- Реализует отложенную инициализацию объекта-одиночки.

### Минусы:

- Добавляет неочевидные зависимости по данным. По сути, хитрая глобальная переменная
- Усложняет тестирование
- Нарушает принцип единственности ответственности
- Сложно рефакторить, если потребуется несколько экземпляров

## 37. Паттерны «Ленивая инициализация» и «Пул объектов».

[преза](#) конспекта нет ни на сайте рефакторинга, ни в книжке Гаммы-Хелли...

### Ленивая инициализация:

**Отложенная инициализация или «ленивая» инициализация** — это способ доступа к объекту, скрывающий за собой механизм, позволяющий отложить создание этого объекта до момента первого обращения. Необходимость ленивой инициализации может возникнуть по разным причинам: начиная от желания снизить нагрузку при старте приложения и заканчивая оптимизацией редко используемого функционала. И действительно, не все функции приложения используются всегда и, тем более, сразу, потому создание объектов, реализующих их, вполне рационально отложить до лучших времён.

- Действие не выполняется до тех пор, пока не нужен его результат
- Используется повсеместно, для ускорения запуска и экономии на редких вычислениях (Just-In-Time-компиляция (код не используется, не компилируется), ленивые структуры данных (списки в Haskell (тут все построено на

- ленивости, например можем объявить список всех простых чисел и не будет переполнения памяти), seq в F#, ленивые вычисления (Haskell, Lazy<T> в .NET)
- Имеет те же проблемы с многопоточностью, что и одиночка. Если используем в многопоточном окружении, надо подумать над синхронизацией между потоками, чтобы не вызывать функцию, выполняющую вычисления несколько раз.
    - Во многих языках поддерживается по умолчанию, в C# например Lazy Initialization – Инициализация по требованию. Это самый простой способ – реализовать проверку поля на null и в случае необходимости заполнять его данными. Самый простой вариант, доступный с первых версий языка, — это создание неинициализированной переменной и проверка её на null перед возвращением. Если переменная равна null, создаём объект и присваиваем этой переменной, а потом его возвращаем. При повторном обращении объект уже будет создан и мы сразу его вернём.

## Пул объектов:

**Объектный пул (англ. object pool)** — порождающий паттерн (шаблон) проектирования, набор инициализированных и готовых к использованию объектов. Когда системе требуется объект, он не создается, а берется из пула. Когда объект больше не нужен, он не уничтожается, а возвращается в пул.

### Мотивация:

Зачем он нужен? Вкратце — для повышения производительности, когда инициализация нового объекта приводит к большим затратам. Но тут важно понимать, что встроенный в .NET сборщик мусора прекрасно справляется с уничтожением легких короткоживущих объектов, поэтому применимость пула ограничивается следующими критериями:

- дорогие для создания и/или уничтожения объекты (примеры: сокеты, потоки, неуправляемые ресурсы);
- очистка объектов для переиспользования дешевле создания нового (или ничего не стоит);
- объекты очень большого размера.

### Применение:

- Информация об открытых файлах в DOS.
- Информация о видимых объектах во многих компьютерных играх (хорошим примером является движок Doom). Эта информация актуальна только в течение одного кадра; после того, как кадр выведен, список опустошается.
- Компьютерная игра для хранения всех объектов на карте, вместо того, чтобы использовать обычные механизмы распределения памяти, может завести массив такого размера, которого заранее хватит на все объекты, и свободные ячейки держать в виде связного списка. Такая конструкция повышает скорость, уменьшает фрагментацию памяти и снижает нагрузку на сборщик мусора (если он есть).

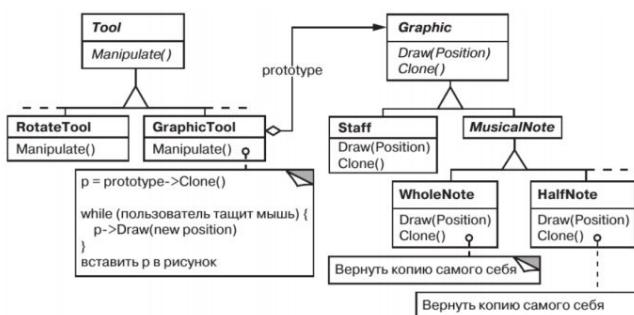
- ▶ Применяется, когда объекты создавать сложно, но каждый объект нужен лишь ненадолго
- ▶ Желательно, чтобы на поддержание объектов в памяти не требовалось много ресурсов, либо объектов в памяти было мало
  - ▶ Например, создать 50000 сетевых соединений “заранее” может быть плохой идеей
- ▶ Следует применять с осторожностью в языках со сборкой мусора — память держит ссылки на объекты
  - ▶ К тому же, в таких языках new отрабатывает мгновенно
- ▶ Следует помнить про многопоточность
  - ▶ Как правило, методы памяти требуют синхронизации

## 38. Паттерн «Прототип».

[презентация](#)

**Прототип** – паттерн, порождающий объекты. Задает виды создаваемых объектов с помощью экземпляра-прототипа и создает новые объекты путем копирования этого прототипа. Мотивация

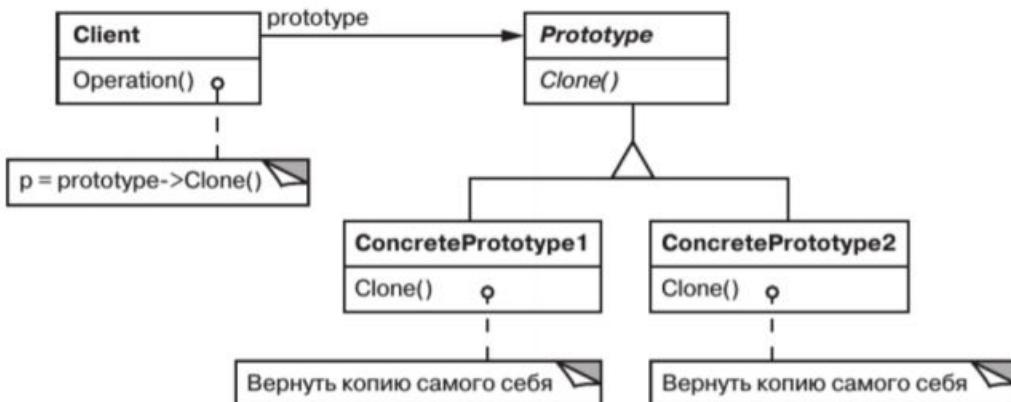
“Прототип”, мотивация



Построить музыкальный редактор удалось бы путем адаптации общего каркаса графических редакторов и добавления новых объектов, представляющих ноты, паузы и нотный стан. В каркасе редактора может присутствовать палитра инструментов для добавления в партитуру этих музыкальных объектов. Палитра может также содержать инструменты для выбора, перемещения и иных манипуляций с объектами. Так, пользователь, щелкнув, например, по значку четверти поместил бы ее тем самым в партитуру. Или, применив инструмент перемещения, сдвигал бы ноту на стане вверх или вниз, чтобы изменить ее высоту. Предположим, что каркас предоставляет абстрактный класс `Graphic` для графических компонентов вроде нот и нотных станов, а также абстрактный класс `Tool` для определения инструментов в палитре. Кроме того, в каркасе имеется предопределенный подкласс `GraphicTool` для инструментов, которые создают графические объекты и добавляют их в документ. Однако класс `GraphicTool` создает некую проблему для проектировщика каркаса. Классы нот и нотных станов специфичны для нашего приложения, а класс `GraphicTool` принадлежит каркасу. Этому классу ничего неизвестно о том, как создавать экземпляры наших музыкальных классов и добавлять их в партитуру. Можно было бы породить от `GraphicTool` подклассы для каждого вида музыкальных объектов, но тогда оказалось бы слишком много классов, отличающихся только тем, какой музыкальный

объект они инстанцируют. Мы знаем, что гибкой альтернативой порождению подклассов является композиция. Вопрос в том, как каркас мог бы воспользоваться ею для параметризации экземпляров GraphicTool классом того объекта Graphic, который предполагается создать. Решение – заставить GraphicTool создавать новый графический объект, копируя или «клонируя» экземпляр подкласса класса Graphic. Этот экземпляр мы будем называть прототипом. GraphicTool параметризуется прототипом, который он должен клонировать и добавить в документ. Если все подклассы Graphic поддерживают операцию Clone, то GraphicTool может клонировать любой вид графических объектов. Итак, в нашем музыкальном редакторе каждый инструмент для создания музыкального объекта – это экземпляр класса GraphicTool, инициализированный тем или иным прототипом. Любой экземпляр GraphicTool будет создавать музыкальный объект, клонируя его прототип и добавляя клон в партитуру. Можно воспользоваться паттерном прототип, чтобы еще больше сократить число классов. Для целых и половинных нот у нас есть отдельные классы, но, быть может, это излишне. Вместо этого они могли бы быть экземплярами одного и того же класса, инициализированного разными растровыми изображениями и длительностями звучания. Инструмент для создания целых нот становится просто объектом класса GraphicTool, в котором прототип MusicalNote инициализирован целой нотой. Это может значительно уменьшить число классов в системе. Заодно упрощается добавление нового вида нот в музыкальный редактор.

## Структура



- Prototype (Graphic) – прототип: объявляет интерфейс для клонирования самого себя;
- ConcretePrototype (Staff – нотный стан, WholeNote – целая нота, HalfNote – половинная нота) – конкретный прототип: реализует операцию клонирования себя;
- Client (GraphicTool) – клиент: создает новый объект, обращаясь к прототипу с запросом клонировать себя. Клиент обращается к прототипу, чтобы тот создал свою копию

## Детали реализации

- **Паттерн интересен только для языков, где мало runtime-информации о типе (C++)**. Прототип особенно полезен в статически типизированных языках вроде C++, где классы не являются объектами, а во время выполнения информации о типе недостаточно или нет вовсе. Меньший интерес данный паттерн представляет для таких языков, как Smalltalk или Objective C, в которых и так уже есть нечто

эквивалентное прототипу (именно – объект класс) для создания экземпляров каждого класса. В языки, основанные на прототипах, например Self, где создание любого объекта выполняется путем клонирования прототипа, этот паттерн просто встроен.

- **Реестр прототипов, обычно ассоциативное хранилище.** Если число прототипов в системе не фиксировано (то есть они могут создаваться и уничтожаться динамически), ведите реестр доступных прототипов. Клиенты должны не управлять прототипами самостоятельно, а сохранять и извлекать их из реестра. Клиент запрашивает прототип из реестра перед его клонированием. Такой реестр мы будем называть диспетчером прототипов. Диспетчер прототипов – это ассоциативное хранилище, которое возвращает прототип, соответствующий заданному ключу. В нем есть операции для регистрации прототипа с указанным ключом и отмены регистрации. Клиенты могут изменять и даже «просматривать» реестр во время выполнения, а значит, расширять систему и вести контроль над ее состоянием без написания кода;
- **Операция Clone.** Самая трудная часть паттерна прототип – правильная реализация операции Clone. Особенно сложно это в случае, когда в структуре объекта есть круговые ссылки. В большинстве языков имеется некоторая поддержка для клонирования объектов. Например, Smalltalk предоставляет реализацию копирования, которую все подклассы наследуют от класса Object. В C++ есть копирующий конструктор. Но эти средства не решают проблему «глубокого и поверхностного копирования». Суть ее в следующем: должны ли при клонировании объекта клонироваться также и его переменные экземпляра или клон просто разделяет с оригиналом эти переменные? Поверхностное копирование просто, и часто его бывает достаточно. Именно такую возможность и предоставляет по умолчанию Smalltalk. В C++ копирующий конструктор по умолчанию выполняет почлененное копирование, то есть указатели разделяются копией и оригиналом. Но для клонирования прототипов со сложной структурой обычно необходимо глубокое копирование, поскольку клон должен быть независим от оригинала. Поэтому нужно гарантировать, что компоненты клона являются клонами компонентов прототипа. При клонировании вам придется решать, что именно может разделяться и может ли вообще. Если объекты в системе предоставляют операции Save (сохранить) и Load (загрузить), то разрешается воспользоваться ими для реализации операции Clone по умолчанию, просто сохранив и сразу же загрузив объект. Операция Save сохраняет объект в буфере памяти, а Load создает дубликат, реконструируя объект из буфера;

## Отличия от других паттернов

- Многие архитектуры начинаются с применения **Фабричного метода** (более простого и расширяемого через подклассы) и эволюционируют в сторону **Абстрактной фабрики**, **Прототипа** или **Строителя** (более гибких, но и более сложных).
- Классы **Абстрактной фабрики** чаще всего реализуются с помощью **Фабричного метода**, хотя они могут быть построены и на основе **Прототипа**.
- Если **Команду** нужно копировать перед вставкой в историю выполненных команд, вам может помочь **Прототип**.

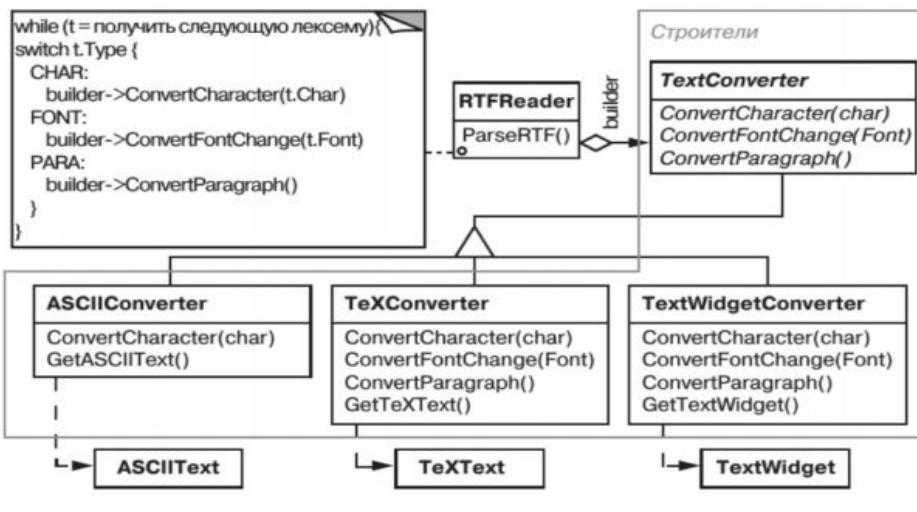
- Архитектура, построенная на **Компоновщиках** и **Декораторах**, часто может быть улучшена за счёт внедрения **Прототипа**. Он позволяет клонировать сложные структуры объектов, а не собирать их заново.
- **Прототип** не опирается на наследование, но ему нужна сложная операция инициализации. **Фабричный метод**, наоборот, построен на наследовании, но не требует сложной инициализации.
- **Снимок** иногда можно заменить **Прототипом**, если объект, состояние которого требуется сохранять в истории, довольно простой, не имеет активных ссылок на внешние ресурсы либо их можно легко восстановить.
- **Абстрактная фабрика**, **Строитель** и **Прототип** могут быть реализованы при помощи **Одиночки**.
- 

## 39. Паттерн «Строитель».

[преза](#) [конспект](#)

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

### Мотивация

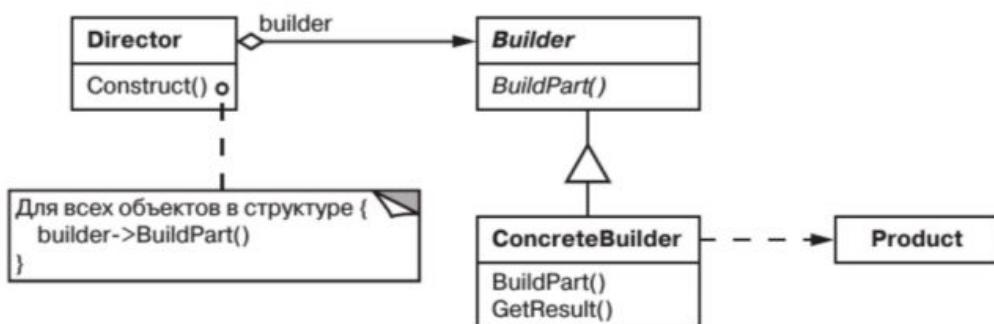


Программа, в которую заложена возможность распознавания и чтения документа в формате RTF (Rich Text Format), должна также «уметь» преобразовывать его во многие другие форматы, например в простой ASCIIтекст или в представление, которое можно отобразить в виджете для ввода текста. Однако число вероятных преобразований заранее неизвестно.

Поэтому должна быть обеспечена возможность без труда добавлять новый конвертор. Таким образом, нужно сконфигурировать класс RTFReader с помощью объекта TextConverter, который мог бы преобразовывать RTF в другой текстовый формат. При разборе документа в формате RTF класс RTFReader вызывает TextConverter для выполнения преобразования. Всякий раз, как RTFReader распознает лексему RTF (простой текст или управляющее слово), для ее преобразования объекту TextConverter посылается запрос. Объекты TextConverter отвечают как за преобразование данных, так и за

представление лексемы в конкретном формате. Подклассы TextConverter специализируются на различных преобразованиях и форматах. Например, ASCIIConverter игнорирует запросы на преобразование чего бы то ни было, кроме простого текста. С другой стороны, TeXConverter будет реализовывать все запросы для получения представления в формате редактора TEX, собирая по ходу необходимую информацию о стилях. А TextWidgetConverter станет строить сложный объект пользовательского интерфейса, который позволит пользователю просматривать и редактировать текст. Класс каждого конвертора принимает механизм создания и сборки сложного объекта и скрывает его за абстрактным интерфейсом. Конвертор отделен от загрузчика, который отвечает за синтаксический разбор RTFдокумента. В паттерне строитель абстрагированы все эти отношения. В нем любой класс конвертора называется строителем, а загрузчик – распорядителем. В применении к рассмотренному примеру строитель отделяет алгоритм интерпретации формата текста (то есть анализатор RTFдокументов) от того, как создается и представляется документ в преобразованном формате. Это позволяет повторно использовать алгоритм разбора, реализованный в RTFReader, для создания разных текстовых представлений RTFдокументов; достаточно передать в RTFReader различные подклассы класса TextConverter.

## Структура



### Участники

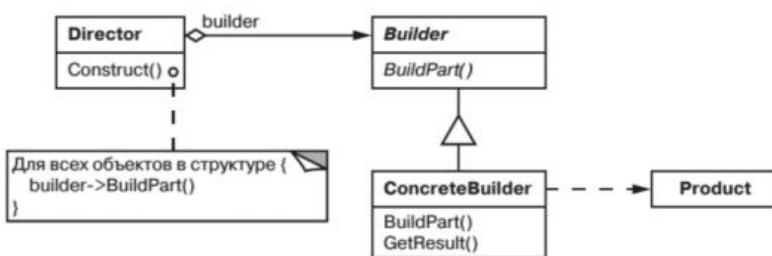
- ❑ **Builder** (TextConverter) – строитель:
  - задает абстрактный интерфейс для создания частей объекта **Product**;
- ❑ **ConcreteBuilder** (ASCIIConverter, TeXConverter, TextWidgetConverter) – конкретный строитель:
  - конструирует и собирает вместе части продукта посредством реализации интерфейса **Builder**;
  - определяет создаваемое представление и следит за ним;
  - предоставляет интерфейс для доступа к продукту (например, `GetASCIIText`, `GetTextWidget`);
- ❑ **Director** (RTFReader) – распорядитель:
  - конструирует объект, пользуясь интерфейсом **Builder**;
- ❑ **Product** (ASCIIText, TeXText, TextWidget) – продукт:
  - представляет сложный конструируемый объект. **ConcreteBuilder** строит внутреннее представление продукта и определяет процесс его сборки;
  - включает классы, которые определяют составные части, в том числе интерфейсы для сборки конечного результата из частей.

## Отношения

- клиент создает объектраспорядитель Director и конфигурирует его нужным объектомстроителем Builder;
- распорядитель уведомляет строителя о том, что нужно построить очередную часть продукта;
- строитель обрабатывает запросы распорядителя и добавляет новые части к продукту;
- клиент забирает продукт у строителя.

## Детали реализации

Обычно существует абстрактный класс Builder, в котором определены операции для каждого компонента, который распорядитель может «попросить» создать. По умолчанию эти операции ничего не делают. Но в классе конкретного строителя ConcreteBuilder они замещены для тех компонентов, в создании которых он принимает участие.



- ▶ Абстрактные и конкретные строители
  - ▶ Достаточно общий интерфейс
- ▶ Общий интерфейс для продуктов не требуется
  - ▶ Клиент конфигурирует распорядителя конкретным строителем, он же и забирает результат
- ▶ Пустые методы по умолчанию

## Отличия от других паттернов

- Многие архитектуры начинаются с применения **Фабричного метода** (более простого и расширяемого через подклассы) и эволюционируют в сторону **Абстрактной фабрики**, Прототипа или **Строителя** (более гибких, но и более сложных).
- Строитель концентрируется на построении сложных объектов шаг за шагом. **Абстрактная фабрика** специализируется на создании семейств связанных продуктов. Строитель возвращает продукт только после выполнения всех шагов, а Абстрактная фабрика возвращает продукт сразу же.
- Строитель позволяет пошагово сооружать дерево **Компоновщика**.

- Паттерн Строитель может быть построен в виде **Моста**: директор будет играть роль абстракции, а строители — реализации.
- Абстрактная фабрика, Строитель и Прототип могут быть реализованы при помощи **Одиночки**.

## 40. Паттерн «Посредник».

[преза](#) [конспект](#)

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

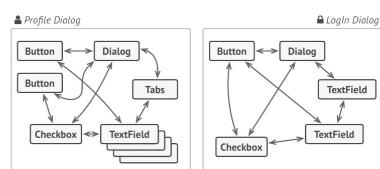
### Проблема

- Большое количество связей между объектами
- Объекты знают слишком много
- Снижается переиспользуемость

### Проблема из реальной задачи (диалог) (пример)

#### ⌚ Проблема

Предположим, что у вас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления — текстовых полей, чекбоксов, кнопок.



Беспорядочные связи между элементами пользовательского интерфейса.

Отдельные элементы диалога должны взаимодействовать друг с другом. Так, например, чекбокс «у меня есть собака» открывает скрытое поле для ввода имени домашнего любимица, а клик по кнопке отправки запускает проверку значений всех полей формы.



Код элементов нужно тратить при изменении каждого диалога.

Прописав эту логику прямо в коде элементов управления, вы поставите крест на их повторном использовании в других местах приложения. Они станут слишком тесно связанными с элементами диалога редактирования профиля, которые не нужны в других контекстах. Поэтому вы сможете использовать либо все элементы сразу, либо ни одного.

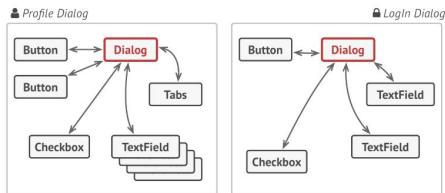
### Решение

Паттерн Посредник заставляет объекты общаться не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно перенаправить тот или иной

запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов

## 😊 Решение

Паттерн Посредник заставляет объекты общаться не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов.



Элементы интерфейса общаются через посредника.

В нашем примере посредником мог бы стать диалог. Скорее всего, класс диалога и так знает, из каких элементов

состоит, поэтому никаких новых связей добавлять в него не придётся.

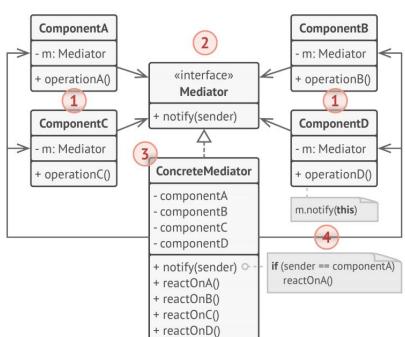
Основные изменения произойдут внутри отдельных элементов диалога. Если раньше при получении клика от пользователя объект кнопки сам проверял значения полей диалога, то теперь его единственной обязанностью будет сообщить диалогу о том, что произошёл клик. Получив извещение, диалог выполнит все необходимые проверки полей. Таким образом, вместо нескольких зависимостей от остальных элементов кнопка получит только одну — от самого диалога.

Чтобы сделать код ещё более гибким, можно выделить общий интерфейс для всех посредников, то есть диалогов программы. Наша кнопка станет зависимой не от конкретного диалога создания пользователя, а от абстрактного, что позволит использовать её и в других диалогах.

Таким образом, посредник скрывает в себе все сложные связи и зависимости между классами отдельных компонентов программы. А чем меньше связей имеют классы, тем проще их изменять, расширять и повторно использовать.

## Компоненты и структура

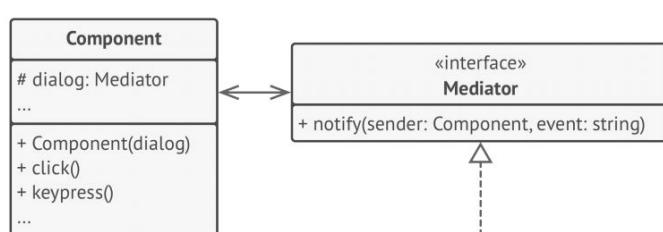
### STRUCTURE



ли события: ссылку на компонент, в котором оно произошло, и любые другие данные.

- 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.
- 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 дета-
- 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 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.

## Пример реализации диалога



## Применимость, преимущества и недостатки

- ▶ Устраняет связанность между классами-коллегами
- ▶ Повышает переиспользуемость классов-коллег
- ▶ Упрощает протоколы взаимодействия объектов
- ▶ Абстрагирует способ кооперирования объектов
- ▶ Централизует управление (потенциальный God Object!)

## Связи с другими паттернами

### ↔ Отношения с другими паттернами

- Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:
  - *Цепочка обязанностей* передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
- Когда вам сложно менять некоторые классы из-за того, что они имеют множество хаотичных связей с другими
  - *Команда* устанавливает косвенную одностороннюю связь от отправителей к получателям.
  - *Посредник* убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
  - *Наблюдатель* передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.
- Посредник и Фасад похожи тем, что пытаются организовать работу множества существующих классов.
  - *Фасад* создаёт упрощённый интерфейс к подсистеме, не внося в неё никакой добавочной функциональности. Сама подсистема не знает о существовании *Фасада*. Классы подсистемы общаются друг с другом напрямую.
  - *Посредник* централизует общение между компонентами системы. Компоненты системы знают только о существовании *Посредника*, у них нет прямого доступа к другим компонентам.
- Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.

Цель *Посредника* — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависи-

мыми от самого посредника. С другой стороны, цель *Наблюдателя* — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.

Довольно популярна реализация *Посредника* при помощи *Наблюдателя*. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.

Но *Посредник* имеет и другие реализации, когда отдельные компоненты жёстко привязаны к объекту посредника. Такой код вряд ли будет напоминать *Наблюдателя*, но всё же останется *Посредником*.

Напротив, в случае реализации посредника с помощью *Наблюдателя* представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети *Наблюдателей*, не имея центрального объекта-*Посредника*.

## 41. Паттерн «Команда».

[преза](#) [конспект](#)

**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.

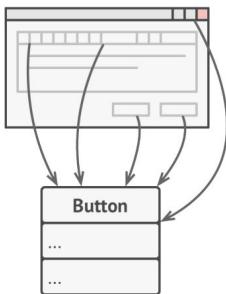
### Проблема (мотивация) (пример)

- ▶ Хотим отделить инициацию 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
- ▶ Но зачем?
- ▶ Команды меню приложения
- ▶ Палитры инструментов
- ▶ ...
- ▶ “Просто вызвать действие” не получится, вызов функции жёстко связывает инициатора и исполнителя

### Проблема из реальной задачи (текстовый редактор) (пример)

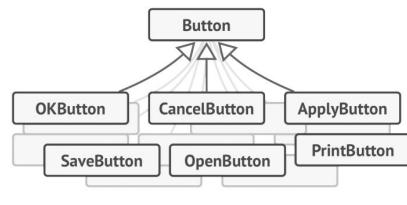
## :( Проблема

Представьте, что вы работаете над программой текстового редактора. Дело как раз подошло к разработке панели управления. Вы создали класс красивых кнопок и хотите использовать его для всех кнопок приложения, начиная от панели управления, заканчивая простыми кнопками в диалогах.



*Все кнопки приложения унаследованы от одного класса.*

Все эти кнопки, хоть и выглядят схоже, но делают разные вещи. Поэтому возникает вопрос: куда поместить код обработчиков кликов по этим кнопкам? Самым простым решением было бы создать подклассы для каждой кнопки и переопределить в них метод действия под разные задачи.



*Множество подклассов кнопок.*

Но скоро стало понятно, что такой подход никуда не годится. Во-первых, получается очень много подклассов. Во-вторых, код кнопок, относящийся к графическому интерфейсу, начинает зависеть от классов бизнес-логики, которая довольно часто меняется.



*Несколько классов дублируют одну и ту же функциональность.*

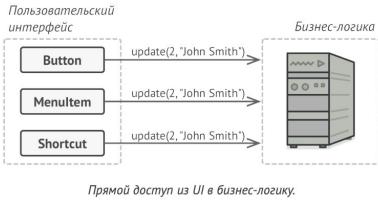
Но самое обидное ещё впереди. Ведь некоторые операции, например, «сохранить», можно вызывать из нескольких мест: нажав кнопку на панели управления, вызвав контекстное меню или просто нажав клавиши `Ctrl+S`. Когда в программе были только кнопки, код сохранения имелся только в подклассе `SaveButton`. Но теперь его придётся продублировать ещё в два класса.

## Решение

## 😊 Решение

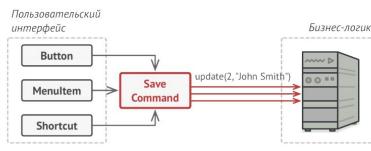
Хорошие программы обычно структурированы в виде слоёв. Самый распространённый пример – слои пользовательского интерфейса и бизнес-логики. Первый всего лишь рисует красивую картинку для пользователя. Но когда нужно сделать что-то важное, интерфейс «просит» слой бизнес-логики заняться этим.

В реальности это выглядит так: один из объектов интерфейса напрямую вызывает метод одного из объектов бизнес-логики, передавая в него какие-то параметры.



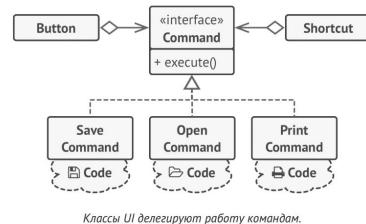
Паттерн Команда предлагает больше не отправлять такие вызовы напрямую. Вместо этого каждый вызов, отличающийся от других, следует завернуть в собственный класс с единственным методом, который и будет осуществлять вызов. Такие объекты называют **командами**.

К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та – позаботится обо всём остальном.



Классы команд можно объединить под общим интерфейсом с единственным методом запуска. После этого одни и те же отправители смогут работать с различными командами, не привязываясь к их классам. Даже больше: команды можно будет взаимозаменять на лету, изменения итоговое поведение отправителей.

Параметры, с которыми должен быть вызван метод объекта получателя, можно загодя сохранить в полях объекта команды. Благодаря этому, объекты, отправляющие запросы, могут не беспокоиться о том, чтобы собрать необходимые для получателя данные. Более того, они теперь вообще не знают, кто будет получателем запроса. Вся эта информация скрыта внутри команды.



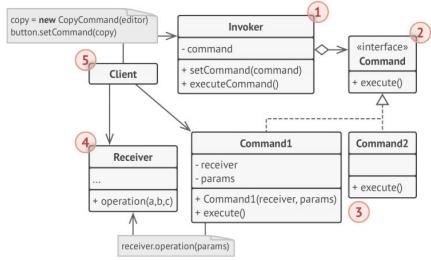
После применения Команды в нашем примере с текстовым редактором вам больше не потребуется создавать уйму подклассов кнопок под разные действия. Будет достаточно единственного класса с полем для хранения объекта команды.

Используя общий интерфейс команд, объекты кнопок будут ссылаться на объекты команд различных типов. При нажатии кнопки будут делегировать работу связанным командам, а команды – перенаправлять вызовы тем или иным объектам бизнес-логики.

Так же можно поступить и с контекстным меню, и с горячими клавишами. Они будут привязаны к тем же объектам команд, что и кнопки, избавляя классы от дублирования.

## Компоненты и структура

### STRUCTURE



1. **Отправитель** хранит ссылку на объект команды и обращается к нему, когда нужно выполнить какое-то действие. Отправитель работает с командами только через их общий интерфейс. Он не знает, какую конкретно команду использует, так как получает готовый объект команды от клиента.

Параметры, с которыми команда обращается к получателю, следует хранить в виде полей. В большинстве случаев объекты команд можно сделать неизменяемыми, передавая в них все необходимые параметры только через конструктор.

2. **Команда** описывает общий для всех конкретных команд интерфейс. Обычно здесь описан всего один метод для запуска команды.

Параметры, с которыми команда обращается к получателю, следует хранить в виде полей. В большинстве случаев объекты команд можно сделать неизменяемыми, передавая в них все необходимые параметры только через конструктор.

3. **Конкретные команды** реализуют различные запросы, следуя общему интерфейсу команд. Обычно команда не делает всю работу самостоятельно, а лишь передаёт вызов получателю, которым является один из объектов бизнес-логики.

Параметры, с которыми команда обращается к получателю, следует хранить в виде полей. В большинстве случаев объекты команд можно сделать неизменяемыми, передавая в них все необходимые параметры только через конструктор.

4. **Получатель** содержит бизнес-логику программы. В этой роли может выступать практически любой объект. Обычно команды перенаправляют вызовы получателям. Но иногда, чтобы упростить программу, вы можете избавиться от получателей, «слив» их код в классы команд.

Параметры, с которыми команда обращается к получателю, следует хранить в виде полей. В большинстве случаев объекты команд можно сделать неизменяемыми, передавая в них все необходимые параметры только через конструктор.

5. **Клиент** создаёт объекты конкретных команд, передавая в них все необходимые параметры, среди которых могут быть и ссылки на объекты получателей. После этого клиент связывает объекты отправителей с созданными командами.

## Применимость, плюсы и минусы

- ▶ Параметризовать объекты выполняемым действием
- ▶ Определять, ставить в очередь и выполнять запросы в разное время
- ▶ Поддержать отмену операций

- ▶ Структурировать систему на основе высокоуровневых операций, построенных из примитивных
- ▶ Поддержать протоколирование изменений

#### **⊕⊖ Преимущества и недостатки**

- ✓ Убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их непосредственно выполняют.
- ✓ Позволяет реализовать простую отмену и повтор операций.
- ✓ Позволяет реализовать отложенный запуск операций.
- ✓ Позволяет собирать сложные команды из простых.
- ✓ Реализует принцип открытости/закрытости.
- ✗ Усложняет код программы из-за введения множества дополнительных классов.

## **Связь с другими паттернами**

#### **↔ Отношения с другими паттернами**

- Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:
  - Цепочка обязанностей передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
  - Команда устанавливает косвенную одностороннюю связь от отправителей к получателям.
  - Посредник убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
  - Наблюдатель передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.
- Обработчики в Цепочке обязанностей могут быть выполнены в виде Команд. В этом случае множество разных операций может быть выполнено над одним и тем же контекстом, коим является запрос.
- Команду и Снимок можно использовать сообща для реализации отмены операций. В этом случае объекты команд будут отвечать за выполнение действия над объектом, а снимки будут хранить резервную копию состояния этого объекта, сделанную перед самым запуском команды.
- Команда и Стратегия похожи по духу, но отличаются масштабом и применением:
  - Команду используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее.
  - С другой стороны, Стратегия описывает разные способы произвести одно и то же действие, позволяя взаимозаменять эти способы в каком-то объекте контекста.
- Если Команду нужно копировать перед вставкой в историю выполненных команд, вам может помочь Прототип.
- Посетитель можно рассматривать как расширенный аналог Команды, который способен работать сразу с несколькими видами получателей.

Но есть и другой подход, в котором сам запрос является Командой, посланной по цепочке объектов. В этом случае одна и та же операция может быть выполнена над множеством разных контекстов, представленных в виде цепочки.

## **42. Паттерн «Цепочка ответственности».**

#### **преза конспект**

**Цепочка ответственности** — 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.

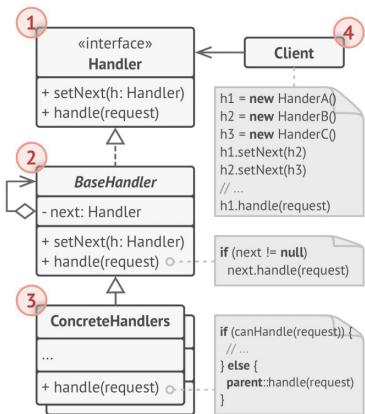
#### **Мотивация**

- Организация контекстной справки
- Если у элемента справки нет, запрос передаётся контейнеру
- Заранее неизвестно, кто в итоге обработает запрос

## Проблема (пример)

Сделать покупку книжки в онлайн магазине. Типа нужно проверить, что пользователь авторизован, потом есть ли у него промокод, тудым сюдым. Если делать не аккуратно, то связи между классами разрастутся и это будет плохо. Вот для этого нужна цепочка. (не увидел смысла сюда много информации вставлять, вроде и так все понятно)

## Структура



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.

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.

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

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

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.

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.

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.

## Применимость

### 💡 Применимость

**💡** Когда программа должна обрабатывать разнообразные запросы несколькими способами, но заранее неизвестно,

**💡** Когда набор объектов, способных обработать запрос, должен задаваться динамически.

какие конкретно запросы будут приходить и какие обработчики для них понадобятся.

**💡** С помощью Цепочки обязанностей вы можете связать потенциальных обработчиков в одну цепь и при получении запроса поочерёдно спрашивать каждого из них, не хочет ли он обработать запрос.

**💡** Когда важно, чтобы обработчики выполнялись один за другим в строгом порядке.

**💡** Цепочка обязанностей позволяет запускать обработчиков последовательно один за другим в том порядке, в котором они находятся в цепочке.

**💡** В любой момент вы можете вмешаться в существующую цепочку и переназначить связи так, чтобы убрать или добавить новое звено.

## Плюсы и минусы

### ⊕ Преимущества и недостатки

- ✓ Уменьшает зависимость между клиентом и обработчиками.
- ✓ Реализует *принцип единственной обязанности*.
- ✓ Реализует *принцип открытости/закрытости*.
- ✗ Запрос может остаться никем не обработанным.

### ► Ослабление связанности

- Дополнительная гибкость при распределении обязанностей
- Получение не гарантировано

Когда использовать:

- Есть более одного объекта-обработчика запросов
- Конечный обработчик неизвестен и должен быть найден автоматически
- Хотим отправить запрос нескольким объектам
- Обработчики могут задаваться динамически

## Связь с другими паттернами

### ↔ Отношения с другими паттернами

- **Цепочка обязанностей**, **Команда**, **Посредник** и **Наблюдатель** показывают различные способы работы отправителей запросов с их получателями:
  - Цепочка обязанностей передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
  - Команда устанавливает косвенную одностороннюю связь от отправителей к получателям.
  - Посредник убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
  - Наблюдатель передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.
- Цепочку обязанностей часто используют вместе с **Компоновщиком**. В этом случае запрос передаётся от дочерних компонентов к их родителям.
- Обработчики в Цепочке обязанностей могут выполнять произвольные действия, независимые друг от друга, а также в любой момент прерывать дальнейшую передачу по цепочке. С другой стороны Декораторы расширяют какое-то определённое действие, не ломая интерфейс базовой операции и не прерывая выполнение остальных декораторов.

Но есть и другой подход, в котором сам запрос является **Командой**, посланной по цепочке объектов. В этом случае одна и та же операция может быть выполнена над множеством разных контекстов, представленных в виде цепочки.

- Цепочка обязанностей и Декоратор имеют очень похожие структуры. Оба паттерна базируются на принципе рекурсивного выполнения операции через серию связанных объектов. Но есть и несколько важных отличий.

Обработчики в Цепочке обязанностей могут выполнять произвольные действия, независимые друг от друга, а также в любой момент прерывать дальнейшую передачу по цепочке. С другой стороны Декораторы расширяют какое-то определённое действие, не ломая интерфейс базовой операции и не прерывая выполнение остальных декораторов.

## 43. Паттерн «Наблюдатель».

[преза](#) [конспект](#)

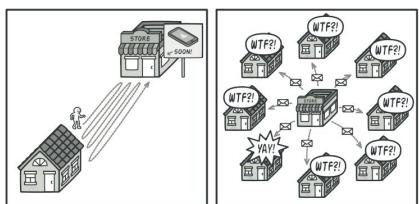
**Наблюдатель** — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах

## Проблема (пример) и решение:

### 😢 Проблема

Представьте, что вы имеете два объекта: Покупатель и Магазин . В магазин вот-вот должны завезти новый товар, который интересен покупателю.

Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время.



Постоянное посещение магазина или спам?

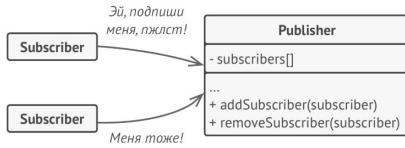
С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.

Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

### 😊 Решение

Давайте называть Издателями те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём Подписчиками .

Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.



Подписка на событие.

Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходить по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.

## Структура

- Издатель владеет внутренним состоянием, изменение которого интересно отслеживать подписчикам. Издатель содержит механизм подписки: список подписчиков и методы подписки/отписки.

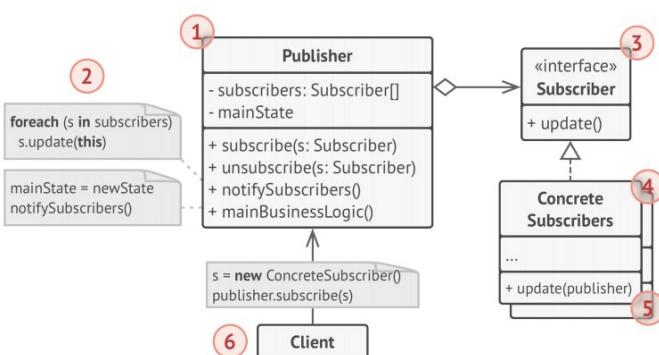
- Когда внутреннее состояние издателя меняется, он оповещает своих подписчиков. Для этого издатель проходит по списку подписчиков и вызывает их метод оповещения, заданный в общем интерфейсе подписчиков.

- Подписчик определяет интерфейс, которым пользуется издатель для отправки оповещения. В большинстве случаев для этого достаточно единственного метода.

- Конкретные подписчики выполняют что-то в ответ на оповещение, пришедшее от издателя. Эти классы должны следовать общему интерфейсу подписчиков, чтобы издатель не зависел от конкретных классов подписчиков.

- По приходу оповещения подписчику нужно получить обновлённое состояние издателя. Издатель может передать это состояние через параметры метода оповещения. Более гибкий вариант — передавать через параметры весь объект издателя, чтобы подписчик мог сам получить требуемые данные. Как вариант, подписчик может постоянно хранить ссылку на объект издателя, переданный ему в конструкторе.

- Клиент создаёт объекты издателей и подписчиков, а затем регистрирует подписчиков на обновления в издателях.



## Применимость, преимущества и недостатки

💡 Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.

⚡ Описанная проблема может возникнуть при разработке библиотек пользовательского интерфейса, когда вам надо дать возможность сторонним классам реагировать на клики по кнопкам.

Паттерн Наблюдатель позволяет любому объекту с интерфейсом подписчика зарегистрироваться на получение оповещений о событиях, происходящих в объектах-издателях.

💡 Когда одни объекты должны наблюдать за другими, но только в определённых случаях.

⚡ Издатели ведут динамические списки. Все наблюдатели могут подписываться или отписываться от получения оповещений прямо во время выполнения программы.

## Отношения с другими паттернами

### ➡ Отношения с другими паттернами

- Цепочка обязанностей, Команда, Посредник и Наблюдатель показывают различные способы работы отправителей запросов с их получателями:
  - Цепочка обязанностей передаёт запрос последовательно через цепочку потенциальных получателей, ожидая, что какой-то из них обработает запрос.
  - Команда устанавливает косвенную одностороннюю связь от отправителей к получателям.

### ⊕ Δ Преимущества и недостатки

- ✓ Издатели не зависят от конкретных классов подписчиков и наоборот.
- ✓ Вы можете подписывать и отписывать получателей на лету.
- ✓ Реализует принцип открытости/закрытости.
- ✗ Подписчики оповещаются в случайном порядке.

Цель *Посредника* — убрать обоюдные зависимости между компонентами системы. Вместо этого они становятся зависимыми от самого посредника. С другой стороны, цель *Наблюдателя* — обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других.

Довольно популярна реализация *Посредника* при помощи *Наблюдателя*. При этом объект посредника будет выступать издателем, а все остальные компоненты станут подписчиками и смогут динамически следить за событиями, происходящими в посреднике. В этом случае трудно понять, чем же отличаются оба паттерна.

Но *Посредник* имеет и другие реализации, когда отдельные компоненты жестко привязаны к объекту посредника. Такой код вряд ли будет напоминать *Наблюдателя*, но всё же останется *Посредником*.

- *Посредник* убирает прямую связь между отправителями и получателями, заставляя их общаться опосредованно, через себя.
- *Наблюдатель* передаёт запрос одновременно всем заинтересованным получателям, но позволяет им динамически подписываться или отписываться от таких оповещений.
- Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но иногда могут работать вместе.

Напротив, в случае реализации посредника с помощью *Наблюдателя* представим такую программу, в которой каждый компонент системы становится издателем. Компоненты могут подписываться друг на друга, в то же время не привязываясь к конкретным классам. Программа будет состоять из целой сети *Наблюдателей*, не имея центрального объекта-*Посредника*.

## 44. Паттерн «Состояние».

[преза](#) [конспект](#)

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

Паттерн Состояние невозможно рассматривать в отрыве от концепции машины состояний, также известной как [стейтмашина](#) или [конечный автомат](#).

### Проблема (пример)

Основная идея в том, что программа может находиться в одном из нескольких состояний, которые всё время сменяют друг друга. Набор этих состояний, а также переходов между ними, предопределён и конечен. Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые происходят с ней. Такой подход можно применить и к отдельным объектам. Например, объект Документ может принимать три состояния: Черновик , Модерация или Опубликован . В каждом из этих состояний метод опубликовать будет работать по-разному:

- Из черновика он отправит документ на модерацию.
- Из модерации — в публикацию, но при условии, что это сделал администратор.
- В опубликованном состоянии метод не будет делать ничего.

Машину состояний чаще всего реализуют с помощью множества условных операторов, if либо switch , которые проверяют текущее состояние объекта и выполняют соответствующее поведение. Наверняка вы уже реализовали хотя бы одну машину состояний в своей жизни, даже не зная об этом. Как насчёт вот такого кода, выглядит знакомо?

Основная проблема такой машины состояний проявится в том случае, если в Документ добавить ещё десяток состояний. Каждый метод будет состоять из увесистого условного оператора, перебирающего доступные состояния. Такой код крайне сложно поддерживать. Малейшее изменение логики переходов заставит вас перепроверять работу всех методов, которые содержат условные операторы машины состояний. Путаница и нагромождение условий особенно сильно проявляется в старых проектах.

Набор возможных состояний бывает трудно предопределить заранее, поэтому они всё время добавляются в процессе эволюции программы. Из-за этого решение, которое выглядело простым и эффективным в самом начале разработки, может впоследствии стать проекцией большого макаронного монстра.

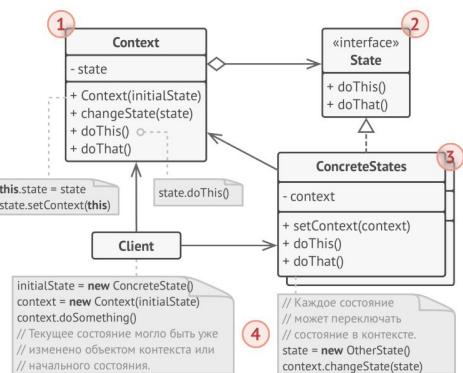
### Решение

Паттерн Состояние предлагает создать отдельные классы для каждого состояния, в котором может пребывать объект, а затем вынести туда поведения, соответствующие этим состояниям.

Вместо того, чтобы хранить код всех состояний, первоначальный объект, называемый контекстом, будет содержать ссылку на один из объектов-состояний и делегировать ему работу, зависящую от состояния.

Благодаря тому, что объекты состояний будут иметь общий интерфейс, контекст сможет делегировать работу состоянию, не привязываясь к его классу. Поведение контекста можно будет изменить в любой момент, подключив к нему другой объект-состояние. Очень важным нюансом, отличающим этот паттерн от [Стратегии](#), является то, что и контекст, и сами конкретные состояния могут знать друг о друге и инициировать переходы от одного состояния к другому.

## Структура



создавать целые иерархии классов состояний, чтобы обобщить дублирующий код.

Состояние может иметь обратную ссылку на объект контекста. Через неё не только удобно получать из контекста нужную информацию, но и осуществлять смену его состояния.

1. **Контекст** хранит ссылку на объект состояния и делегирует ему часть работы, зависящей от состояний. Контекст работает с этим объектом через общий интерфейс состояний. Контекст должен иметь метод для присваивания ему нового объекта-состояния.
2. **Состояние** описывает общий интерфейс для всех конкретных состояний.
3. **Конкретные состояния** реализуют поведения, связанные с определённым состоянием контекста. Иногда приходится
4. И контекст, и объекты конкретных состояний могут решать, когда и какое следующее состояние будет выбрано. Чтобы переключить состояние, нужно подать другой объект-состояние в контекст.

## Применимость

- Когда у вас есть объект, поведение которого кардинально меняется в зависимости от внутреннего состояния, причём типов состояний много, и их код часто меняется
- Когда код класса содержит множество больших, похожих друг на друга, условных операторов, которые выбирают поведения в зависимости от текущих значений полей класса.
- Когда вы сознательно используете табличную машину состояний, построенную на условных операторах, но вынуждены мириться с дублированием кода для похожих состояний и переходов

## Преимущества и недостатки

## Преимущества и недостатки

- ✓ Избавляет от множества больших условных операторов машины состояний.
- ✓ Концентрирует в одном месте код, связанный с определённым состоянием.
- ✓ Упрощает код контекста.

algavadev@gmail.com (#14892)

355 Поведенческие паттерны / Состояние

- ✗ Может неоправданно усложнить код, если состояний мало и они редко меняются.

## Отношения с другими паттернами

### Отношения с другими паттернами

- **Мост, Стратегия и Состояние** (а также слегка и **Адаптер**) имеют схожие структуры классов — все они построены на принципе «композиции», то есть делегирования работы другим объектам. Тем не менее, они отличаются тем, что решают разные проблемы. Помните, что паттерны — это не только рецепт построения кода определённым образом, но и описание проблем, которые привели к данному решению.
- **Состояние** можно рассматривать как надстройку над **Стратегией**. Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу вложенным объектам-помощникам. Однако в *Стратегии* эти объекты не знают друг о друге и никак не связаны. В *Состоянии* сами конкретные состояния могут переключать контекст.

## 45. Паттерн «Посетитель».

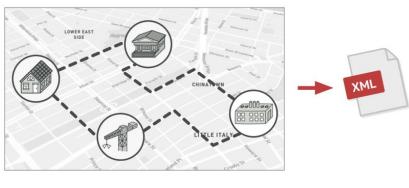
[преза](#) [конспект](#)

**Посетитель** — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

## Проблема (пример)

### Проблема

Ваша команда разрабатывает приложение, работающее с геоданными в виде графа. Узлами графа являются городские локации: памятники, театры, рестораны, важные предприятия и прочее. Каждый узел имеет ссылки на другие, ближайшие к нему узлы. Каждому типу узлов соответствует свой класс, а каждый узел представлен отдельным объектом.

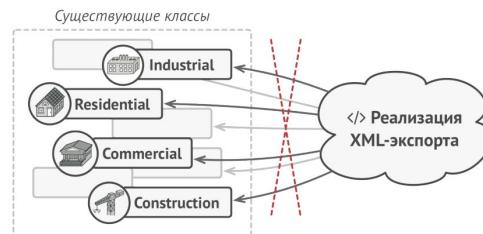


Экспорт геоузлов в XML.

Ваша задача — сделать экспорт этого графа в XML. Дело было бы плёвым, если бы вы могли редактировать классы узлов. Достаточно было бы добавить метод экспорта в каждый тип узла, а затем, перебирая узлы графа, вызывать этот метод для каждого узла. Благодаря полиморфизму, решение получилось бы изящным, так как вам не пришлось бы привязываться к конкретным классам узлов.

Но, к сожалению, классы узлов вам изменить не удалось. Системный архитектор сослался на то, что код классов узлов

сейчас очень стабилен, и от него многое зависит, поэтому он не хочет рисковать и позволять кому-либо его трогать.



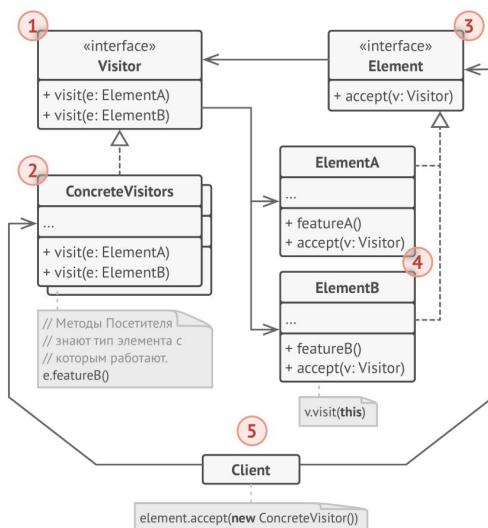
Код XML-экспорта придётся добавить во все классы узлов, а это слишком накладно.

К тому же он сомневался в том, что экспорт в XML вообще уместен в рамках этих классов. Их основная задача была связана с геоданными, а экспорт выглядит в рамках этих классов чужеродно. Была и ещё одна причина запрета. Если на следующей неделе вам бы понадобился экспорт в какой-то другой формат данных, то эти классы снова пришлось бы менять.

## Решение

Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя.

## Структура



1. Посетитель описывает общий интерфейс для всех типов посетителей. Он объявляет набор методов, отличающихся типом входящего параметра, которые нужны для запуска операции для всех типов конкретных элементов. В языках, поддерживающих перегрузку методов, эти методы могут иметь одинаковые имена, но типы их параметров должны отличаться.
2. Конкретные посетители реализуют какое-то особенное поведение для всех типов элементов, которые можно подать через методы интерфейса посетителя.
3. Элемент описывает метод принятия посетителя. Этот метод должен иметь единственный параметр, объявленный с типом интерфейса посетителя.
4. Конкретные элементы реализуют методы принятия посетителя. Цель этого метода — вызвать тот метод посещения, который соответствует типу этого элемента. Так посетитель узнает, с каким именно элементом он работает.
5. Клиентом зачастую выступает коллекция или сложный составной объект, например, дерево Компоновщика. Зачастую клиент не привязан к конкретным классам элементов, работая с ними через общий интерфейс элементов.

## Применимость

- Когда вам нужно выполнить какую-то операцию над всеми элементами сложной структуры объектов, например, деревом
- Когда над объектами сложной структуры объектов надо выполнять некоторые не связанные между собой операции, но вы не хотите «засорять» классы такими операциями.
- Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии

## Преимущества и недостатки

### Преимущества и недостатки

- ✓ Упрощает добавление операций, работающих со сложными структурами объектов.
- ✓ Объединяет родственные операции в одном классе.
- ✓ Посетитель может накапливать состояние при обходе структуры элементов.
- ✗ Паттерн не оправдан, если иерархия элементов часто меняется.
- ✗ Может привести к нарушению инкапсуляции элементов.

## Отношения с другими паттернами

### ↔ Отношения с другими паттернами

- Посетитель можно рассматривать как расширенный аналог Команды, который способен работать сразу с несколькими видами получателей.
- Вы можете выполнить какое-то действие над всем деревом Компоновщика при помощи Посетителя.
- Посетитель можно использовать совместно с Итератором. Итератор будет отвечать за обход структуры данных, а Посетитель — за выполнение действий над каждым её компонентом.

## 46. Паттерн «Хранитель».

[преза](#) [конспект](#)

**Хранитель (Снимок)** — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

### Мотивация

- ▶ Хотим уметь фиксировать внутреннее состояние объектов
- ▶ И восстанавливать его при необходимости
- ▶ Не раскрывая внутреннего устройства объектов кому не надо

### Проблема (пример)

Предположим, что вы пишете программу текстового редактора. Помимо обычного редактирования, ваш редактор позволяет менять форматирование текста, вставлять картинки и прочее.

В какой-то момент вы решили сделать все эти действия отменяемыми. Для этого вам нужно сохранять текущее состояние редактора перед тем, как выполнить любое действие. Если потом пользователь решит отменить своё действие, вы достанете копию состояния из истории и восстановите старое состояние редактора.

Чтобы сделать копию состояния объекта, достаточно скопировать значение его полей. Таким образом, если вы сделали класс редактора достаточно открытым, то любой другой класс сможет заглянуть внутрь, чтобы скопировать его состояние.

Казалось бы, что ещё нужно? Ведь теперь любая операция сможет сделать резервную копию редактора перед своим действием. Но такой наивный подход обеспечит вам уйму проблем в будущем. Ведь если вы решите провести рефакторинг — убрать или добавить парочку полей в класс редактора — то придётся менять код всех классов, которые могли копировать состояние редактора

Но это ещё не все. Давайте теперь рассмотрим сами копии состояния редактора. Из чего состоит состояние редактора? Даже самый примитивный редактор должен иметь несколько полей для хранения текущего текста, позиции курсора и прокрутки экрана. Чтобы сделать копию состояния, вам нужно записать значения всех этих полей в некий «контейнер». Скорее всего, вам понадобится хранить массу таких контейнеров в качестве истории операций, поэтому удобнее всего сделать их объектами одного класса. Этот класс должен иметь много полей, но практически никаких методов. Чтобы другие объекты могли записывать и читать из него данные, вам придётся сделать его поля публичными. Но это приведёт к той же проблеме, что и с открытым классом редактора. Другие классы станут зависимыми от любых изменений в классе контейнера, который подвержен тем же изменениям, что и класс редактора. Получается, нам придётся либо открыть классы для всех желающих, испытывая массу хлопот с поддержкой кода, либо оставить классы закрытыми, отказавшись от идеи отмены операций. Нет ли какого-то другого пути?

## Решение

Все проблемы, описанные выше, возникают из-за нарушения инкапсуляции. Это когда одни объекты пытаются сделать работу за других, влезая в их приватную зону, чтобы собрать необходимые для операции данные.

Паттерн Снимок поручает создание копии состояния объекта самому объекту, который этим состоянием владеет. Вместо того, чтобы делать снимок «извне», наш редактор сам сделает копию своих полей, ведь ему доступны все поля, даже приватные.

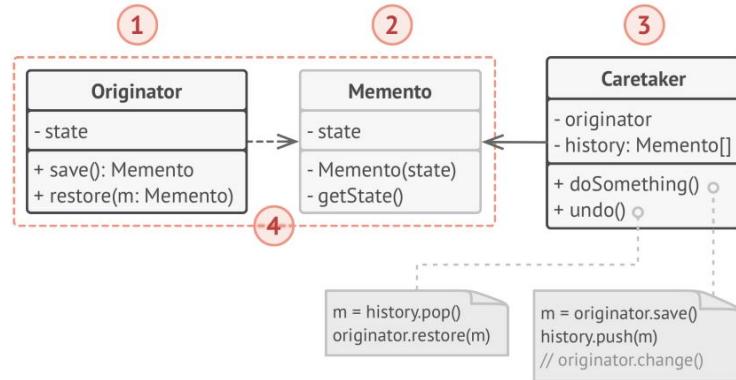
Паттерн предлагает держать копию состояния в специальном объекте-снимке с ограниченным интерфейсом, позволяющим, например, узнать дату изготовления или название снимка. Но, с другой стороны, снимок должен быть открыт для своего создателя, позволяя прочесть и восстановить его внутреннее состояние

Такая схема позволяет создателям производить снимки и отдавать их для хранения другим объектам, называемым опекунами. Опекунам будет доступен только ограниченный интерфейс снимка, поэтому они никак не смогут повлиять на «внутренности» самого снимка. В нужный момент опекун может попросить создателя восстановить своё состояние, передав ему соответствующий снимок.

В примере с редактором вы можете сделать опекуном отдельный класс, который будет хранить список выполненных операций. Ограниченный интерфейс снимков позволит демонстрировать пользователю красивый список с названиями и датами выполненных

операций. А когда пользователь решит откатить операцию, класс истории возьмёт последний снимок из стека и отправит его объекту редактор для восстановления.

## Структура



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.

## Применимость

Когда вам нужно сохранять мгновенные снимки состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии

Когда прямое получение состояния объекта раскрывает приватные детали его реализации, нарушая инкапсуляцию.

## Преимущества и недостатки

### ⊕⊖ Преимущества и недостатки

- ✓ Не нарушает инкапсуляции исходного объекта.
- ✓ Упрощает структуру исходного объекта. Ему не нужно хранить историю версий своего состояния.

algsavelev@gmail.com (#14892)

324 Поведенческие паттерны / Снимок

- ✗ Требует много памяти, если клиенты слишком часто создают снимки.
- ✗ Может повлечь дополнительные издержки памяти, если объекты, хранящие историю, не освобождают ресурсы, занятые устаревшими снимками.
- ✗ В некоторых языках (например, PHP, Python, JavaScript) сложно гарантировать, чтобы только исходный объект имел доступ к состоянию снимка.

## Отношения с другими паттернами

### ↔ Отношения с другими паттернами

- **Команду** и **Снимок** можно использовать сообща для реализации отмены операций. В этом случае объекты команд будут отвечать за выполнение действия над объектом, а снимки будут хранить резервную копию состояния этого объекта, сделанную перед самым запуском команды.
- **Снимок** можно использовать вместе с **Итератором**, чтобы сохранить текущее состояние обхода структуры данных и вернуться к нему в будущем, если потребуется.
- **Снимок** иногда можно заменить **Прототипом**, если объект, состояние которого требуется сохранять в истории, довольно простой, не имеет активных ссылок на внешние ресурсы либо их можно легко восстановить.

## 47. Паттерн «Интерпретатор».

[преза](#) [конспект](#)(стр.237)

[еще ссылка](#)

### Назначение паттерна Interpreter

- Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.
- Отражает проблемную область в языке, язык – в грамматику, а грамматику – в иерархии объектно-ориентированного проектирования.

### Решаемая проблема

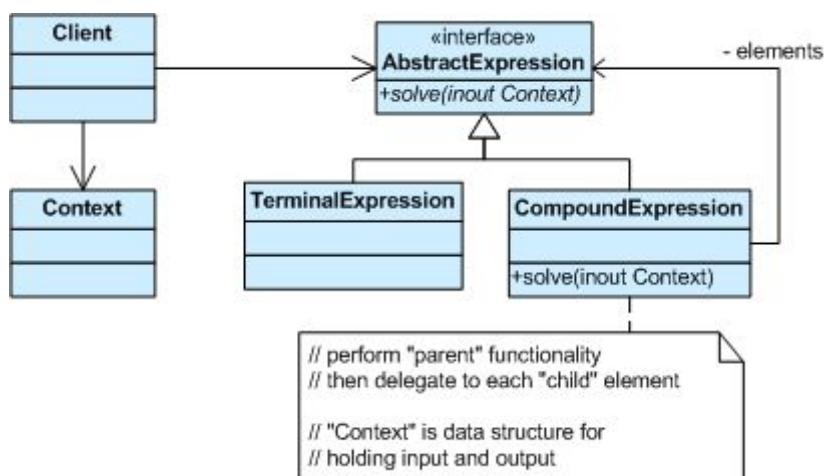
Пусть в некоторой, хорошо определенной области периодически случается некоторая проблема. Если эта область может быть описана некоторым “языком”, то проблема может быть легко решена с помощью “интерпретирующей машины”.

### Структура паттерна Interpreter

Паттерн Interpreter моделирует проблемную область с помощью рекурсивной грамматики. Каждое грамматическое правило может быть либо составным (правило ссылается на другие правила) либо терминальным (листовой узел в структуре “дерево”).

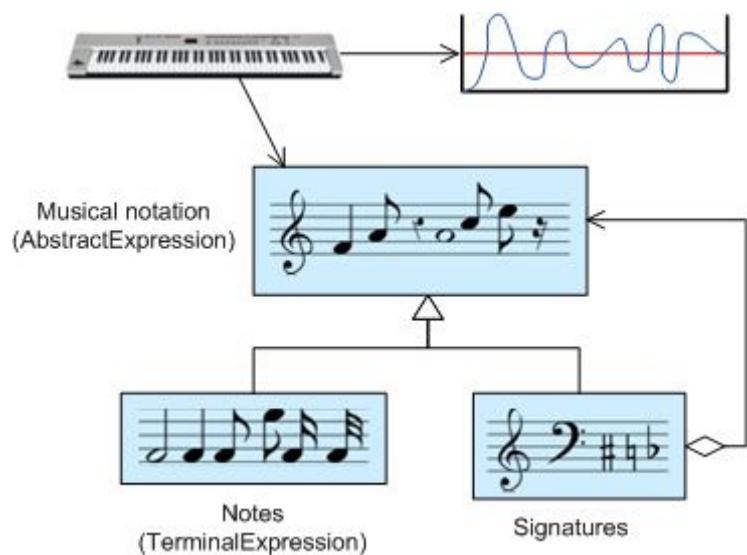
Для рекурсивного обхода “предложений” при их интерпретации используется [паттерн Composite](#).

### UML-диаграмма классов паттерна Interpreter



## Пример паттерна Interpreter

Паттерн Interpreter определяет грамматическое представление для языка и интерпретатор для интерпретации грамматики. Музыканты являются примерами интерпретаторов. Тональность и продолжительность звуков могут быть описаны нотами. Такое представление является музыкальным языком. Музыканты, используя ноты, способны воспроизвести оригинальные частоту и длительность каждого представленного звука.



## Использование паттерна Interpreter

1. Определите “малый” язык, “инвестиции” в который будут оправданы.
2. Разработайте грамматику для языка.
3. Для каждого грамматического правила (продукции) создайте свой класс.
4. Полученный набор классов организуйте в структуру с помощью паттерна Composite.
5. В полученной иерархии классов определите метод `interpret(Context)`.
6. Объект `Context` инкапсулирует информацию, глобальную по отношению к интерпретатору.  
Используется классами во время процесса “интерпретации”.

## Особенности паттерна Interpreter

- Абстрактное синтаксическое дерево интерпретатора – пример паттерна [Composite](#).
- Для обхода узлов дерева может применяться паттерн [Iterator](#).
- Терминальные символы могут разделяться с помощью [Flyweight](#).
- Паттерн Interpreter не рассматривает вопросы синтаксического разбора. Когда грамматика очень сложная, должны использоваться другие методики.

## 48. Паттерн «Спецификация».

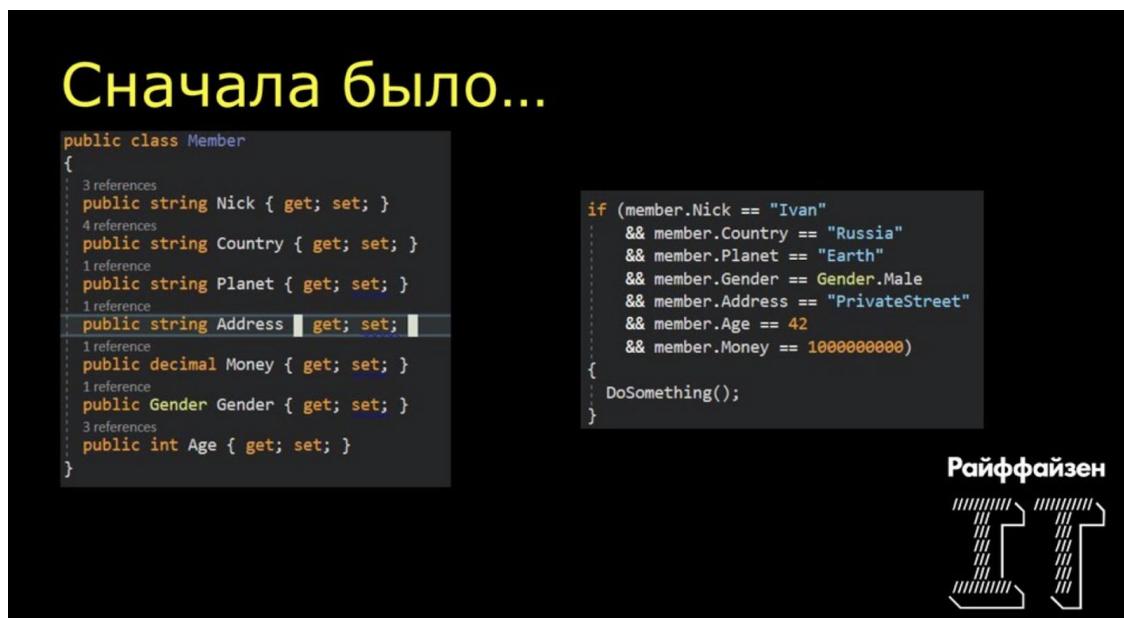
кажется нет презы и конспекта тоже...

[ссылка](#)

«Спецификация» в программировании — это шаблон проектирования, посредством которого представление правил бизнес логики может быть преобразовано в виде цепочки объектов, связанных операциями булевой логики.

### Проблема

Пусть у нас есть в классе некоторая сущность (как на картинке Address) и ее нужно проверять. Как видно на картинке справа может получаться вот такой большой if. Если будет много таких проверок, то будет плохо. И что-то изменять будет очень тяжело (отладка и изменения очень тяжелый и душный процесс)



Получается, что нужно придумать способ, которым можно нормально проверять такие вот цепочки булевых операций, который поступают, например из бд. Т.е. нужно сделать фильтр объектов. Вот для этого нужна спецификация.

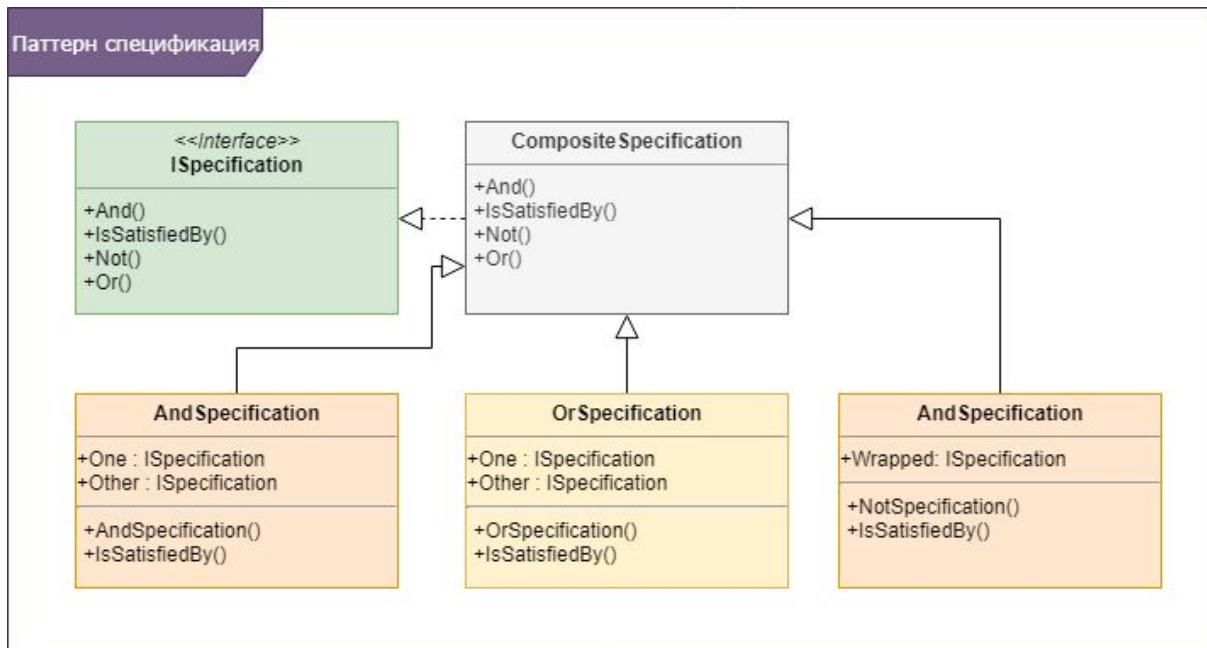
### Еще одно определение

The image shows a presentation slide with a dark background. The title of the slide is '1 Определение паттерна' (1 Definition of the pattern). Below the title, there is a text block that defines the 'Specification' pattern as a template for specifying business rules. At the bottom of the slide, there is a snippet of code for an 'ISpecification<T>' interface with a single method 'IsSatisfiedBy(T entity)'.

## Применение

- фильтрация объектов
- уменьшение дублирования кода
- в случае изменения бизнес требования нужно будет изменить запрос в одном месте, не меняя кучу кода

## UML



## 49. Паттерн «Итератор».

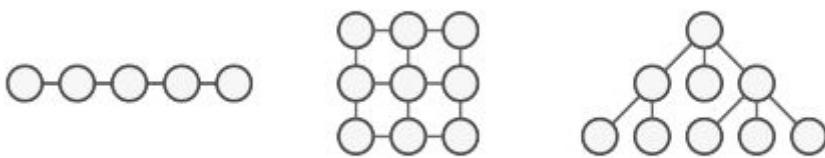
[преза](#) [конспект](#)

### Суть паттерна

Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

### Проблема

Коллекции — самая распространённая структура данных, которую вы можете встретить в программировании. Это набор объектов, собранный в одну кучу по каким-то критериям.



*Разные типы коллекций.*

Но как бы ни была структурирована коллекция, пользователь должен иметь возможность последовательно обходить её элементы, чтобы проделывать с ними какие-то действия.

Но каким способом следует перемещаться по сложной структуре данных? Например, сегодня может быть достаточным обход дерева в глубину, но завтра потребуется возможность перемещаться по дереву в ширину.

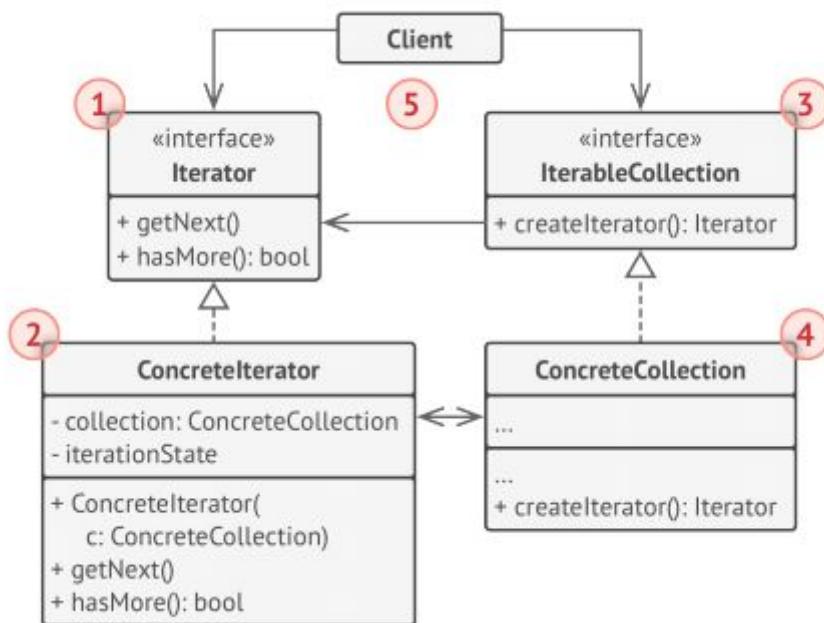
Добавляя всё новые алгоритмы в код коллекции, вы понемногу размываете её основную функцию, которая заключается в эффективном хранении данных.

## Решение

Идея паттерна Итератор состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс. Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом. К тому же, если вам понадобится добавить новый способ обхода, вы сможете создать отдельный класс итератора, не изменяя существующий код коллекции.

## Структура

1. **Итератор** описывает интерфейс для доступа и обхода элементов коллекции.
2. **Конкретный итератор** реализует алгоритм обхода какой-то конкретной коллекции. Объект итератора должен сам отслеживать текущую позицию при обходе коллекции, чтобы отдельные итераторы могли обходить одну и ту же коллекцию независимо.
3. **Коллекция** описывает интерфейс получения итератора из коллекции. Как мы уже говорили, коллекции не всегда являются списком. Это может быть и база данных, и удалённое API, и даже дерево Компоновщика. Поэтому сама коллекция может создавать итераторы, так как она знает, какие именно итераторы способны с ней работать.
4. **Конкретная коллекция** возвращает новый экземпляр определённого конкретного итератора, связав его с текущим объектом коллекции. Обратите внимание, что сигнатура метода возвращает интерфейс итератора. Это позволяет клиенту не зависеть от конкретных классов итераторов.
5. **Клиент** работает со всеми объектами через интерфейсы коллекции и итератора. Так клиентский код не зависит от конкретных классов, что позволяет применять различные итераторы, не изменяя существующий код программы. В общем случае клиенты не создают объекты итераторов, а получают их из коллекций. Тем не менее, если клиенту требуется специальный итератор, он всегда может создать его самостоятельно.



### Применимость

- Когда у вас есть сложная структура данных, и вы хотите скрыть от клиента детали её реализации (из-за сложности или вопросов безопасности).
- Когда вам нужно иметь несколько вариантов обхода одной и той же структуры данных.
- Когда вам хочется иметь единый интерфейс обхода различных структур данных.

### Преимущества и недостатки

+Упрощает классы хранения данных.

+Позволяет реализовать различные способы обхода структуры данных.

+Позволяет одновременно перемещаться по структуре данных в разные стороны.

-Не оправдан, если можно обойтись простым циклом.

## 50. Антипаттерны «Круговая зависимость», «Последовательная связность», «Вызов предка», «Проблема Йо-Йо».

В литературе есть термин “**Антипаттерны**”, имеющий тот же смысл, что и паттерны, но наоборот: если паттерны — это часто встречающиеся решения, приводящие к

известным преимуществам, то антипаттерны — это часто встречающиеся решения, приводящие к известным проблемам. Оказывается, что знание антипаттернов для архитектора даже полезнее, чем знание паттернов, поскольку позволяет избежать очень частых и очень дорогостоящих ошибок.

**Circular dependency, или “круговая зависимость”** — это когда два или больше компонентов зависят друг от друга циклически, например, первый зависит от второго, второй от третьего, а третий от первого. Появляется чаще всего от незнания принципа Dependency Inversion или от неудачных попыток добавить callback-и, чтобы организовать реально двухстороннее общение компонент. Обратите внимание, что компоненты, обменивающиеся данными в обе стороны — это не Circular dependency, это нормально. Circular dependency — это круговая зависимость во время компиляции или круговая зависимость в графе вызовов.

Как бороться: прежде всего применением принципа Dependency Inversion, пятого принципа SOLID. Если есть круговая зависимость, сделайте так, чтобы обе абстракции, зависящие друг от друга, зависели от интерфейсов и реализовывали соответствующие интерфейсы. Также тогда можно будет применить Dependency Injection для автоматического внедрения зависимостей и конфигурирования системы, что не обязательно, но приятное дополнение к более аккуратной архитектуре. Поскольку чаще всего круговые зависимости связаны с необходимостью вызова колбэков, может помочь паттерн «Наблюдатель» — чтобы один компонент подписывался на события другого, а не дёргался другим напрямую. Ещё может помочь более аккуратно разбить функциональность на слои, чтобы вызовы могли выполняться только от верхних слоёв к нижним (и использовать «Наблюдатель» в обратную сторону), но это сложно сделать пост-фактум. Зато если вы изначально проектируете систему в слоистом стиле, вероятность появления круговых зависимостей будет существенно меньше.

**Sequential coupling, или «последовательная связность»** — необходимость вызывать методы в строго определённом порядке, ошибившись в котором можно всё сломать, или даже просто необходимость обязательно вызвать второй метод, если вызван первый. Антипаттерн это потому, что компилятор не в состоянии это проверить, а всё, что не проверяется компилятором, рано или поздно сделают не так. Самый распространённый пример — метод `init()`, который обязательно надо вызвать после конструктора объекта, иначе тот будет в неконсистентном состоянии. Когда-нибудь его забудут вызывать практически наверняка.

Решения проблемы последовательной связности в порядке от самых тактических до самых стратегических:

- проектировать интерфейсы абстракций так, чтобы все операции были «атомарны», то есть делали всю работу за раз;
- использовать паттерн «шаблонный метод», где инкапсулировать эту самую необходимую последовательность, чтобы прикладные программисты о ней не думали;
- использовать паттерны «абстрактная фабрика» или «строитель», чтобы там спрятать сложность инициализации подсистемы в правильном порядке;

- использовать Dependency Injection, чтобы переложить порядок инициализации вообще на стороннюю библиотеку — но помните, что Dependency Injection имеет и свои недостатки, связанные со сложностью отладки и отсутствием контроля во время компиляции.

**Call Super, или «вызов предка»** — это необходимость вызвать из переопределённого метода потомка переопределяемый метод предка. Плохо это, как и в предыдущем антипаттерне, потому, что компилятор не может проверить, вызвали вы метод предка или нет.

Для борьбы с этим антипаттерном особо эффективен «шаблонный метод» — тут как раз уже есть предок, где его можно разместить. Шаблонный метод позволяет переопределить поведение родителя, а не требует этого, к тому же предоставляет фиксированные точки расширения, позволяющие выполнить это определение и точно ничего не сломать. Это, конечно, уменьшает гибкость (и поэтому в GUI-библиотеках так обычно не делают), но увеличивает надёжность системы (и поэтому в GUI-библиотеках так всё-таки делают иногда).

**Yo-yo problem, или «проблема йо-йо»** — когда у нас поток управления, как йо-йо, передаётся от потомка к предку, от предка к потомку, от потомка к предку и т.д. Такие ситуации не редки в продакшн-коде и являются следствиями благих намерений — красивой объектно ориентированной архитектуры, где всё общее поведение вынесено в предка, а потомки его только параметризуют своими действиями.

Проблема это потому, что чтобы понять, как работает каждый класс в иерархии наследования, вам надо понять всю иерархию наследования одновременно, нет чёткого разделения ответственности между потомками и предками, и при отладке управление прыгает между файлами.

Если проблема йо-йо не очень запущенна, с ней можно бороться, просто перераспределив функциональность между предками и потомками, так, чтобы вызовы всегда были в каком-то одном направлении. В более тяжёлых случаях потребуется разделить иерархию наследования на несколько. На самом деле, вызовы предка — это обычно вызовы каких-то вспомогательных методов, которые нужны всем потомкам и находятся в предке просто для того, чтобы быть им доступными. Тогда можно вынести такие методы в отдельный класс. Если поведение таких методов тоже бывает разным в зависимости от чего-то, поможет паттерн «Мост».

Проблема йо-йо становится критичной, если иерархия наследования достаточно глубока, и вообще не возникает, если классы не наследуются друг от друга, а только реализуют интерфейсы. Так что хороший способ избежать проблемы йо-йо — избегать иерархий наследования глубины больше трёх, и вообще использовать наследование только как механизм полиморфных вызовов, используя композицию как способ переиспользовать общую функциональность.

## 51. Антипаттерны «Активное ожидание», «Скрытие ошибки», «Магические числа», «Магические строки».

**Busy waiting**, или «активное ожидание» — это ситуация, когда мы ожидаем наступления какого-то события в цикле, постоянно проверяя, не наступило ли оно. Вариант этого антипаттерна — использование циклов для задержки: в конце концов, подождать N миллисекунд можно, выполнив M итераций цикла for с пустым телом. Плохо это из-за того, что процессор-то не знает, что цикл ожидания на самом деле ничего не делает, и старается исполнить его с максимальной возможной скоростью, полностью занимая одно из ядер.

Тем не менее, это не всегда плохо, а во встроенных системах может быть вполне валидным способом организации работы — например, для опроса датчиков.

Процессоры же обычных настольных компьютеров или ноутбуков, напротив, спроектированы с учётом того, что будут применяться в основном в интерактивных приложениях, где 99% времени приходится ждать пользователя, так что работа на полную мощность для них скорее исключение, чем правило. Тем не менее, даже для обычных процессоров активное ожидание иногда применяется по делу — например, примитив синхронизации SemaphoreSlim в .NET перед тем, как заблокировать поток с помощью планировщика, ждёт в активном ожидании несколько (скорее несколько тысяч) циклов в надежде, что критическая область быстро освободится. Поскольку вызов планировщика трудозатратен, а критические области часто бывают короткими, это позволяет сделать синхронизацию в разы эффективней. Тем не менее, активное ожидание очень часто происходит по ошибке, особенно в коде людей, только познакомившихся с принципами многопоточного программирования.

Бороться с этим антипаттерном можно с помощью планировщика, если он есть на целевой машине. Любая нормальная ОС с планировщиком имеет системные вызовы, позволяющие усыпить поток до наступления какого-то события. Если планировщика нет, то есть аппаратные возможности организации асинхронного исполнения — таймеры и аппаратные прерывания. Если и этого нет, то наверняка железо как раз заточено под активное ожидание и будет работать с ним вполне нормально.

**Error hiding**, или «скрытие ошибки» — ситуация, когда сообщение об ошибке показывается без какой-либо диагностической информации или не показывается вовсе. Так происходит, естественно, из благих намерений не пугать пользователя, и в надежде, что проблема некритична и если о ней не сообщать, то её и не заметят. Так часто делают продукты, нацеленные на массового покупателя — «Oops, something went wrong» в браузере является хорошим примером такого поведения. Антипаттерном это становится в том случае, когда диагностику оказывается вообще невозможно получить.

Бороться с этим антипаттерном довольно просто, но сложно психологически. Надо убедить себя, что пользователи не придут с вилами и факелами к офису вашей компании, если программа будет изредка падать, а наоборот, будут писать багрепорты, что позволит быстро выявить, локализовать и устраниить проблемы с кодом.

Есть известный принцип «Fail fast», рекомендующий заканчивать работу, как только выявлена проблема. Чем быстрее вы «Fail», тем больше у вас будет попыток сделать

что-нибудь хорошее. Дайте программе упасть. Помогите программе упасть, добавив в код как можно больше различных проверок. Чем быстрее код упадёт, тем больше шансов, что проблема будет выявлена на этапе тестирования, но если она всё-таки просколит QA, тем быстрее пользователи о ней сообщат и вы сможете её поправить.

Ещё хорошее правило — логировать все интересные вещи, происходящие с программой. Как минимум, все исключения, но можно и внешние события, и крупные изменения состояния, чтобы понять по логу, что произошло.

**Magic numbers, Magic strings**, или «магические числа», «магические строки» — это целых два антипаттерна, похожих, но имеющих несколько разные последствия. **Магические числа** — это практически все встречающиеся в коде числовые литералы, за исключением 0 и иногда 1. Плохи они тем, что вы можете не иметь идей, почему это число именно такое, как написано в коде, и если надо что-то поменять, то какие именно числа в коде надо менять.

**Магические строки** — это, соответственно, строковые литералы в коде. Их, внезапно, тоже быть не должно. Строки бывают двух крупных категорий — те, которые показываются пользователю, и те, которые используются только внутри кода. Строки, которые показываются пользователю, надо локализовывать. Локализация требует массы усилий, а если о ней заранее не подумать, усилий требуется десятикратно больше.

Строки, которые пользователю не показываются, тоже не стоит оставлять в виде литералов, потому что можно в строке опечататься, она где-нибудь с чем-нибудь не сравнится и всё пойдёт не так. Выносите строки в константы, если они вам нужны, или старайтесь от таких строк вообще избавиться.

Бороться с магическими числами очень просто — вынесите их в именованные константы . С магическими строками сложнее — если они не показываются пользователю, то тоже константы. Если показываются, то надо использовать технологию переводов, поддержанную в вашем технологическом стеке.

## 52. Антипаттерны «Божественный объект», «Поток лавы».

Перейдём к более стратегическим антипаттернам, оказывающим влияние на всю архитектуру системы. Первый и наиболее известный из них — это **God Object** (или «божественный объект»). God Object — это когда в системе (или в компоненте) всем процессом вычислений управляет один класс, знает про все остальные и пользуется ими просто как хранилищем данных. Почему это плохо — потому что такой класс имеет тенденцию иметь впечатляющие размеры и чрезвычайную сложность, сопровождать его очень тяжело, а переиспользовать и вовсе невозможно.

Божественные объекты обычно сурово нарушают принцип единственности ответственности, представляя собой хаотичное объединение разных ответственостей в один класс. Часто это выражается в большом количестве полей и методов.

Вот что можно сделать, если у вас завёлся God Object.

- Передать больше ответственности классам-данным. Они ведь наверняка могут проявить некоторую самостоятельность и делать часть работы сами. Затем можно

посмотреть, что ещё можно из божественного объекта передать классам-данным, и продолжать в таком духе, пока не получится нормальная архитектура.

- Разделить методы класса на группы, соответствующие контрактам, выполняемым God Object-ом. Возможно, уже существуют классы, в которые можно перенести эти группы, например, те же классы-данные, либо какие-то ещё классы системы.  
Возможно, нет, тогда их надо создать, так, чтобы каждый класс отвечал за что-то одно (принцип единственности ответственности всегда должен направлять рефакторинг).
- Убрать непрямые зависимости: если объекты A и B лежат внутри объекта G, может так случиться, что A может быть в B, а B — в G. Это позволит God Object-у G меньше знать, что всегда позитивно.

**Lava Flow или «поток лавы».** Суть антипаттерна в том, что эксперименты, быстрый поиск решения и прототипирование выполняются прямо в основном коде, после чего, когда решение найдено, остатки этих экспериментов не вычищаются должным образом и застывают в коде навсегда — потому что трогать их страшно, а вдруг это кому-то всё-таки нужно и если тронуть, то сломается.

В более общем виде Lava Flow представляет собой любой незакрытый технический долг, постепенно накапливающийся в системе, избавиться от которого очень сложно, даже если это просто мёртвый код. Появляется он, как правило, как быстрые фиксы к каким-то насущным проблемам, которые потом никогда не заменяются нормальным решением.

Lava Flow проще предупредить (насколько это возможно), чем от него потом избавиться. Все эксперименты, concept proof, варианты решения, «быстро попробовать кое-что» и т.д. и т.п. следует всегда делать в отдельных ветках, которые потом безжалостно прибивать, получив нужное знание.

Если всё-таки Lava Flow уже в проекте, причём серьёзно мешает разработке, требуются весьма радикальные меры. Если удаётся его победить небольшими рефакторингами, то хорошо — но помните, что постепенная борьба с техническим долгом будет неизбежно приводить к новым багам, которые нельзя чинить новыми костылями. Кроме того, убиение одного простого костыля на пару строчек приводит зачастую к перепроектированию большого куска приложения, чтобы «сделать нормально». Если такого перепроектирования требуется выполнять много, то имеет смысл провести полноценный архитектурный реинжиниринг системы — проанализировать архитектуру как она есть и разработать архитектуру как она должна быть, плюс план наиболее безболезненного перехода от одного к другому. Это может быть очень долгий и сложный процесс, который, к тому же, приводит к полной остановке разработки — нет смысла реализовывать новую функциональность илификсить баги, если система всё равно будет практически переписана в ближайшем будущем. Бывали ситуации, когда такой процесс занимал в индустриальных проектах до полугода сконцентрированных усилий. Как вариант всегда стоит рассматривать идею «выкинуть всё и написать заново».

### 53. Антипаттерны «Функциональная декомпозиция», «Полтергейст», «Золотой молоток».

**Functional Decomposition**, или «функциональная декомпозиция» — антипаттерн, заключающийся в проектировании объектно-ориентированной системы в функциональном стиле. Характеризуется архитектурой, построенной на вызывающих друг друга функциях, которые постепенно декомпозирируют и решают задачу. Обратите внимание, что это антипаттерн только в том случае, если вы программируете на объектно-ориентированном языке.

Если вдруг такая проблема возникла, что можно сделать:

- вернуться к требованиям, попробовать построить объектно-ориентированную модель предметной области «с нуля»;
- потом надо отобразить эту модель на уже существующий код, в духе «вот этот метод этого объекта у нас реализуется такой-то и такой-то функцией». На этом этапе не советуют кидаться рефакторить код, потому что запутаешься, цель — задокументировать и привязать к требованиям существующий код, чтобы потом понять, что с ним делать.
- Собственно рефакторинг можно начинать с лёгких целей — классы с одним методом можно прибить, переместив код из них в другие классы.
- Затем кластеризовать то, что получилось, в группы методов, которые соответствуют классам из предметной области, и сделать их полноценными классами.
- Если в классе нет состояния, можно сделать его статическим классом, в надежде, что состояние у него потом заведётся в процессе рефакторинга само собой. Если нет, можно перенести код в какой-нибудь из настоящих классов, либо решить для себя, что это на самом деле класс-сервис, у которого состояния быть и не должно. Так бывает, например, когда класс выполняет операции над двумя другими классами и каждый из этих классов играет равную роль в этой операции (так что её нельзя по смыслу перенести в один из этих классов).
- Есть ещё один радикальный метод борьбы с запущенной функциональной декомпозицией — свалит весь код в одну кучу, сделав God Object. А с God Object мы уже знаем как бороться.

Наконец, самый популярный в программистском сообществе антипаттерн — **Golden Hammer**, «золотой молоток». Это ситуация, при которой все проблемы пытаются решить одним инструментом. Например, вы любите C++ и вам кажется хорошей идеей разрабатывать на C++ веб-сервисы. Или вы любите Python и вам кажется хорошей идеей разрабатывать на Python настольные приложения с богатым пользовательским интерфейсом.

Собственно, антипаттерн — это не владеть выбранным технологическим стеком (это всегда хорошо), а нежелание (или даже активное сопротивление) посмотреть по сторонам. Пересесть на новый язык программирования и стек технологий у опытного программиста занимает не больше недели, ещё через пару месяцев он станет вполне продуктивным, а через полгода будет владеть технологией в совершенстве.

Важная проблема золотого молотка (почему он, собственно, антипаттерн, помимо потери эффективности разработки) — это то, что используемые технологии сильно влияют на создаваемое решение даже в плане требований. Например, если поддержка вебсервисов не сильная сторона C++, вы будете пытаться навязать клиенту приложение на голых сокетах, даже если ему нужно интегрироваться с другими системами по SOAP. Даже подсознательно все требования будут проходить через фильтр «а не слишком ли это сложно сделать на технологии X?», и в итоге клиент получит не тот продукт, что ему нужен, а тот, который не очень сложно запрограммировать на вашей любимой технологии.

Как бороться с «золотым молотком» — прежде всего, постараться убедить себя, что вы не Java-программист, C++-программист, Ассемблер-программист, а просто программист. Что если у вашей команды нет экспертизы в технологии X, это повод таковую экспертизу получить. Язык и технология — это инструменты решения задачи, а не определение вашей личности. Может помочь обучение. Например, внутрикорпоративные семинары, где опытные разработчики рассказывают другим командам про свои любимые технологии и оказывают поддержку коллегам в их внедрении. Ещё очень хорошо открывают глаза на вещи поездки на конференции (технические или даже научные), митапы.

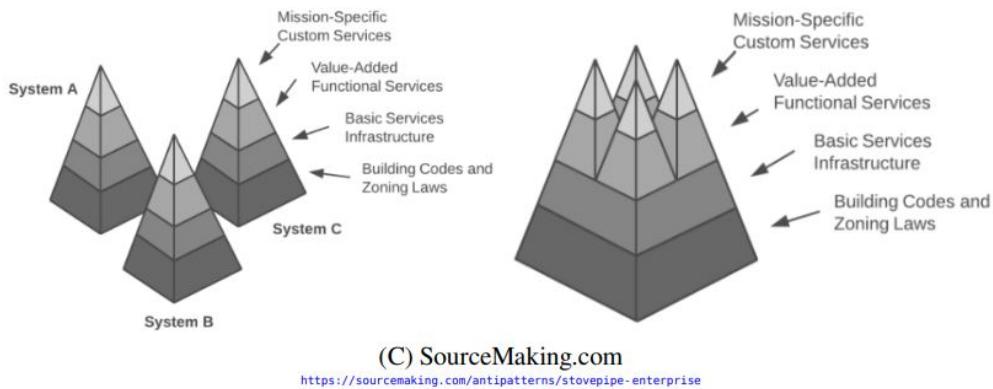
Архитектура тоже может помочь в борьбе с «золотым молотком». Если проектировать систему как набор слабо связанных заменяемых компонентов, можно добиться того, чтобы каждый компонент можно было разрабатывать на своём технологическом стеке.

И главное — не надо бояться новых технологий, новых языков и т.д.

## 54. Антипаттерны «Остров автоматизации», «Stovepipe system».

Антипаттерны архитектуры — это антипаттерны ещё более стратегические, чем рассмотренные ранее. Они зачастую затрагивают решения на уровне архитектур приложений целой организации и больше ошибки менеджмента, чем ошибки технических специалистов.

**Stovepipe Enterprise (Island of Automation), «Организация в стиле печной трубы», «Остров автоматизации»** — ситуация, когда в организации есть несколько проектов, которые нисколько не помогают друг другу. Такое в индустриальных компаниях встречается довольно часто, иногда из-за кажущейся невозможности организовать переиспользование кода на разных языках и технологиях, иногда по кажущимся соображениям авторского права. В любом случае, идеи, знания, типовые архитектуры переиспользовать можно, и хорошие организации к этому всячески стремятся. «Печная труба» тут символизирует старую печную трубу, которую часто приходится латать, и её латают чем попало, превращая в хаотичное нагромождение заплаток. Суть дела поясняет картинка:



Как бороться с такой ситуацией:

- использовать существующие промышленные стандарты.
- Фиксировать технологии и типовую архитектуру на уровне предприятия.
- Вообще, создать и поддерживать инфраструктуру переиспользования.

**Stovepipe System, «Система в стиле печной трубы»** — уточнение Stovepipe Enterprise для отдельной системы, состоящей из компонентов. Характеризуется отсутствием единого стандарта по взаимодействию между подсистемами и интеграцией подсистем по принципу «как получится» («ad-hoc»). В этом случае отработанные решения по взаимодействию между компонентами опять-таки не могут помочь друг другу, добавление нового компонента или изменение связей между существующими — большая работа, а тестирование такой системы — то ещё удовольствие. Как с этим бороться — как обычно, введением внутренних стандартов. Тут могут помочь и уже существующие стандарты, и использование известных паттернов интеграции — например, Enterprise Service Bus. Более «тактическое» решение — выделение единого интерфейса для подсистем и реализация всего взаимодействия в терминах этого интерфейса (так обычно устроены pluginные системы, например).

## 55. Антипаттерны «Привязка к поставщику», «Подразумеваемая архитектура», «Проектирование комитетом».

**Антипаттерн Vendor Lock-In, или «зависимость от поставщика»** — ситуация, когда архитектура вашей системы или её реализация зависит от третьестороннего коммерческого решения, которое вы не можете контролировать. Казалось бы, почему бы не разрабатывать приложение по поддержке электронного обучения HwProj на Ruby, молодой и перспективной технологии? Потому что HwProj используется через семь лет после своего запуска, а Ruby нет — специалистов, способных поддерживать HwProj, не найти, баги не фиксируются, новые фичи не реализованы. Это весьма типичная ситуация для приложений, у многих из них ужасно долгий жизненный цикл, длиющийся десятки лет.

Собственно, долгий жизненный цикл приложений — не антипаттерн, а суровая реальность. Антипаттерн — завязываться без нужды на вендора, особенно если нет оснований для уверенности, что он будет поддерживать свою технологию на протяжении ещё хотя бы пары десятилетий.

**Architecture By Implication, или «неявная архитектура»** — это антипаттерн архитектуры, отрицающий необходимость явной спецификации архитектуры в случае, если уже есть опыт создания подобных приложений.

Казалось бы, это логично, если опыт есть, есть и типовая архитектура с предыдущего проекта, и технологии, и даже наработки, зачем ещё раз делать одно и то же? Затем, что требования всё-таки могут различаться, и задачи, кажущиеся похожими, могут содержать в себе скрытые риски, которые при отсутствии явной разработки архитектуры могут остаться незамеченными.

Обратите внимание, переиспользовать архитектурные решения с предыдущих проектов никто не запрещает, причём выработка типовых архитектур и локальной библиотеки паттернов даже очень приветствуется, антипаттерн тут — отсутствие явной архитектурной документации вообще.

Как с этим бороться — требовать явного описания архитектуры системы всегда.

**Design By Committee, «проектирование комитетом»** — попытка принимать архитектурные решения простым большинством голосов («A camel is a horse designed by a committee»). Обсуждать архитектуру большими и шумными компаниями хорошо и правильно, но решения должны либо приниматься единогласно, либо должен быть один человек, который их принимает и отвечает за них. Иначе попытка включить в дизайн идеи, соображения и индивидуальные предпочтения каждого приводит к сложной, объёмной и внутренне противоречивой архитектуре.

Что можно сделать, чтобы избежать этой ситуации — во-первых, ограничить размер команды. Идеал — 4 человека, почти идеал — 6-7 человек. У Брукса в книжке «Мифический человеко-месяц» была интересная идея, что в команде собственно программирует систему только один человек, остальные занимаются вспомогательными задачами. Такая модель не прижилась, но в любой команде всё равно можно добиться разделения ролей, и надо его строго придерживаться. Должен быть явный главный архитектор, product owner, разработчики, эксперты и т.д. Это всё может быть одно лицо или целые группы, но роли должны быть чётко определены и лицо, принимающее решения, должно быть одно.

## 56. Понятие архитектурного стиля, трёхзвенная архитектура.

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

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

Есть ещё **архитектурные шаблоны** — это именованный набор ключевых проектных решений по эффективной организации подсистем, применимых для повторяемых технических задач проектирования в различных контекстах и предметных областях.

Архитектурные стили применимы уже не для всех проектов вообще, а в предпочтительных для каждого стиля предметных областях — одни стили хорошо работают во встроенных системах, другие — сетевых приложениях, третьи — в информационных системах. И решения, диктуемые стилями, применяются.

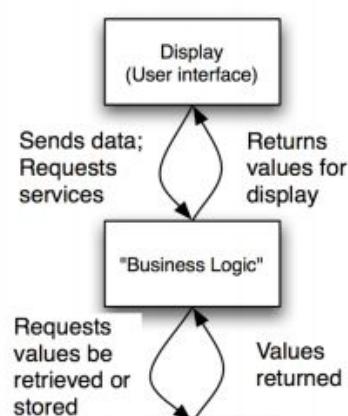
Архитектурные стили используются, чтобы получить следующие преимущества. •  
Переиспользование архитектуры. Создание архитектуры может быть сложной задачей, особенно когда у вас есть только требования и «чистый лист». Выбор стиля сужает пространство решений и направляет архитектурную мысль. При этом для новых задач можно применять хорошо известные и изученные решения, обладающие известными достоинствами и недостатками применительно к вашей ситуации.

- Переиспользование кода. Часто у архитектурных стилей бывают неизменяемые части, которые можно один раз реализовать, а затем переиспользовать в каждой системе. Это существенно сокращает затраты на разработку.
- Упрощение общения и понимания системы. Как и в случае с паттернами, достаточно просто назвать стиль по имени и не надо объяснять, как что устроено — опытные разработчики вас сразу поймут.
- Упрощение интеграции приложений, на тактическом уровне за счёт переиспользования стандартов и middleware, типичных для стиля, на стратегическом — за счёт того, что понятно, куда и как встраиваться, не нарушив архитектурные ограничения каждого из приложений.
- Применение специфичных для стиля методов анализа. Поскольку стили накладывают ограничения на структуру систем, иногда эти ограничения достаточно жёсткие, чтобы про систему можно было что-то доказать или что-то посчитать.
- Специфичные для стиля методы визуализации — тот же Pipes and Filters удобно рисовать с помощью визуальных языков, ориентированных на данные, таких б как Data Flow Diagram. Или становится возможным использование предметноориентированных языков.

Стили определяются тремя основными соображениями: набор используемых в стиле элементов, набор правил, по которым эти элементы соединяются, и семантика, стоящая за элементами.

Архитектурные шаблоны — это более специализированная вещь, чем стили, и несколько более «тактическая» (хотя и не настолько, как паттерны). Архитектурные шаблоны диктуют типовые решения для типовых задач, например.

**State-Logic-Display**, также известный как «трёхзвенная архитектура», часто рассматривается как архитектурный стиль, часто — как архитектурный шаблон, так что вообще разделение на архитектурные шаблоны и архитектурные стили весьма условно. Если трёхвенка — это только способ решения одной из задач, то это

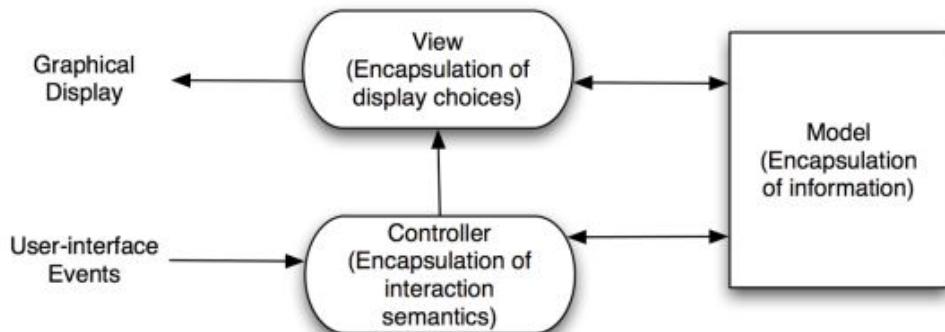


архитектурный шаблон. Вообще, трёхзвенка предполагает разделение приложения на часть, отвечающую за представление и взаимодействие с пользователем (и ничего больше), бизнес-логику и слой хранения данных.

Наличие чёткого разделения ответственности между слоями делает каждый из них управляемым, а функциональность — переиспользуемой. Например, пользовательский интерфейс легко поменять, сделав вместо десктопного приложения мобильное, не меняя бизнес-логики.

Главные архитектурные ограничения — клиент не знает ничего о БД и не может с ней напрямую взаимодействовать, бизнес-логика сама не пытается хранить информацию и не содержит ничего, что касается взаимодействия с пользователем, база данных не пытается делать ничего нетривиального с данными. Приложения, устроенные таким образом — это подавляющее большинство вебприложений и информационных систем (которые чаще всего сами веб-приложения), многопользовательские игры. В них клиент может быть очень продвинутым и очень сложным, но он не вправе заниматься реализацией игровой логики и не может хранить данные, кроме тех, что относятся к взаимодействию с пользователем (например, его логин).

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



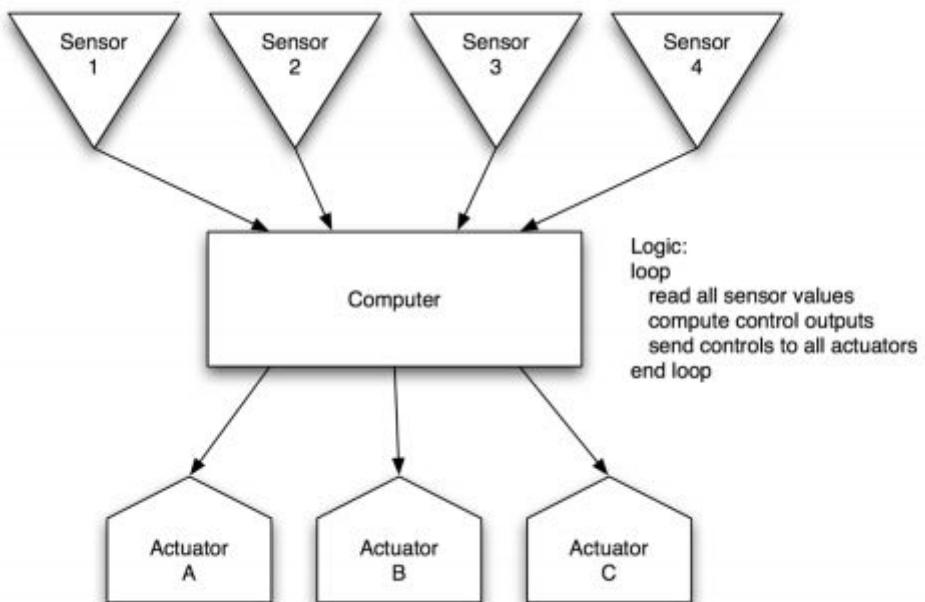
**Model-View-Controller** — это архитектурный шаблон, который иногда классифицируют как паттерн проектирования.

View отвечает за отображение данных пользователю и только за это. Пользовательский ввод поступает в Controller, ответственность которого — обеспечить логику взаимодействия с пользователем и при необходимости управлять для этого View. Model — компонент, хранящий в себе все данные и, как правило, включающий в себя также и бизнеслогику приложения. Контроллер обрабатывает пользовательский ввод, сообщает о требуемых действиях модели, модель их выполняет, меняет данные и рассыпает нотификацию о том, что в ней что-то изменилось. Эту нотификацию получает представление, читает информацию из модели и обновляет себя.

Архитектурные ограничения: представление может только читать из модели, команды модели может отдавать только контроллер. Сигнал об обновлениях модель

рассыпает сама, что позволяет иметь несколько разных представлений, отображающих одну модель, которые будут синхронно обновляться. Контроллер — единственное место системы, через которое проходят все команды пользователя. Чем это хорошо — так же, как в трёхзвенке, имеется чёткое разделение ответственности между компонентами и возможность выкинуть представление (или контроллер) и заменить на новое.

Model-View-Controller отличается от трёхвенки тем, что не специфицирует, кому где хранить данные (поэтому применим в приложениях, которым данные особо хранить не надо), но требует наличия выделенного элемента, отвечающего за обработку пользовательского ввода (в трёхвенке это делал Display, вместе с выводом информации). Типичные приложения, построенные по такому принципу — это большинство десктопных приложений с развитым пользовательским интерфейсом.



**Sense-Compute-Control** — архитектурный шаблон, применяющийся прежде всего в робототехнике и схожих областях (например, «умных домах»). Он предполагает разделение работы системы на три фазы — снятие показания с датчиков, вычисление управляющего воздействия, посылка управляющего воздействия на актуаторы.

При этом вся система работает в бесконечном цикле, попеременно выполняя эти три фазы.

Хорошо это тем, что чётко определяет архитектуру системы и для простых систем фактически решает все архитектурные вопросы. Для сложных систем это может быть только самый внешний каркас процесса работы, но поэтому это и не архитектурный стиль, а всего лишь шаблон.

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

Главная программа/подпрограммы (**наверно Структурный, в конспекте не написано**) — стиль, опирающийся на функциональную декомпозицию. Задача разбивается на подзадачи, каждая из которых решается отдельной функцией, которая в свою очередь может вызывать другие функции и т.д., пока задачи не станут достаточно простыми. Стиль хорош простотой и предсказуемостью поведения системы, скоростью работы. Плох тем, что для любой нетривиальной задачи функций (и даже уровней декомпозиции) получится существенно больше, чем можно удержать в голове, и между ними потребуется передавать данные (опять-таки, в очень больших количествах), которым очень сложно обеспечить инкапсуляцию. В общем, такой подход реально хорошо подходит для небольших систем, но если получается больше нескольких тысяч строк кода, всё становится плохо управляемым. Формально компонентами в таком стиле являются функции, соединителями — вызовы, с передачей параметров, ограничениями — обычно граф вызовов должен образовывать ориентированный ациклический граф, но при необходимости колбэков и взаимной рекурсии это ограничение может нарушаться (что несколько усложняет анализ программы).

«Сырой» **объектно-ориентированный стиль** — когда программа представляется в виде набора взаимодействующих объектов в обычном объектно-ориентированном стиле. Компонентами в этом стиле выступают объекты, соединителями — вызовы методов, ограничения — объекты должны полностью контролировать своё внутреннее состояние и менять его можно только через вызовы методов, при этом реализация этих методов должна быть полностью скрыта от других объектов.

Достоинства — прежде всего, близость к предметной области и «обычному» человеческому мышлению. Сущности предметной области и так имеют своё состояние и поведение, это более-менее прямолинейно ложится в код. Кроме того, с точки зрения техники программирования каждый объект независим, так что его можно разрабатывать отдельно, и главное, думать о нём отдельно: рассматривать систему как набор взаимодействующих агентов, которые вместе решают задачу, но каждый из них более-менее обособлен, его можно отдельно разрабатывать и отдельно тестировать. Ещё одно важное тактическое соображение — реализацию объектов можно смело менять, пока выполняются их инварианты, во внешнем мире это гарантированно ничего не сломает.

Есть и недостатки. Главный — это, пожалуй, отсутствие строгих топологических ограничений на связи между объектами, что приводит к тенденции «чистых» объектноориентированных программ превращаться в месиво из объектов, где каждый вынужден знать о каждом. Второй недостаток (по сравнению с функциональным программированием, по крайней мере, и даже со структурным) — склонность к побочным эффектам. Методы обычно

меняют состояние объекта, так что разная последовательность вызовов одних и тех же методов с одними и теми же параметрами приводит к разным результатам. Это существенно усложняет анализ.

**Слоистый стиль** и его возможные вариации. Суть этого стиля в том, что мы разделяем систему на слои, где каждый слой может пользоваться слоями ниже и предоставляет интерфейс для слоёв выше, при этом сам ничего о них не зная.

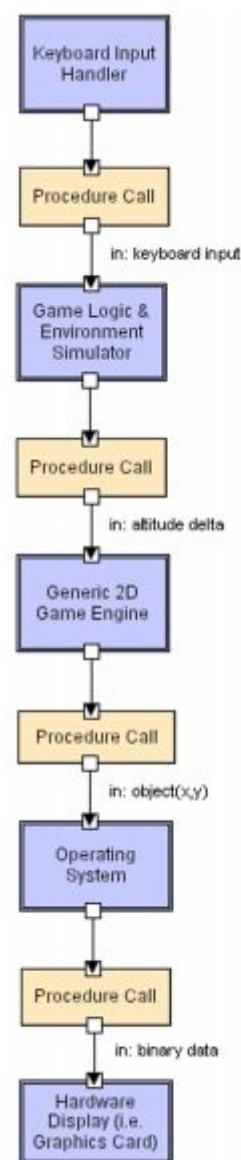
Компонентами в таком стиле выступают сами слои, они могут быть сколь угодно сложно устроены внутри, но это их детали реализации. Соединителями — протоколы общения слоёв (или просто программные интерфейсы).

Преимущества слоистого стиля — это в первую очередь постепенное повышение уровня абстракции от низких уровней к высоким. В строгом варианте, когда слой может общаться только со слоем непосредственно ниже, это позволяет вообще не думать о реализации всей системы, а просто программировать в терминах предоставляемой слоем абстракции.

Ещё слоистость существенно облегчает сопровождение системы. Поскольку каждый уровень влияет только на уровни выше, влияние каждого изменения легко оценить и отследить. При этом можно использовать разные реализации каждого уровня, лишь бы они удовлетворяли общему интерфейсу.

Однако есть и проблемы. Во-первых, уровневый стиль оказывается не всегда применим — взаимодействие между элементами системы может быть таким, что не позволяет себя упорядочить по уровням.

Во-вторых, проблемой может стать производительность системы. Сложные уровневые архитектуры имеют тенденцию обрастиать функциями, которые просто прокидывают запрос на уровень ниже, что ведёт к ненужному оверхеду на вызовы.

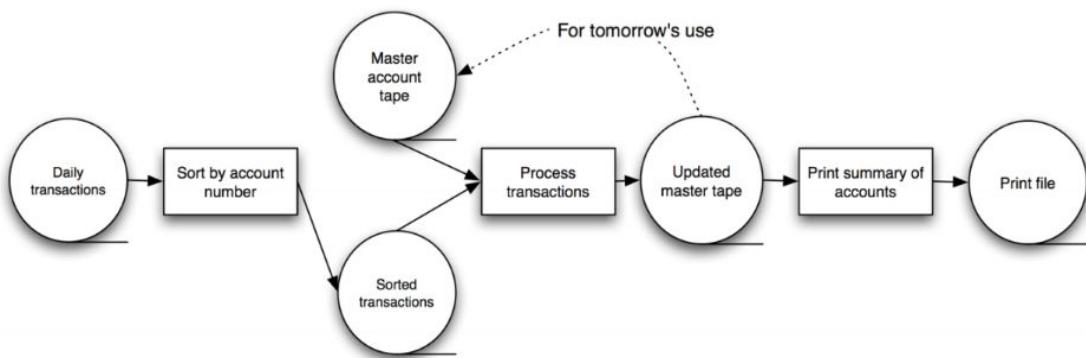


Если производительность критична, уровневый стиль может всё-таки хорошо подойти, но может и нет (особенно, если перестараться с количеством уровней).

Тем не менее, уровневость — это хорошо, поэтому иногда стоит даже проделать дополнительную работу, чтобы разделить систему на уровни. Уровневая архитектура считается довольно стандартной и, например, профстанд

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

### Пакетная обработка.



Итого, система в стиле «пакетная обработка» строится как набор отдельных программ, которые выполняются последовательно, обмениваясь данными средствами операционной системы. Это давно уже не ленты, а файлы, либо pipes или named pipes<sup>2</sup>. При такой схеме между процессами требуется передавать в явном виде всё, что необходимо им для работы — сами данные, конфигурацию, управляющие команды.

Стиль хоть и древний, но очень популярен до сих пор. По сути, Linux way с большим количеством маленьких утилит, из которых с помощью пайпов выстраиваются конвейеры — это и есть пакетная обработка. Так что этот стиль применяется практически во всех скриптах под Linux и без него трудно представить работу системных администраторов, DevOps и т.д.

Важное преимущество этого стиля — независимость отдельных программ. Они могут быть написаны на каком угодно языке и какой угодно технологии, лишь бы умели читать из входного потока. Можно их вообще не писать, а переиспользовать уже готовые в любой части конвейера. Этакая микросервисная архитектура внутри одной машины.

Важный недостаток этого стиля — неторопливость работы. Каждая программа — отдельный процесс, даже просто их запуск трудоёмок по времени и памяти, да и

коммуникации между процессами, хоть и не так тяжелы, как коммуникации по сети, всё-таки гораздо медленнее общения потоков внутри процесса.

**Каналы и фильтры (или «pipes and filters»)** — стиль, в котором программа представляется в виде набора фильтров, которые как-то преобразуют данные, идущие по каналам. При этом фильтры независимы друг от друга, то есть не имеют разделяемого состояния и ничего не знают про фильтры до и после них. Всё, что они видят — это данные в своих входных каналах. Собственно, фильтры являются единственным типом элементов в такой архитектуре, а каналы — единственным типом соединителей.

«Каналы и фильтры» похожи на «пакетную обработку», но не требуют, чтобы каждый фильтр был отдельной программой, что помогает победить проблемы с производительностью в пакетной обработке. Кроме того, каналы и фильтры имеют тенденцию образовывать сложные сети, в отличие от пакетной обработки, где всё в основном линейно.

Бывают варианты каналов и фильтров:

- конвейеры — где фильтры связаны просто в линейную цепочку, очень топологически простой стиль, подходящий для несложной логики обработки (хотя сами фильтры могут быть сколь угодно сложны);
- ограниченные каналы — где канал представляет собой очередь с ограниченным количеством элементов, блокирующую фильтр-источник, если очередь переполнена.
- Типизированные каналы — где каналы знают тип передаваемых данных, и фильтры могут подключаться только к каналам правильного типа.

Преимущества этого стиля таковы.

- Поведение системы — это просто последовательное применение поведений компонентов. Так что о нём легко рассуждать, его легко понять, такие системы легко поддерживать.
- Легко добавлять, заменять и переиспользовать фильтры. Специально продумывать интеграцию компонентов не нужно, она получается сама собой.
- Широкие возможности для анализа. Поскольку есть чёткие ограничения на потоки данных, систему можно рассматривать просто как граф из фильтров с рёбрами каналами, что делает применимыми все алгоритмы анализа графов.
- Широкие возможности для параллелизма. Каждый фильтр может работать одновременно со всеми остальными, либо в отдельном потоке, либо в отдельном процессе на другой машине (что, кстати, делает фильтры естественными кандидатами в микросервисы).

Недостатки тоже есть:

- Последовательное исполнение — что странно противоречит достоинству про параллелизм. Но пока первые фильтры из сети не сделают своё дело, следующие за ними к работе приступить не могут. Это не важно, когда данных много и вся сеть занята их обработкой, но если данные поступают лишь иногда, они должны последовательно пройти через все фильтры, при этом большая часть фильтров будет простаивать.
- Проблемы с интерактивными приложениями, поскольку данные идут по фильтрам в одном направлении и непонятно, как ими управлять.
- Пропускная способность всей системы определяется самым “узким” элементом.

**Blackboard** — это архитектурный стиль с общей памятью. Есть центральное хранилище данных, тот самый «Blackboard», есть компоненты, которые знают только про Blackboard, не имеют своего состояния и, собственно, выполняют полезную работу. Процесс работы системы устроен так, что каждый компонент смотрит на Blackboard, (возможно) находит там данные, которые может преобразовать, преобразовывает их и записывает обратно на Blackboard, в надежде, что какой-то другой компонент сможет преобразовать их дальше.

Весь процесс вычислений управляет только через Blackboard, поэтому если мы хотим, чтобы некоторые действия выполнялись последовательно, нам надо хранить на Blackboard какой-то токен исполнения и обновлять его при выполнении действий.

Такой подход хорош тем, что компоненты могут быть полностью независимы и работать параллельно (и разрабатываться независимо, кстати), система легко масштабируется и расширяется добавлением новых компонентов, и главное, что вам не надо даже понимать алгоритм решения задачи — вы просто бросаете в систему компоненты пока задача не решится. Это же и главный недостаток этого стиля — задача легко может и не решиться. Поэтому такой стиль применяется прежде всего в системах искусственного интеллекта, где задача и так вполне может быть нерешаемой. Ещё, кстати, Blackboard, как правило, является узким местом системы, поскольку ему надо синхронизировать доступ независимых компонентов к себе.

## 60. Событийно-ориентированные стили, Publish-Subscribe.

Стили с неявным вызовом — общее название разных вариантов событийноориентированных стилей или вообще стилей, где хотя бы иногда используются оповещения вместо явных вызовов методов. Во всех таких стилях есть «слушатели», которые могут подписываться на события, и при наступлении события система сама вызывает всех зарегистрированных слушателей.

В таких архитектурах компоненты имеют два вида интерфейсов — обычный набор методов, и события, на которые можно подписываться. В качестве соединителей используются либо прямые вызовы методов, либо неявные вызовы слушателей по наступлению события (почему стили и называются стилями с неявным вызовом).

Инварианты всех таких стилей:

- те, кто производит события, не знают, кто и как на них отреагирует — они, как правило, технически имеют список подписавшихся, но обычно не вправе даже узнать их количество, не говоря уж о том, чтобы что-то делать с подписавшимися напрямую;
- не делается никаких предположений о том, как событие будет обработано и будет ли вообще — источник просто нотифицирует систему о наступлении события, а уж подписан на него кто-нибудь или нет, в каком порядке кто подписан и т.д. — не его дело.

Преимущества всех таких стилей — это переиспользуемость компонентов и лёгкость конфигурирования системы. Высокая переиспользуемость достигается за счёт очень низкой связности между компонентами, ведь источник событий вправе вообще ничего не знать о тех, кто им пользуется. Лёгкость конфигурирования, как во время компиляции, так и во время выполнения, достигается за счёт того, что подписки на события можно легко менять, меняя при этом всю функциональность системы.

Недостатков, тем не менее, тоже довольно много.

- Зачастую неинтуитивная структура системы. Без применения дополнительных архитектурных ограничений подписки на события превращаются в хаотичный клубок, в котором хаотично распространяются нотификации.
- Компоненты не управляют последовательностью вычислений. Работа системы состоит в генерации событий и реакций на события, и делать что-то в правильном порядке в сколько-нибудь сложной системе может оказаться проблематичным.
- Непонятно, кто отреагирует на запрос и в каком порядке придут ответы. Компонент, генерирующий события, не вправе предполагать, что на событие кто-то отреагирует, поэтому если это событие, например, «мне нужны данные для дальнейшей работы», мы не вправе рассчитывать на ответ. А если надо запросить несколько разных источников, то неизвестно, кто и когда ответит. Поэтому такие системы принципиально асинхронны.
- Тяжело отлаживаться.
- Ситуации, очень похожие на гонки, даже если у вас всего один поток. Классическая гонка — это когда результат работы программы зависит от случайного порядка переключения потоков планировщиком. Гонка в событийных системах — это когда результат работы программы зависит от случайного порядка вызова обработчиков при нотификации. Обычно событийные системы хоть и не позволяют закладываться на определённый порядок вызова обработчиков, всё же вызывают их в порядке подписывания, что делает процесс хоть сколько-нибудь детерминированным.

«Издатель-подписчик» — самый простой стиль из стилей с неявным вызовом, и вместе с тем весьма популярный и эффективный. Есть издатели, они публикуют сообщения (синхронно или асинхронно, то есть дожидаясь их обработки либо нет). Есть подписчики, которые подписываются на издателей и получают от них события. Есть маршрутизаторы, задача которых — быть посредниками между издателями и подписчиками, фильтровать и маршрутизировать сообщения. При этом в самых простых конфигурациях без маршрутизаторов прекрасно обходятся, но они могут быть полезны: например, маршрутизатор может по очереди отправлять сообщения то одному подписчику, то другому, реализуя тем самым балансировку нагрузки. Компонентами в таком архитектурном стиле являются издатели, подписчики и маршрутизаторы, соединителями — сетевые протоколы или очереди сообщений, либо, если дело происходит на одной машине, механизмы наподобие паттерна «Наблюдатель». В качестве данных в этом стиле выступают подписки, нотификации о произошедших событиях, публикуемая издателями информация. Ограничения — издатели ничего не знают о подписчиках, подписчики, как правило, ничего не знают друг о друге, только об издателях.

Преимущества этого стиля, как и обычно для событийно-ориентированных схем — очень низкая связность между компонентами, но при этом высокая эффективность

распространения информации, отчасти за счёт свойственной этому стилю простоты топологии, отчасти за счёт наличия маршрутизаторов.

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

**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).

Модель предметной области выражается прежде всего в коде — сущности предметной области становятся классами языка программирования, используемого для реализации. Также для обсуждения предметной области с экспертами и для документирования модели часто используются диаграммы. Немаловажную роль как в создании модели, так и в её “фиксации” и улучшении играет ещё и устное общение. Неудачные названия классов или методов, неуклюжие и непонятные описания взаимодействий прекрасно слышны в разговоре и являются поводом для того, чтобы посмотреть на модель ещё раз. Сам естественный язык часто подсказывает правильные архитектурные решения — что неудивительно, язык сотни лет оттачивался как раз для того, для чего он нужен в DDD — для передачи сути понятий.

Модель определяет единый язык, на котором должны общаться все члены команды и эксперты предметной области, которые им помогают. Если какой-то термин зафиксирован как имя класса, использование его синонимов ни в диалогах, ни в документации, ни тем более в коде не допускаются, можно использовать только имя класса.

Модель появляется не только благодаря применению единого языка и последовательного именования всех нужных сущностей, она также “выкристаллизовывается” в процессе непрерывного рефакторинга и уточнения.

**Единый язык** — одно из ключевых понятий предметно-ориентированного проектирования. Необходимость его введения связана с тем, что программисты и специалисты предметной области разговаривают на разных профессиональных жаргонах и изначально не понимают друг друга.

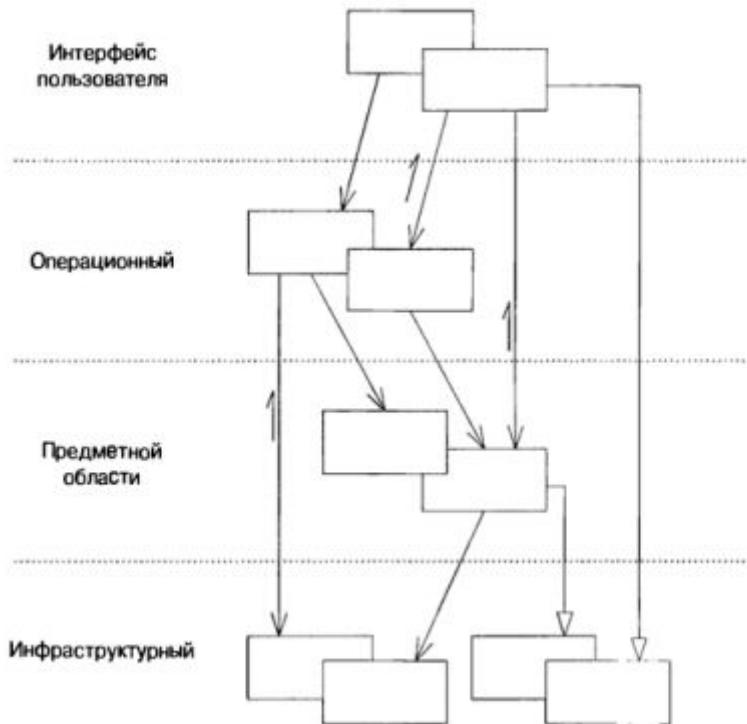
Поэтому предметно-ориентированное проектирование предписывает выработку единого языка и использование его повсюду в проекте, от составления технического задания до имён методов, переменных и тестов. В единый язык входят термины предметной области — как правило, они становятся именами классов, отношения между понятиями и действия, которые сущности могут выполнять — это имена

методов, также в язык попадают имена паттернов, используемых при проектировании системы, даже если их в предметной области нет (например, “репозиторий” или “спецификация”). Ещё единый язык должен содержать 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).

## Изоляция предметной области

Хорошо, допустим, мы смогли построить модель предметной области. Но этого недостаточно, чтобы получить работающую программу — нужен ещё пользовательский интерфейс, сеть, база данных, всякого рода вспомогательный код, логирование, обработка ошибок, юнит-тесты и т.д. Основная идея DDD в том, что всё это должно быть архитектурно отделено от модели предметной области, чтобы код модели — код, в котором сосредоточена вся суть приложения — мог быть максимально простым, небольшим по размеру и содержащим только существенные для смысла системы вещи. Всё остальное должно находиться в отдельных модулях и просто использовать классы доменной модели.

Самый популярный способ достижения такого разделения (хотя и не единственный) — уровневая архитектура. Программа реализуется в виде набора уровней, где каждый уровень может непосредственно взаимодействовать только с уровнями ниже. Способов разделения на уровни бывает много (трёхзвенная 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.



Уровень интерфейса отвечает только за взаимодействие с пользователем и реализует только отображение и обработку событий. Никакой содержательной логики в нём быть не должно. Операционный уровень (application layer) занимается координацией действий бизнес-объектов, которые находятся на уровне предметной области. Операционный уровень тоже должен быть очень простым, его ответственность — это инициализировать систему и по сигналу от интерфейса инициировать нужные пользователю операции. Инфраструктурный уровень содержит все вспомогательные вещи, неспецифичные для данной предметной области, например, код работы с базой данных или код работы с сетью.

Уровень предметной области содержит бизнес-объекты, которые ничего интересного кроме, собственно, реализации бизнес-правил, не делают. Их реализация должна быть максимально простой, и именно их реализации следует уделять максимум внимания, потому как именно уровень предметной области определяет конкурентные преимущества и полезность программы.

Операционный уровень специфичен для каждого конкретного приложения, тогда как уровень предметной области может разделяться между несколькими приложениями в семействе. На операционном уровне из классов уровня предметной области собирается то, что делает конкретно наше приложение, операционный уровень же координирует бизнес-объекты. Но бизнес-регламенты (например, последовательность действий в бизнеспроцессе) должны быть реализованы на уровне предметной области, потому как они хоть и координируют действия других объектов, но объективно присутствуют в предметной области и могут быть переиспользованы в других приложениях. Операционному уровню также запрещается иметь состояние (кроме, быть может, состояния, необходимого для общения с пользователем приложения, типа прогресса операций).

Инфраструктурный уровень поддерживает все уровни выше. Он может обеспечивать архитектурную среду для всех остальных уровней (например, Java Beans или ROS), может содержать специфичный для приложения код работы с третьесторонними библиотеками и технологиями (например, Object-Relational Mapping). Может показаться противоестественным, что базовые классы для уровней выше могут быть на инфраструктурном уровне (то есть деревья наследования растут вверх), но если подумать, кто о ком больше знает — предок или потомок, то становится понятно, что концептуально всё ок.

Уровням ниже, естественно, иногда требуется общаться с уровнями выше. Даже инфраструктурный уровень, если он не состоит только из библиотек, может инициировать действия на уровнях выше (например, получение сетевого пакета вызывает обработку на операционном уровне). Поскольку общаться напрямую уровни не могут, применяются приёмы “косвенного” вызова — callback-и (или виртуальные методы — hook-и), паттерн “Наблюдатель”, “дедушка всех паттернов” MVC. Такой подход несколько затрудняет отладку, но позволяет разрабатывать каждый уровень независимо, и не думать, как будут пользоваться кодом уровня другие.

## 62. DDD, основные структурные элементы модели предметной области.

В программе модель реализуется с помощью различных “элементарных” блоков, и не всегда тривиально. В первую очередь, следует подумать над ассоциациями. Если в предметной области одно из направлений ассоциации более приоритетно, чем другое (например, “страна – президент”, скорее всего, предполагает доступ от страны к президенту), то имеет смысл сделать ассоциацию односторонней. В идеале все или почти все ассоциации должны быть односторонними, потому что двунаправленная ассоциация делает невозможным понять один объект в отрыве от другого. Также следует по возможности минимизировать множественность (возможно, добавив квалификаторы, типа “страна – год – президент”) и минимизировать количество ассоциаций вообще. Меньше связей — проще анализ и сопровождение.

Следующая задача — решить, кто в модели будет сущностью, а кто — объектомзначением. Сущность имеет собственную идентичность, не зависящую от её состояния, тогда как объект-значение определяется только значениями своих атрибутов.

Часто в качестве идентификатора используют некий суррогатный атрибут, например, GUID, назначаемый объекту при создании. Иногда этот атрибут даже становится доступен пользователю (например, трек-номер почтового отправления), иногда нет (например, у каждого документа в Google Docs есть уникальный Id, который нигде в интерфейсе не показан, поэтому в одной папке вполне может быть два документа с одним именем).

От способа идентификации требуется, чтобы он переживал сохранение, загрузку, передачу в другую систему и представление там в другом формате. Соответственно, нужна операция идентификации, которая по двум данным объектам может сказать, один и тот же объект или нет. Ссыльное равенство, очевидно, не подходит для большинства практически полезных случаев, поэтому способ идентификации на самом деле важное архитектурное решение.

В DDD разрешено и даже поощряется то, что считается дурным тоном в ООП — классы без состояния, фактически представляющие собой набор статических методов. В DDD их называют “службами”. В службы выносится код, который не имеет естественного места в “настоящих объектах” — часто это операция, выполняемая над объектами нескольких разных классов, где все объекты равноправны, поэтому нельзя естественным образом поместить эту операцию в один из классов-участников — например, различного рода диспетчеры или менеджеры. Службам, как правило, запрещается иметь состояние, так что вызывать методы службы можно отовсюду и в любом порядке, не заботясь о порядке вызовов.

Последней элементарной частью модели является модуль. Да, это модули в обычном для языков программирования смысле, DDD лишь призывает использовать их не как техническое средство группировки классов, а как реальный механизм

декомпозиции предметной области. Классы следует объединять в модули не по принципу высокой внутренней связности (cohesion) и низкой внешней зависимости (coupling), а семантически. Модуль — это важная часть модели, модули должны быть определены так, чтобы повышать “объясняющую способность” модели предметной области. А высокая связность и низкая зависимость при этом получатся сами собой. Если это не так, имеет смысл порефакторить модель. Имена модулей должны естественным образом вписываться в единый язык модели.

## 63. DDD, паттерн «Агрегат».

**У литвинова был доп:**

Прагматичная сторона полезности агрегата.

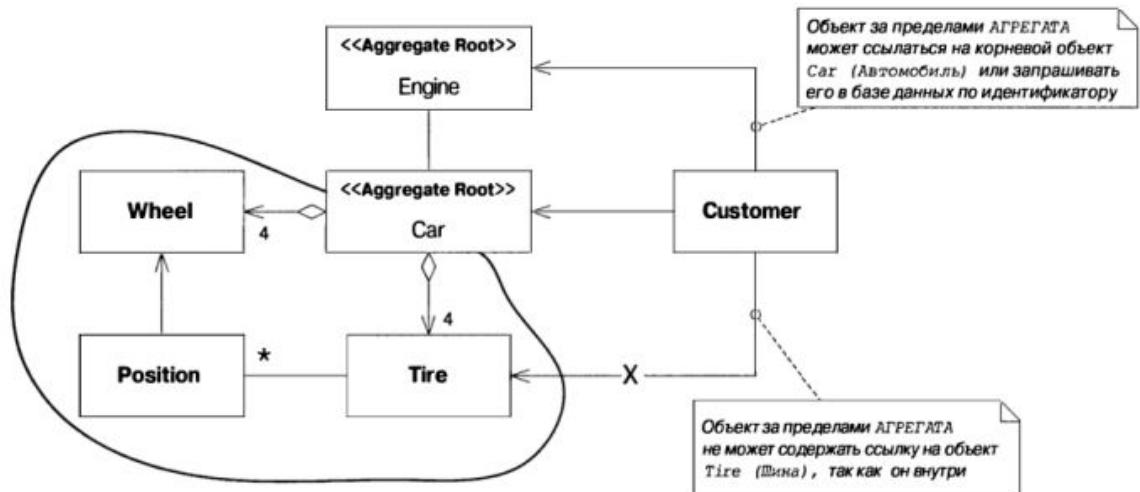
**Ответ:**

Агрегат позволяет изолировать друг от друга разные операции над базами данных, чтобы транзакции не откатывались из-за того, что кто-то успел уже что-то поменять. Если мы захватили корень агрегата это означает что никто больше нам данные не испортит.

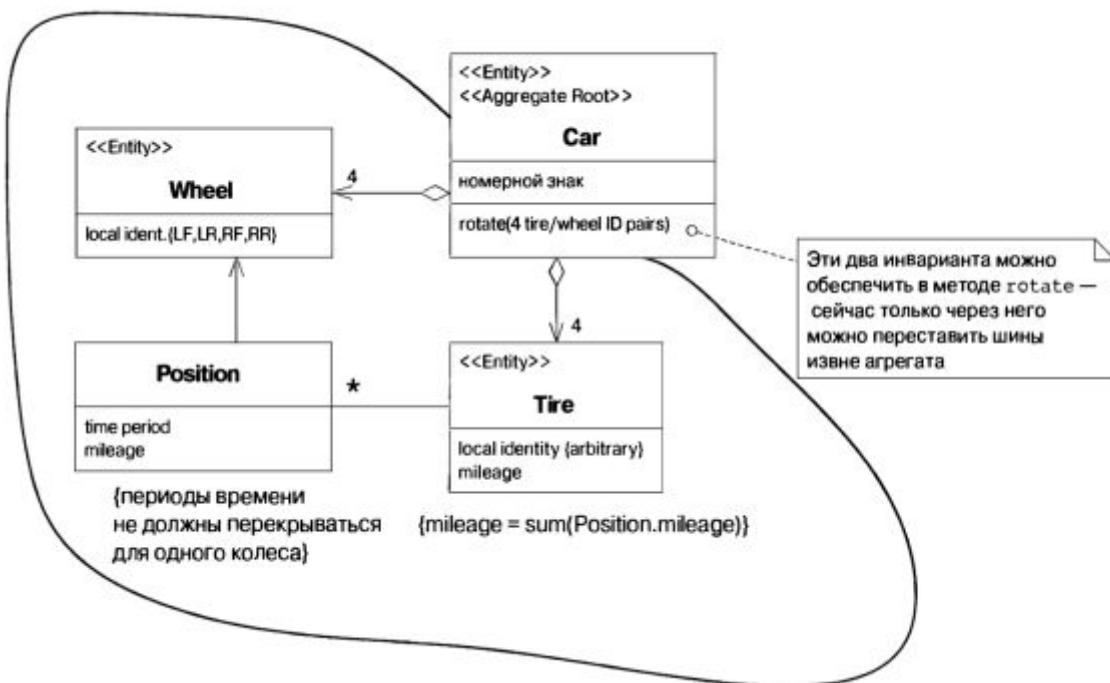
Тут, как обычно, возникают проблемы с идентичностью объекта, если это сущность, а не значение, но кроме этого, добавляются проблемы с целостностью объекта и с излишней сложностью управления его состоянием. Для решения этих проблем применяются некоторые широко известные паттерны проектирования.

Первый паттерн — паттерн “Агрегат”. Агрегат — это изолированный кусок модели, имеющий “корень” и “границу”. Доступ ко всем элементам агрегата возможен только через его корень. Корень представляет собой объект-сущность и имеет свою идентичность в рамках всего приложения. Остальные объекты в агрегате могут быть как сущностями, так и значениями, и они должны поддерживать свою идентичность только в рамках агрегата. Для чего это нужно — для упрощения и декомпозиции модели. Когда мы пишем агрегат, мы можем думать только о взаимосвязи объектов внутри агрегата, а когда мы пишем код, пользующийся агрегатом, мы можем думать только о его корне. Так что агрегат — это такая единица инкапсуляции программы. Как класс, только включает в себя, возможно, несколько классов. С технической точки зрения аккуратное разделение на агрегаты в разы облегчает обновления в базе данных: транзакции, затрагивающие разные агрегаты, гарантированно не мешают друг другу.

Содержимое агрегата на самом деле может быть видно снаружи, и не возбраняется отдавать ссылки на внутренние объекты агрегата, просто извне их нельзя хранить. Кроме того, у агрегата (как у класса) есть свои инварианты, и за поддержание их отвечает корень. Поэтому объекты, которые отдаются вовне, скорее всего, должны быть немутабельны, или быть копиями объектов внутри агрегата. Вот небольшой пример.



Тут машина — агрегат, содержащий в себе колёса и шины, поскольку пользователю может быть важно, на каком колесе какая шина, но глобально идентифицировать шины не требуется. Двигатель же наоборот, хоть и является неотъемлемой частью автомобиля, имеет собственный номер, так что уникален глобально (и на самом деле номер двигателя иногда проверяют безотносительно номера автомобиля). Автомобиль же предоставляет всю функциональность для управления шинами извне и поддерживает свои инварианты:



## 64. DDD, паттерны «Фабрика», «Репозиторий».

Следующий паттерн — “Фабрика”. В DDD это несколько более архитектурный паттерн, чем похожий паттерн “Абстрактная Фабрика” Банды Четырёх, и на самом деле может использовать абстрактную фабрику для реализации. Задача фабрики —

создать и проинициализировать агрегат (или даже просто один объект) так, чтобы выполнялись его инварианты. Фабрика инкапсулирует сложную операцию создания объекта, не раскрывая его внутреннюю структуру, и при этом избавляет сам агрегат от необходимости уметь создавать себя. Фабрика вправе знать о структуре агрегата и манипулировать его внутренним содержимым непосредственно, а код извне — нет, так что без фабрики пользоваться агрегатом было бы зачастую невозможно:



Клиент передаёт фабрике параметры, указывающие, что ему нужно, фабрика создаёт продукт и возвращает клиенту результат. Процесс создания атомарен в том смысле, что если что-то пошло не так, то клиент не получит результата вообще, как будто ничего не было.

Фабрики могут использоваться не только для создания, но и для восстановления объекта из внешнего хранилища. В этом случае они не должны присваивать объекту новый идентификатор или вообще как-либо портить его идентичность.

**Репозиторий** отвечает за хранение, сохранение и загрузку объектов при необходимости, предоставляя, как правило, глобальный доступ к объектам. Чаще всего репозиторий представляет собой прослойку между приложением и базой данных, при этом репозиторий сам может держать объекты в памяти и возвращать их по запросу. Нужно это для того, чтобы минимизировать количество переходов по ссылке с целью найти какой-либо объект, и вообще упростить задачу поиска.

Репозиторий может использовать развитый язык запросов, чтобы позволять клиенту найти объект. Часто это просто какой-то метод идентификации (типа суррогатного Id), но вполне может быть выборка объекта или коллекции объектов по их атрибутам или каким-то другим признакам (например, участии в отношениях с другими объектами).

Опять таки, может использоваться паттерн “Спецификация”, про который чуть позже. Репозиторий за интерфейсом запросов прячет метод реального получения объекта: внутри он может использовать ORM-библиотеку + реляционную БД, может объектно-ориентированную БД, может (и часто использует) фабрики для создания объектов или восстановления объектов из базы. Задача репозитория — делать так, чтобы обо всём этом не надо было заботиться клиентскому коду:



65. Говорящие интерфейсы, функции без побочных эффектов, assertions, замкнутые операции.

66. Ограниченный контекст, непрерывная интеграция, карта контекстов.

Центральной идеей методологии предметно-ориентированного проектирования является единая модель предметной области и единый язык, что хорошо и правильно, но если над проектом работает сотня человек, это проблематично.

Итак, предметно-ориентированное проектирование в случае проекта, над которым работают несколько команд, сталкивается с проблемой того, что команды неизбежно имеют разные видения продукта. Эту проблему можно решать,

- либо поддерживая модель интегрированной — но тогда затраты на поддержание её целостности будут слишком велики (в идеале — все команды должны будут непосредственно общаться со всеми остальными, и каждое изменение в модели утверждаться остальными командами), при этом модель наверняка получится слишком общей, чтобы быть полезной;
- либо приняв ситуацию как должное и разрешив модели быть фрагментированной — но тогда это неизбежно затруднит переиспользование кода в рамках проекта и интеграцию системы.

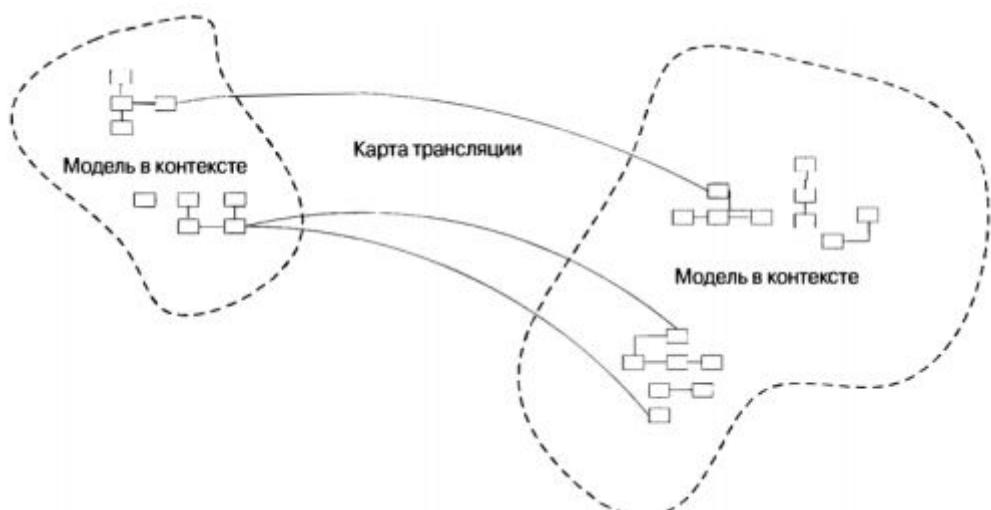
**Ограниченный контекст** (Bounded context) — это кусок предметной области (и, соответственно, реализующего её кода), в которой применима единая модель предметной области. Всё, что внутри ограниченного контекста, должно следовать этой единой модели, иметь единый язык и т.п., всё, что вне — не должно делать никаких предположений о модели и не имеет право её без спроса переиспользовать. То есть, ограниченный контекст — это что-то вроде атомарной области проекта, над которой работает одна команда (как клетка в живом организме, простите за банальную метафору). Разделение системы на ограниченные контексты обычно

следует организационной структуре проекта, которая, в свою очередь, чаще всего следует высокоуровневой структуре системы.

Внутри одного ограниченного контекста может работать довольно большое количество людей, между которыми также может возникнуть недопонимание, напряжение при попытке поддержания единого видения и т.п. Практика показывает, что 3-4 человека, тесно работая вместе, обычно могут договориться, но делить систему на ограниченные контексты по 3-4 человека оказывается непрактично.

Поэтому в рамках ограниченного контекста практикуют «**непрерывную интеграцию**» в её классическом понимании — слияние изменений в основную ветку раз в несколько часов, постоянную (после каждого коммита) сборку и запуск юнит-тестов, обеспечение высокого тестового покрытия, непрерывное общение в рамках команды. Всё это позволяет быстро понять наличие проблем в понимании модели внутри ограниченного контекста и устранить их.

Для интеграции с другими ограниченными контекстами могут использоваться «**карты контекстов**» (Context map). Карта контекстов фиксирует, как понятие из одной модели в рамках одного ограниченного контекста транслируется в понятие из другой модели из другого контекста. Карты трансляции обычно просто описываются на естественном языке (с применением терминов единых языков интегрируемых моделей), но могут и использоваться диаграммы такого примерно вида:

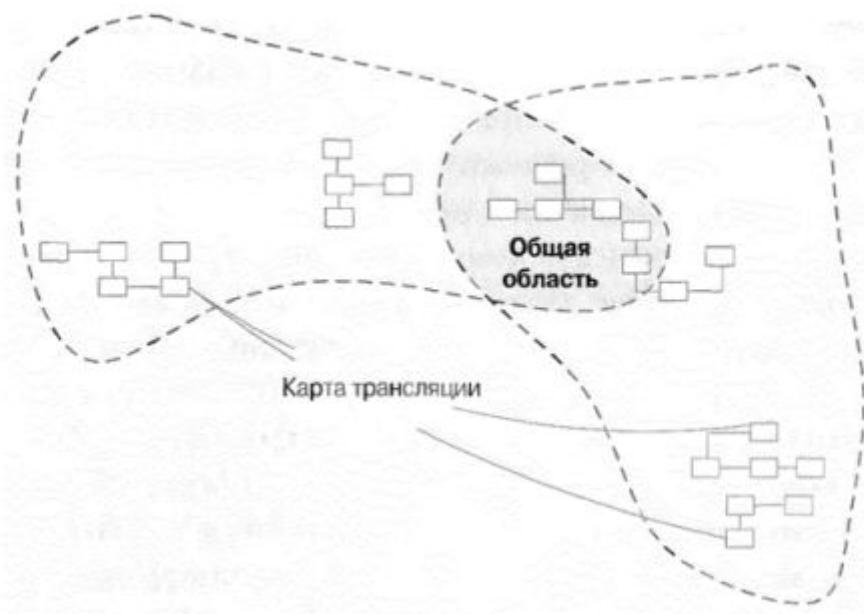


## 67. Подходы к интеграции контекстов.

Есть некоторые типичные ситуации, в которых возможны разные степени интеграции моделей предметной области и разные способы действий, необходимых, чтобы этой интеграции достичь и при этом не навредить разработке.

Первый рассматриваемый здесь паттерн интеграции — «**Общее ядро**» (Shared Kernel) — предполагает наиболее тесную интеграцию, кроме объединения двух

ограниченных контекстов в один. Такой подход предполагает выделение общей части двух ограниченных контекстов и совместную реализацию элементов модели оттуда:



Классы из общей области должны иметь строго одинаковый смысл во всех интегрируемых ограниченных контекстах. Обычно это либо какое-то общее ядро, определяющие базовые понятия всей предметной области проекта, либо это могут быть какие-то классы с данными, которые используются для того, чтобы два ограниченных контекста могли легко и без дополнительных преобразований обмениваться информацией. При этом наличие общего ядра не отрицает применения карты трансляции к элементам модели, которые в общее ядро не попали. Паттерн «Общее ядро» применим только в случае, если две команды могут непосредственно общаться друг с другом, находятся на одном уровне подчинения и не будут мешать друг другу. Классы из общего ядра можно менять только после согласования с обеими командами, так что это дело хлопотное, а необходимость избегать неправильного понимания сущностей и «ересей» в едином языке требует и постоянного неформального общения членов команд.

Паттерн «Заказчик-поставщик» (Customer-Supplier) применяется, когда общение между командами затруднено и/или когда команды находятся в подчинённом отношении друг к другу (не обязательно административно подчинённом, это может быть подчинение в смысле зависимостей между компонентами — например, группа генерации кода может оказаться зависимой от группы синтаксического анализа, особенно если синтаксический анализ нужен не только генерации кода). Особенно паттерн актуален, когда команды имеют потенциально разный цикл релизов и ориентируются на разных конечных пользователей. «Общее ядро» в такой ситуации применять невозможно, поскольку попытка согласовать общее ядро может провалиться из-за близости релиза одной из команд, конфликтующих приоритетов

или конфликтующих миропониманий. Тогда одна из команд рискует оказаться заблокированной.

Чтобы этого избежать, следует явно зафиксировать отношения между командами. Одна из команд выступает в роли заказчика (одного из заказчиков) — участвует в митингах по планированию, поставляет задачи. Вторая команда выступает в роли поставщика, стараясь удовлетворить выставляемым требованиям, в соответствии со своим планом, приоритетами и видением предметной области.

При этом обе команды могут работать в разных ограниченных контекстах, взаимодействующих через чётко определённые точки интеграции. Паттерн предполагает, что обе команды находятся в одной иерархии управления, чтобы возможные конфликты не надо было эскалировать через всё руководство организации. Конфликты неизбежны, если у поставщика несколько заказчиков, а поскольку конфликты могут привести к полной блокировке команды-заказчика, они должны быстро и эффективно разрешаться.

Паттерн «Конформист» (Conformist) применяется, когда команда полностью зависит от компонента, на который никак не может повлиять. Например, это может быть какой-то legacy-код или целое legacy-приложение, либо платформа, на которой мы ведём разработку, либо большая open-source-библиотека (в этих двух случаях как-то повлиять мы можем, большинство адекватных разработчиков принимают пуллреквесты или багрепорты, но это обычно хлопотно, не быстро и требует существенно большего погружения в код компонента, чем вам, возможно, хочется). Это может быть и технология, навязанная «сверху». «Конформист» предполагает, что вы просто принимаете модель и миропонимание «основного» компонента, и подстраиваете свою модель предметной области под модель, которая там используется. Это обычно очень не нравится программистам из-за синдрома «not invented here» — часто всё, что сделали не лично мы, кажется нам каким-то бредом (тем более если это навязано сверху). Тем не менее, часто паттерн «Конформист» на самом деле хорошая идея, потому что legacy-код вполне может оказаться написанным командой, которая уже 20 лет в предметной области и знает о ней существенно больше, чем знаете вы, или имеете надежду узнать в разумные сроки. Практика показывает, что популярные сторонние компоненты обычно отражают вполне хорошее понимание предметной области, и принять его для своего кода правильно — может сэкономить массу времени на анализ и поможет избежать типичных ошибок.

Паттерн «Предохранительный уровень» (Anticorruption Layer) применяется, когда «Конформист» применить нельзя, потому что модель предметной области стороннего компонента вам противна (ну или просто не подходит), однако компонент своё дело делает. В этом случае рекомендуется реализовать программную прослойку между вашим кодом и компонентом, и использовать компонент через эту прослойку, чётко задокументировав, как понятия из одной модели преобразуются в другую. Обратите внимание, речь идёт не о механическом преобразовании данных в разные форматы (хотя это, безусловно, тоже может потребоваться), а именно о преобразовании понятий между разными ограниченными контекстами. То есть создание класса-обёртки просто потому, что вам название какого-то класса или

метода не нравится — вполне оправдано. Неудивительно, ведь модель предметной области — это скорее про единый язык, чем про функциональность.

С технической точки зрения прослойка (тот самый «предохранительный уровень») может быть большим и сложным куском кода, выполняющим нетривиальные действия. При реализации могут помочь паттерны проектирования «Фасад», «Адаптер», службы (то есть статические классы), различного рода трансляторы, кеши и т.д.:



Паттерн «Отдельное существование» (Separate Ways) применяется, когда даже «Предохранительный уровень» неприменим (например, из-за дороговизны разработки подсистемы преобразования), или вообще в ситуациях, когда преимущества от интеграции меньше затрат на неё. «Отдельное существование», как понятно из паттерна, предполагает отсутствие интеграции вообще — вы принимаете решения игнорировать существование другого проекта и не пытаться оттуда ничего переиспользовать.

Паттерн «Служба с открытым протоколом» (Open Host Service) — это что-то вроде «Заказчик-Поставщик» со стороны поставщика, у которого уж очень много заказчиков, так что приглашать их всех на planning-сессии и учесть все их пожелания не получится. В этом случае рекомендуется в каком-то смысле вообще не рассматривать «хотелки» заказчиков (за исключением требований) и строить модель предметной области самим. После чего предоставить публичный интерфейс, представляющий эту модель, и опубликовать саму модель вместе с интерфейсом (обычно в виде текстовой документации). А заказчикам предлагается самим подстраиваться под опубликованную модель (применяя паттерн «Конформист», или «Предохранительный уровень», если они сильно недовольны).

Паттерн «Общедоступный язык» (Published Language) применяется, когда предметная область достаточно популярна, чтобы в ней появилось очень много «заказчиков» и много «поставщиков». Тогда рекомендуется общими усилиями разработать модель предметной области, удовлетворяющую всех поставщиков, стандартизовать её и навязать заказчикам. Эта общая модель предметной области не должна навязывать ничего касательно реализации и должна быть максимально простой, но вместе с тем достаточно выразительной. Для реализации компонентов и сервисов в рамках этой модели могут использоваться свои ограниченные контексты (и не одни), не обязательно совпадающие с опубликованной моделью. Общая

модель служит скорее языком и общей средой для общения различных систем в предметной области.

## 68. Смыслоное ядро, приёмы дистилляции, абстрактное ядро.

Ключевой момент предметно-ориентированного проектирования больших систем — это выделение их самой содержательной части, вынесение из неё всего, непосредственно к ней не относящегося, и аккуратный её дизайн. Эта содержательная часть предметной области называется **смыслоное ядро** (Core Domain).

Только смысловое ядро фактически составляет know-how, то, что выделяет ваш продукт на фоне конкурентов и то, что представляет настоящую ценность, всё остальное в принципе можно смело выложить в open source (а лучше взять из open source, потому что наверняка уже десять раз реализовано). Поэтому смысловое ядро должно быть самой проработанной частью системы.

Смыслоное ядро, поскольку оно самая важная часть системы, должно быть как можно меньше (чтобы с ним легче было работать) и чётко отделённым от остальных компонентов системы. Процесс выделения смыслового ядра Эванс называет **дистилляцией** и приводит несколько дальних советов про то, как это делать.

Первый приём, позволяющий выделить смысловое ядро — это **Domain Vision Statement**. Domain Vision Statement — это короткий документ (примерно на одну страницу), описывающий смысловое ядро и его полезность. Обычно его имеет смысл делать в самом начале проекта, вместе с описанием проекта вообще.

**Выделенное ядро** (Highlighted Core) — это способ уже на готовой модели обозначить смысловое ядро. Бывает двух видов. Первый — дистилляционный документ — это 3-7 страниц текста про то, что составляет смысловое ядро и как его элементы взаимодействуют друг с другом. По сути, развёрнутый Domain Vision Statement, с архитектурой смыслового ядра, как правило, часть архитектурной документации. Второй — Flagged Core — это когда элементы ядра выделены на существующей модели, например, специальным.

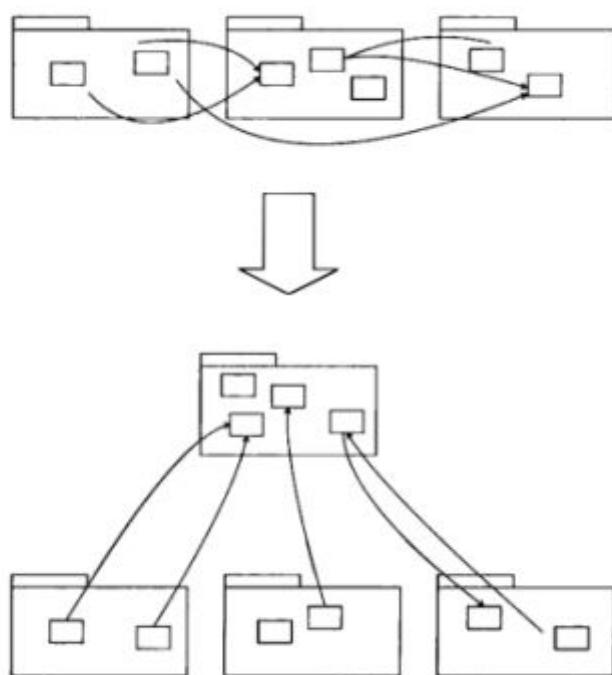
Далее выполняется собственно дистилляция, то есть избавление кода смыслового ядра от всего лишнего. Делается это вынесением неспецифичной для приложения функциональности в отдельные модули. Собственно видов неспецифичной функциональности в DDD выделяют два:

- Неспециализированные подобласти (Generic Subdomains) — куски кода, важные для предметной области, но неспецифичные для конкретной системы;
- Связный механизм (Cohesive Mechanism) — куски кода, неспецифичные для предметной области вообще. Это могут быть разные технические вещи, например, работа с графиками. Иногда оказывается, что хитрая структура самописных классов —

это не более чем замаскированный граф, и тогда стоит просто изменить архитектуру так, чтобы можно было переиспользовать одну из готовых библиотек.

Иногда даже после этих ухищрений смысловое ядро остаётся всё ещё слишком большим, чтобы его можно было адекватно сопровождать. Тогда можно применить приём «**Абстрактное ядро**» — когда смысловое ядро состоит только из абстрактных классов/интерфейсов, которые потом реализуются отдельными модулями.

Радикально сложность модели это всё равно не уменьшит (но так и должно быть, это «essential complexity»), но хотя бы позволит избавиться от ненужных подробностей реализации. Схема такая:



## 69. Крупномасштабная структура, метафора системы, разбиение по уровням.

Крупномасштабная структура — это самая «архитектурная» часть архитектуры, набор основных правил, которым следует вся система или группа систем. Например, может быть задекларировано, что система в целом имеет уровневую архитектуру, и уровни такие-то и такие-то. Тогда как реализация самих уровней может выполняться в совершенно разных архитектурных стилях. Более того, просто задекларировать, что система имеет уровневую архитектуру — это уже зафиксировать её крупномасштабную структуру, хоть и слишком общую, чтобы быть реально полезной. Вообще, крупномасштабная структура системы — это чаще всего уточнённый под конкретную систему архитектурный стиль плюс набор специфичных для системы

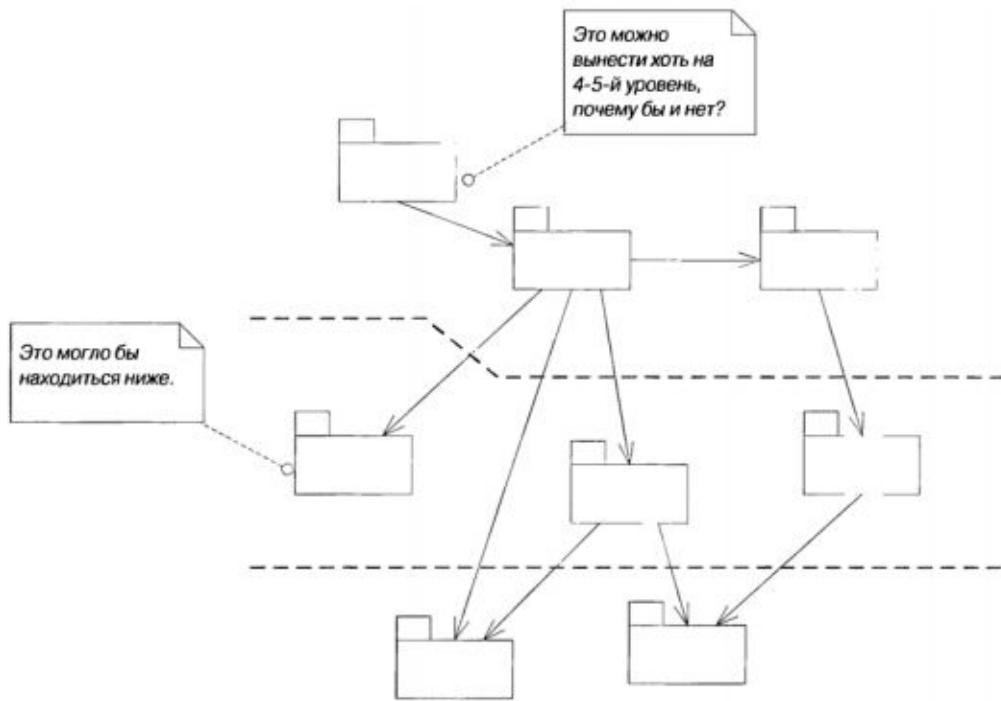
ограничений (например, перечисление основных компонентов и их ответственостей, правила, по которым в систему можно добавлять новые компоненты).

Крупномасштабная структура не фиксируется раз и навсегда, как бы ни был велик соблазн это сделать. По мере роста понимания предметной области в процессе **переработки знаний** и по мере получения обратной связи о том, насколько удобна выбранная архитектура в коде крупномасштабная архитектура вполне может меняться и эволюционировать (естественно, совместно с моделью и с кодом).

Небольшие проекты вполне могут жить без явно задокументированной крупномасштабной структуры — если у вас всего пять классов, то их взаимоотношения будут вполне достаточны в качестве архитектуры. Но надо принимать во внимание, что все большие проекты начинались как небольшие, и если о крупномасштабной структуре не подумать, в какой-то момент возникнет хаос.

**Метафора системы** — некая аналогия, определяющая, как в целом понимать систему. Разные метафоры используются в программировании повсеместно, из-за незримости ПО и желания всё-таки его как-то представлять. Например, метафора рабочего стола для пользовательских интерфейсов, файервола в сетевой безопасности и т.д., даже деревья в программировании называются деревьями, потому что похожи на... деревья (а могли бы — связными ациклическими графами). Однако метафору не всегда легко подобрать, и она не всегда оказывается удачной.

Наиболее типичная крупномасштабная структура — это **уровневая**. Уровневый архитектурный стиль мы неоднократно уже обсуждали, но есть ещё одно важное соображение, касающееся использования уровней как именно крупномасштабной структуры — нельзя чисто механически делить систему на уровни топологической сортировкой графа зависимостей компонентов. Например,



Разбиение по уровням, как и любое разбиение в архитектуре вообще, должно сообщать дополнительную информацию об идее реализации системы, выражать мысль архитектора и объяснять происходящее. Причём, объяснять — на самом деле, самое важное: качество архитектуры, как научной теории в представлении философов науки 20-го века, определяется её объясняющей способностью. Не важно, что архитектура позволяет быстро или эффективно что-то реализовать, она всё равно плоха, если не объясняет происходящего. Кстати, поэтому результаты автоматической генерации диаграмм по коду — не архитектура ни разу. Чтобы избежать бессмысленного разбиения на уровни, надо каждому уровню придать какое-то значение из предметной области и ограничения, с ним связанные.

## 70. Типичные уровни в производственных системах.

Для систем автоматизации производства типична следующая четырёхуровневая крупномасштабная структура:

Принятия решений	Аналитические механизмы	Практически отсутствует как состояние, так и его изменение	Анализ управления Оптимизация использования Сокращение рабочего цикла
Регламентный	Стратегии Связи-ограничения (на основании целей или закономерностей данной отрасли)	Медленное изменение состояния	Приоритет изделий Предписанные регламенты изготовления деталей
Операционный	Состояние, отражающее реальное положение дел (деятельности и планов)	Быстрое изменение состояния	Инвентарная опись Учет состояния незаконченных деталей
Потенциальный	Состояние, отражающее реальное положение дел (ресурсов)	Изменение состояния в среднем темпе	Возможности оборудования Наличие оборудования Перемещение по территории

Самый нижний уровень, Потенциальный, как и у нас, отражает наши возможности — оборудование, производственные площади, контракты с постоянными поставщиками и т.д. Операционный уровень отвечает за оперативное управление — это конкретно какие детали сейчас есть на складе, статус производства, оперативный план и его выполнение, рабочие на рабочих местах. Регламентный уровень отвечает за ограничения, которых мы должны придерживаться при управлении и планировании — технические регламенты, законодательство, локальные акты и требования (график отпусков, например), стратегия предприятия. Уровень Принятия решений отвечает за стратегическое управление — анализ и оптимизацию рабочего цикла. Каждый уровень имеет разный темп изменения ситуации: потенциальный меняется относительно редко — оборудование или помещения вряд ли часто меняются в течение одного рабочего дня. Данные на операционном уровне обновляются постоянно. Данные на регламентном уровне обновляются вообще раз в несколько лет. Уровень принятия решений своего состояния как правило вообще не содержит, пользуясь данными, которые предоставляют уровни ниже. Это можно использовать для планирования схемы БД и оптимизации запросов.

## 71. Типичные уровни в финансовых системах.

Для финансовых систем более типична тоже четырёхуровневая структура, но с другими уровнями:

Принятия решений	Аналитические механизмы	Практически отсутствует как состояние, так и его изменение	Анализ рисков Анализ портфелей Средства ведения переговоров
Регламентный	Стратегии Связи-ограничения (на основании целей или закономерностей данной отрасли)	Медленное изменение состояния	Пределы резервов Цели размещения активов
Обязательство	Состояние, отражающее сделки и договоры с клиентами	Изменение состояния в среднем темпе	Соглашения с клиентами Соглашения по синдикации
Операционный	Состояние, отражающее реальное положение дел (деятельности и планов)	Быстрое изменение состояния	Состояние кредитов Начисления Выплаты и распределения

Тут Потенциальный уровень отсутствует вообще, потому что финансовые системы не требуют особого оборудования (по крайней мере, того, чем надо реально управлять — если вы не Сбербанк), они оперируют деньгами. Собственно финансовые операции проводятся на операционном уровне, самом нижнем в такой схеме. Однако добавляется уровень Обязательств — это на самом деле то, в чём заинтересована финансовая организация и то, что она на самом деле продаёт (например, кредит — выглядит он так, будто вам продают деньги, но на самом деле это обмен обязательствами). На уровне Обязательств существуют договора, сделки и т.д. Регламентный уровень и уровень Принятия решений имеют тот же смысл и те же особенности, что и для производственных систем.

## 72. Стили «Уровень знаний», «Подключаемые компоненты».

Помимо уровневой, бывают и другие высокоуровневые структуры, причём тысячи их. Например, **Подключаемые компоненты** (Pluggable Component Framework) — когда у нас есть ядро системы и плагинная система, которая позволяет динамически подключать/отключать плагины, которые и реализуют большую часть полезной функциональности.

Ещё интересная структура — это «Уровень знаний», когда система содержит в себе данные операционного уровня и метаданные (данные о данных) на уровне знаний:



Уровень знаний — это обычные объекты в памяти, просто они определяют, что и как можно делать с объектами операционного уровня (например, рефлексия — на уровне знаний объекты типа Type/Class, на операционном уровне — обычные объекты). Такой подход мы использовали, естественно, при проектировании систем визуального моделирования и метамоделирования, там правила визуального языка естественно представляются с помощью уровня знаний, который можно редактировать даже прямо во время выполнения, от чего операционный уровень начинает вести себя по-другому.

В финансовых системах такой подход позволяет на уровень знаний вынести финансовые продукты — например, хитрые правила начисления процентов по вкладам, которые постоянно меняются и реализовывать их в коде было бы весьма неразумно. Регламенты с уровня знаний там затем применяются к финансовым потокам на операционном уровне, при этом система становится очень гибкой — при наличии годного редактора правила может менять даже финансист, который программировать вообще не умеет. Конечно, у такого подхода есть и недостаток — логика уровня знаний не проверяется компилятором и имеет ту семантику, которую криворукие программисты реализуют, так что чрезмерное увлечение такими делами (когда у вас вся система представляется в виде правил на XML, например) может превратить поддержку в настоящий кошмар.

### 73. Архитектура распределённых систем: понятие распределённой системы, типичные архитектурные стили.

Распределённые системы:

- ▶ Компоненты приложения находятся в компьютерной сети
- ▶ Взаимодействуют через обмен сообщениями
- ▶ Основное назначение — работа с общими ресурсами
- ▶ Особенности:
  - ▶ Параллельная работа
  - ▶ Независимые отказы
  - ▶ Отсутствие единого времени

Виды сущностей в распределенной системе:

- ▶ Узлы-процессы-потоки — сущности уровня ОС (или сами вычислительные узлы, если ОС не поддерживает даже процессы)

- ▶ Объекты — обычные объекты из ООП, с интерфейсами, описанными на IDL, вызывающие друг друга по сети
- ▶ Компоненты — более высокоуровневые сущности, как правило, предполагают middleware
- ▶ Веб-сервисы — ещё более высокоуровневые сущности, независимые приложения с чётко определённым способом их использовать

#### Виды взаимодействия

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

#### Типичные архитектурные стили

- ▶ Уровневая архитектура
  - ▶ 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

## 74. Межпроцессное сетевое взаимодействие, модель OSI, стек протоколов TCP/IP, сокеты, протоколы «запрос-ответ», протокол HTTP.

**Межпроцессное взаимодействие** (англ. *inter-process communication, IPC*) — обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром ОС или процессом, использующим механизмы ОС и реализующим новые возможности IPC. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.

**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 определённые функции при таком взаимодействии.

**Стек протоколов TCP/IP** — набор сетевых протоколов, на которых базируется [Интернет](#). Обычно в стеке TCP/IP верхние 3 уровня ([прикладной](#), [представления](#) и [сеансовый](#)) модели OSI объединяют в один — прикладной. Поскольку в таком стеке не предусматривается унифицированный протокол передачи данных, функции по определению типа данных передаются приложению.

Уровни стека TCP/IP:

*Канальный уровень* описывает каким образом передаются пакеты данных через физический уровень, включая [кодирование](#) (то есть специальные последовательности битов, определяющих начало и конец пакета данных).

*Сетевой уровень* изначально разработан для передачи данных из одной (под)сети в другую. Примерами такого протокола является X.25 и [IPC](#) в сети [ARPANET](#). С развитием концепции глобальной сети в уровень были внесены дополнительные возможности по передаче из любой сети в любую сеть, независимо от протоколов нижнего уровня, а также возможность запрашивать данные от удалённой стороны.

Протоколы *транспортного уровня* могут решать проблему негарантированной доставки сообщений («дошло ли сообщение до адресата?»), а также гарантировать правильную последовательность прихода данных.

На *прикладном уровне* работает большинство сетевых приложений. Эти программы имеют свои собственные протоколы обмена информацией, например, [HTTP](#) для [WWW](#), [FTP](#) (передача файлов), [SMTP](#) (электронная почта), [SSH](#) (безопасное соединение с удалённой машиной), [DNS](#) (преобразование символьных имён в [IP-адреса](#)) и многие другие.

**Сокет** (англ. *socket* — разъём) — название [программного интерфейса](#) для обеспечения обмена данными между [процессами](#). Процессы при таком обмене могут исполняться как на одной [ЭВМ](#), так и на различных ЭВМ, связанных между собой [сетью](#). Сокет — [абстрактный](#) объект, представляющий конечную точку соединения.

Следует различать [клиентские](#) и [серверные](#) сокеты. Клиентские сокеты грубо можно сравнить с конечными аппаратами [телефонной сети](#), а серверные — с [коммутаторами](#). Клиентское приложение (например, [браузер](#)) использует только клиентские сокеты, а серверное (например, [веб-сервер](#), которому браузер посыпает запросы) — как клиентские, так и серверные сокеты.

### “Запрос-ответ” поверх TCP

+ Использование потоков вместо набора пакетов

- ▶ Удобная отправка больших объёмов данных
- ▶ Один поток на всё взаимодействие

+ Интеграция с потоками ОО-языков

+ Надёжность доставки

- ▶ Отсутствие необходимости проверок на уровне бизнес-логики
  - ▶ Уведомления в пакетах с ответом
  - ▶ Упрощение реализации
- Тяжеловесность коммуникации

## HTTP

- ▶ Пример протокола “запрос-ответ”
- ▶ Реализован поверх TCP
- ▶ Соединение на всё время взаимодействия
- ▶ Маршалинг данных в ASCII
  - ▶ MIME
- ▶ HTTP 2.0
  - ▶ Бинарный протокол
  - ▶ Обязательное шифрование
  - ▶ Мультиплексирование запросов в одном TCP соединении
  - ▶ “Предсказывающая посылка данных”

## 75. Удалённые вызовы процедур (RPC). Protobuf, gRPC.

Удалённый вызов процедур, реже Вызов удалённых процедур (от англ. Remote Procedure Call, RPC) — класс технологий, позволяющих компьютерным программам вызывать функции или процедуры в другом адресном пространстве (на удалённых компьютерах, либо в независимой сторонней системе на том же устройстве). Обычно реализация

RPC-технологии включает в себя два компонента: сетевой протокол для обмена в режиме клиент-сервер и язык сериализации объектов (или структур, для необъектных RPC).

**Protocol Buffers** — [протокол сериализации](#) (передачи) структурированных данных, предложенный [Google](#) как эффективная бинарная альтернатива текстовому формату [XML](#). Разработчики сообщают, что Protocol Buffers проще, компактнее и быстрее, чем [XML](#), поскольку осуществляется передача бинарных данных, оптимизированных под минимальный размер сообщения

Механизм сериализации-десериализации данных

- ▶ Компактное бинарное представление
- ▶ Декларативное описание формата данных, генерация кода для языка программирования
- ▶ Поддерживается Java, Python, Objective-C, C++, Go, JavaNano, Ruby, C#
- ▶ Бывает v2 и v3, с некоторыми синтаксическими различиями
- ▶ Хитрый протокол передачи,  
<https://developers.google.com/protocol-buffers/docs/encoding>
- ▶ До 10 раз компактнее XML

## gRPC

- ▶ средство для удалённого вызова (RPC)
- ▶ Работает поверх protobuf
- ▶ Разрабатывается Google
- ▶ Поддерживает C++, Java, Objective-C, Python, Ruby, Go, C#, Node.js

Сервисы описываются в том же .proto-файле, что и протокол protobuf-а

- ▶ В качестве типов параметров и результатов — message-и protobuf-а

```
service RouteGuide {
```

```
rpc GetFeature(Point) returns (Feature) {}
```

```
rpc ListFeatures(Rectangle) returns (stream Feature) {}
```

```
rpc RecordRoute(stream Point) returns (RouteSummary) {}
```

```
rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
```

```
}
```

- ▶ Сборка — плагином grpc к protoc

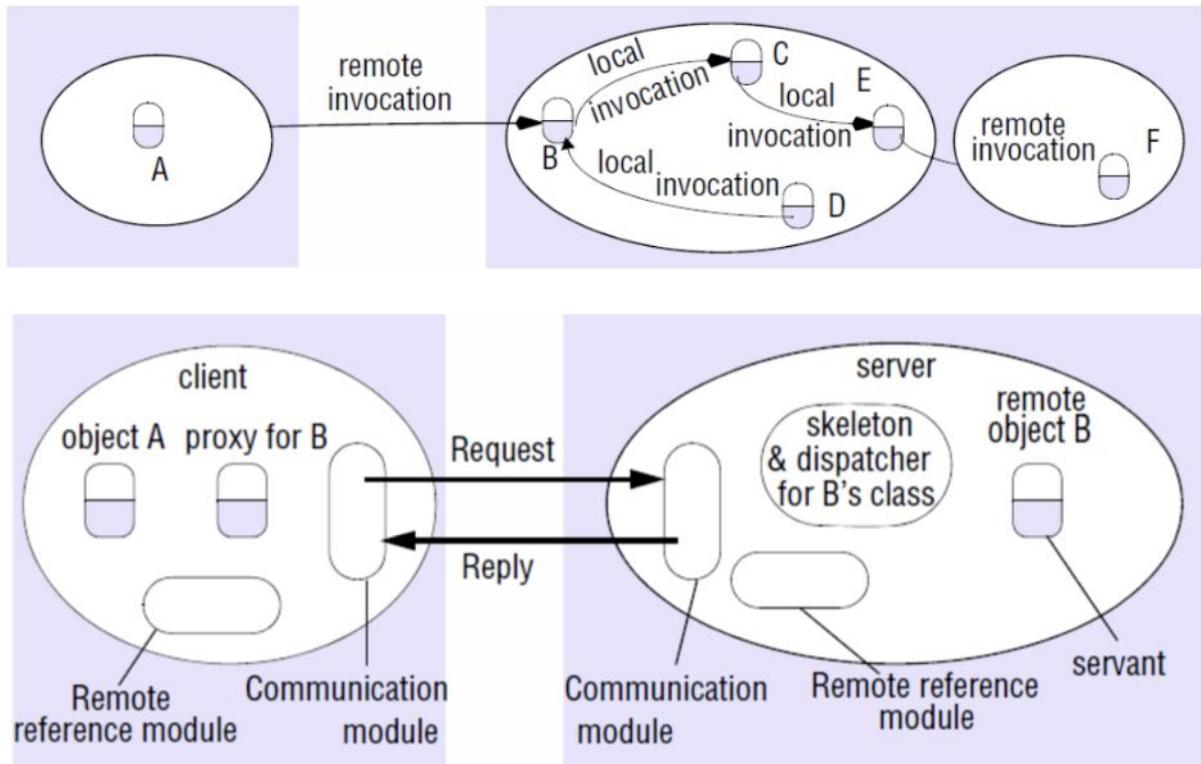
## 76. Удалённые вызовы методов (RMI).

Продолжение идей RPC

- ▶ Программирование через интерфейсы
- ▶ Работа поверх протоколов “запрос-ответ”
- ▶ At-least-once или at-most-once семантика вызовов
- ▶ Прозрачность синтаксиса вызовов
- ▶ Особенности ОО-программ
- ▶ Наследование, полиморфизм
- ▶ Передача параметров по ссылкам
- ▶ Исключения
- ▶ Распределённая сборка мусора

## Локальные и удалённые вызовы

- ▶ Локальные и удалённые объекты
- ▶ Интерфейсы удалённых объектов
- ▶ Ссылки на удалённые объекты
- ▶ Как параметры или результаты удалённых вызовов



## 77. Веб-сервисы, SOAP. WCF.

SOAP (от англ. Simple Object Access Protocol — простой протокол доступа к объектам) — протокол обмена структурированными сообщениями в распределённой вычислительной среде.

Windows Communication Foundation (WCF) — программный фреймворк, используемый для обмена данными между приложениями, входящий в состав .NET Framework.

- ▶ Simple Object Access Protocol
- ▶ Web Services Description Language
- ▶ Universal Discovery, Description and Integration

## SOAP-сообщение

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://example.org/alert">
      <m:msg>Get up at 6:30 AM</m:msg>
    </m:alert>
  </env:Body>
</env:Envelope>
```

### Достоинства SOAP-based сервисов

- ▶ Автоматический режим описания сервисов
- ▶ Автоматическая поддержка описаний SOAP-клиентом
- ▶ Автоматическая валидация сообщений
- ▶ Валидность xml
- ▶ Проверка по схеме
- ▶ Проверка SOAP-сервером
- ▶ Работа через HTTP
- ▶ Хоть через обычный GET

### Недостатки SOAP-based сервисов

- ▶ Огромный размер сообщений
- ▶ Сложность описаний на клиенте и сервере
- ▶ Один запрос — один ответ
- ▶ Поддержка транзакций на уровне бизнес-логики

- ▶ Сложности миграции при изменении описания

## Пример: WCF

- ▶ Платформа для создания веб-сервисов
- ▶ Часть .NET Framework, начиная с 3.0
- ▶ Умеет WSDL, SOAP и т.д., очень конфигурируема
- ▶ Автоматическая генерация заглушек на стороне клиента
- ▶ ABCs of WCF:
  - ▶ Address
  - ▶ Binding
  - ▶ Contract



## 78. Очереди сообщений, RabbitMQ.

На Java можно юзать тут паттерн “Наблюдатель”.

### Очереди сообщений

#### Коротко:

Используются для гарантированной доставки сообщений

- ▶ Даже если отправитель и получатель доступны в разное время
- ▶ Локальное хранилище сообщений на каждом устройстве
- ▶ Реализуют модель “издатель-подписчик”, но могут работать и в режиме “точка-точка”
- ▶ Как правило, имеют развитые возможности маршрутизации, фильтрации и преобразования сообщений
- ▶ Разветвители, агрегаторы, преобразователи порядка

#### Подробнее:

Сообщения, наряду с блоками вычисления и хранения, составляют три основных блока почти в каждой блок-схеме системы. Очереди сообщений, по существу, являются связующим звеном между различными процессами в ваших приложениях и обеспечивают надежный и масштабируемый интерфейс взаимодействия с другими подключенными системами и устройствами.

**Очередь** — структура данных с дисциплиной доступа к элементам «первый пришёл — первый вышел». Добавление элемента возможно лишь в конец очереди, выборка — только из начала очереди, при этом выбранный элемент из очереди удаляется.

## Использование очереди сообщений

Десять причин, почему очереди сообщений являются жизненно важным компонентом для любой архитектуры или приложения:

- **Слабое связывание** — очереди сообщений создают неявные интерфейсы обмена данными, которые позволяют процессам быть независимыми друг от друга т.е вы просто определяете формат сообщений отправляемых от одного процесса другому.
- **Избыточность** — Очереди позволяют избежать случаев незадокументированного использования ресурсов процесса(например памяти) в результате хранения необработанной (лишней) информации.
- **Масштабируемость** — очереди сообщений позволяют распределить процессы обработки информации. Таким образом, они позволяют легко наращивать скорость, с которой сообщения добавляются в очередь и обрабатываются.
- **Эластичность** и возможность выдерживать пиковые нагрузки — очереди сообщений могут выполнять роль своего рода буфера для накопления данных в случае пиковой нагрузки, смягчая тем самым нагрузку на систему обработки информации и не допуская ее отказа.
- **Отказоустойчивость** — очереди сообщений позволяют отделить процессы друг от друга, так что если процесс, который обрабатывает сообщения из очереди падает, то сообщения могут быть добавлены в очередь на обработку позднее, когда система восстановится.
- **Гарантированная доставка** — использование очереди сообщений гарантирует, что сообщение будет доставлено и обработано в любом случае (пока есть хотя бы один обработчик).
- **Гарантированный порядок доставки** — большая часть систем очередей сообщений способны обеспечить гарантии того, что данные будут обрабатываться в определённом порядке (чаще всего в том порядке в котором они поступили).
- **Буферизация** — очереди сообщений позволяет отправлять и получать сообщения при этом работая с максимальной эффективностью, предлагая буферный слой — процесс записи в очередь может происходить настолько быстро, насколько быстро это в состоянии выполнить очередь сообщений, а не обработчик сообщения.
- **Понимание потоков данных** — очереди сообщений позволяют выявлять узкие места в потоках данных приложения, легко можно определить какая из очередей забивается, какая простаивает и определить что необходимо делать — добавлять новых обработчиков сообщений или оптимизировать текущую архитектуру.
- **Асинхронная связь** — очереди сообщений предоставляют возможность асинхронной обработки данных, которая позволяет поместить сообщение в очередь

без обработки, позволяя системе обработать сообщение позднее, когда появится возможность.

В целом области применения очередей сообщений включают в себя:

- Обработку данных
- Буферизацию потоков данных
- Управление процессами
- Интеграцию и взаимодействие систем

Вышеприведённые области должны дать вам идеи где можно использовать очереди сообщений. Если они не являются стандартной частью вашего инструментария, вы, вероятно, пропустили то, что может уменьшить сложность вашей системы и решить кучу проблем.

## 79. Архитектурный стиль REST.

**Коротко:**

Сервер и клиенты системы надёжной передачи сообщений

► Сообщение посыпается на сервер и хранится там, пока его не

заберут

► Продвинутые 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

**Подробнее:**

RabbitMQ – это брокер сообщений. Его основная цель – принимать и отдавать сообщения. Его можно представлять себе, как почтовое отделение: когда Вы бросаете письмо в ящик, Вы можете быть уверены, что рано или поздно почтальон доставит его адресату [видимо, автор ни разу не имел дела с Почтой России]. В этой аналогии RabbitMQ является одновременно и почтовым ящиком, и почтовым отделением, и почтальоном.

Наибольшее отличие RabbitMQ от почтового отделения в том, что он не имеет дела с бумажными конвертами – RabbitMQ принимает, хранит и отдает бинарные данные – сообщения.

В RabbitMQ, а также обмене сообщениями в целом, используется следующая терминология:

- **Producer** (поставщик) – программа, отправляющая сообщения. В схемах он будет представлен кругом с буквой «Р»

- **Queue** (очередь) – имя «почтового ящика». Она существует внутри RabbitMQ. Хотя сообщения проходят через RabbitMQ и приложения, хранятся они только в очередях. Очередь не имеет ограничений на количество сообщений, она может принять сколько угодно большое их количество – можно считать ее бесконечным буфером. Любое количество поставщиков может отправлять сообщения в одну очередь, также любое количество подписчиков может получать сообщения из одной очереди. В схемах очередь будет обозначена стеком и подписана именем
- **Consumer** (подписчик) – программа, принимающая сообщения. Обычно подписчик находится в состоянии ожидания сообщений. В схемах он будет представлен кругом с буквой «С»

Поставщик, подписчик и брокер не обязаны находиться на одной физической машине, обычно они находятся на разных.

## Hello World!

Первый пример не будет особо сложным – давайте просто отправим сообщение, примем его и выведем на экран. Для этого нам потребуется две программы: одна будет отправлять сообщения, другая – принимать и выводить их на экран.

Общая схема такова:



Поставщик отправляет сообщения в очередь с именем «hello», а подписчик получает сообщения из этой очереди.

## Пример, отправитель

```
using System;
using RabbitMQ.Client;
using System.Text;

class Send
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using (var connection = factory.CreateConnection())
        {
            using (var channel = connection.CreateModel())
            {
                channel.QueueDeclare(queue: "hello", durable: false, exclusive: false,
                                     autoDelete: false, arguments: null);

                string message = "Hello World!";
                var body = Encoding.UTF8.GetBytes(message);

                channel.BasicPublish(exchange: "", routingKey: "hello",
                                     basicProperties: null, body: body);
            }
        }
    }
}
```

## Пример, получатель

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Text;

class Receive
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using (var connection = factory.CreateConnection())
        using (var channel = connection.CreateModel())
        {
            channel.QueueDeclare(queue: "hello", durable: false, exclusive: false, autoDelete: false, arguments: null);

            var consumer = new EventingBasicConsumer(channel);
            consumer.Received += (model, ea) =>
            {
                var body = ea.Body;
                var message = Encoding.UTF8.GetString(body);
                Console.WriteLine($" [x] Received {0}", message);
            };
            channel.BasicConsume(queue: "hello", autoAck: true, consumer: consumer);
        }
    }
}
```

## 80. Микросервисы, peer-to-peer.

- ▶ Набор небольших сервисов
- ▶ Разные языки и технологии
- ▶ Каждый в собственном процессе
- ▶ Независимое развёртывание
- ▶ Децентрализованное управление

Основные особенности

- ▶ Микросервисы и SOA (*service-oriented architecture*)
- ▶ Smart endpoints and dumb pipes
- ▶ Проектирование под отказ
- ▶ Асинхронные вызовы
- ▶ Децентрализованное управление данными

### **peer-to-peer**

Одноранговая, децентрализованная, или пиринговая (англ. peer-to-peer, P2P — равный к равному) сеть — оверлейная компьютерная сеть, основанная на равноправии участников. Часто в такой сети отсутствуют выделенные серверы, а каждый узел (peer) является как клиентом, так и выполняет функции сервера. В отличие от архитектуры клиент-сервера, такая организация позволяет сохранять работоспособность сети при любом количестве и любом сочетании доступных узлов. Участниками сети являются все пиры.

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

## 81. Развёртывание и балансировка нагрузки, Docker.

### Docker

- ▶ Средство для “упаковки” приложений в изолированные контейнеры
- ▶ Что-то вроде легковесной виртуальной машины
- ▶ Широкий инструментарий: DSL для описания образов,

### Docker Image

- ▶ Окружение и приложение
- ▶ Состоит из слоёв
- ▶ Все слои read-only
- ▶ Образы делят слои между собой как процессы делят динамические библиотеки

- ▶ На основе одного образа можно создать другой

Docker Container

- ▶ Образ с дополнительным write слоем
- ▶ Содержит один запущенный процесс
- ▶ Может быть сохранен как

DockerHub

- ▶ Внешний репозиторий образов
- ▶ Официальные образы
- ▶ Пользовательские образы
- ▶ Приватные репозитории
- ▶ Простой CI/CD
- ▶ Высокая доступность

## Dockerfile

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

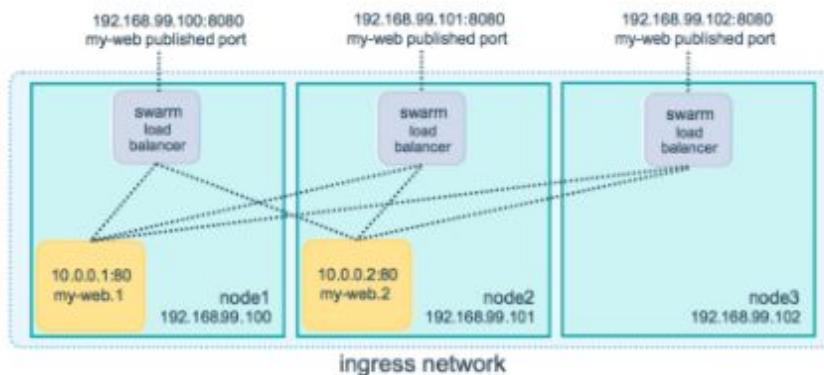
## Балансировка нагрузки

docker-compose.yml

```
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "80:80"
    networks:
      - webnet
networks:
  webnet:
```

## Swarm-ы

- ▶ Машина, на которой запускается контейнер, становится главной
- ▶ Другие машины могут присоединяться к swarm-у и получать копию контейнера
- ▶ Docker балансирует нагрузку по машинам



## 82. Архитектура системы контроля версий Git.

### 4.1. Ключевые требования

Git, как известно, распределённая система контроля версий. Появился он в результате грустной истории с ядром Linux — оно некоторое время хранилось в проприетарной распределённой системе BitKeeper (2000 год, кстати, одна из первых), но потом разработчики ядра что-то не поделили с компанией-производителем (BitMover), и им пригрозили отзывом лицензии. Линусу Торвальдсу пришлось срочно разрабатывать новую систему контроля версий (потому что ему не нравилась система CVS, в которой так же хранились исходники Linux в то время). Поначалу это был просто набор скриптов на bash для управления патчами, приходившими по почте. С самого начала основными требованиями к разработке были следующие.

- Распределённая разработка с тысячей коммитеров — так же, как в BitKeeper и не так, как в CVS, должны были быть поддержаны процессы, при которых каждый участник может работать у себя локально, не мешая ничьей работе и сам определяя, когда его часть готова к публикации. Для большого проекта с открытым исходным кодом, над которым работают тысячи никому ничего не должных энтузиастов, это необходимо.
- Защита от порчи исходников — возможность отменить мердж, смерджиться вручную. Опять-таки, тысячи энтузиастов, которые никому ничего не должны, и половина из которых вообще толком программировать не умеет.
- Высокая скорость работы — речь идёт о сотнях тысяч коммитов всё-таки.

Несколько иронично то, что BitKeeper сам в 2016 году стал опенсорсным, и при этом не то чтобы очень популярен. Никогда не ссорьтесь с opensource-сообществом по поводу лицензий на ПО.

#### 4.2. Представление репозитория

Когда мы набираем git init, создаётся папка .git, где лежит вся информация гитового репозитория. Она имеет следующую структуру:

- HEAD — ссылка на текущую ветку, которую зачекаутили в рабочей папке;
- index — staging area, то место, где формируется информация о текущем коммите;
- config — конфигурационные опции гита для этого репозитория;
- description — «is only used by the GitWeb program, so don't worry about it» (c) Git Book;
- hooks/ — хук-скрипты (возможность исполнить произвольный код при каком-то действии типа коммита);
- info/ — тоже локальные настройки репозитория, сюда можно вписать игнорируемые файлы, которые вы не хотите писать в .gitignore, чтобы их не коммитить;
- objects/ — самое интересное, тут лежит собственно то, что хранится в репозитории;
- refs/ — тут лежат указатели на объекты из objects (ветки, как мы увидим в дальнейшем);
- ... — прочие штуки, которые появляются в процессе жизни репозитория и нам пока не интересны.

## 83. Архитектура командной оболочки Bash.

Примерно 70К строк кода

Исходный автор — Brian Fox, maintainer — Chet Ramey

Первый релиз — 1989

Написан на С

Использует библиотеку Readline

Вот как выглядит диаграмма, характеризующая на высоком уровне архитектуру Bash:

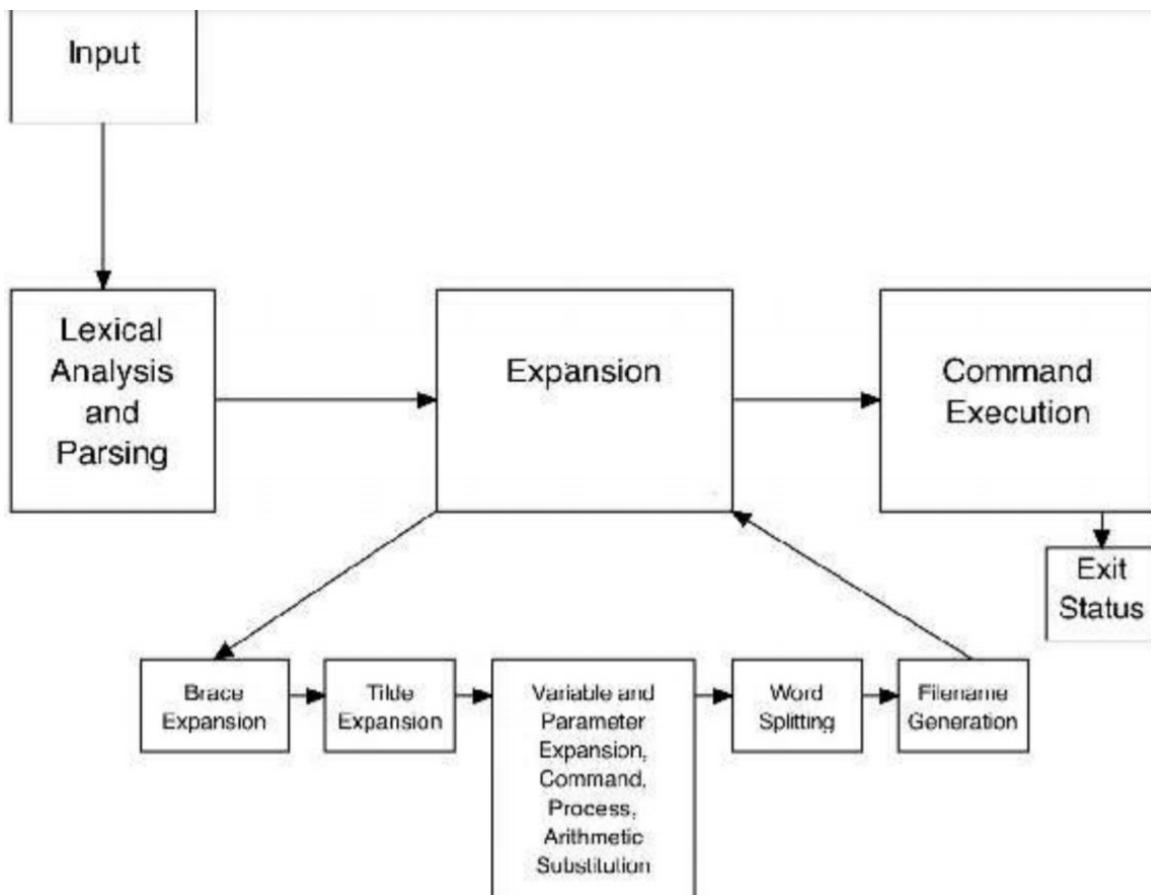


Диаграмма показывает поток данных между основными компонентами системы (в основном, Exit Status на самом деле компонентом системы не является, но я когда-то говорил, что неформальные диаграммы из квадратиков и стрелочек очень популярны и весьма полезны). Ввод (с консоли или из файла) поступает на вход лексическому и синтаксическому анализатору, далее последовательность слов, полученная парсером, отдаётся expansion, который на самом деле представляет собой конвейер, последовательно применяющий преобразования к последовательности слов. Сначала выполняется подстановка фигурных скобок, затем тильды, затем доллара, затем разделение на слова (после парсера, такие дела), затем раскрытие шаблонов. Дальше то, что получилось, подаётся на вход исполнялке команды, которая отвечает за перенаправление ввода-вывода, пайпы, группы процессов и т.д., в итоге получается результат выполнения команды в виде кода возврата.

### 3.1. Ввод с консоли

За ввод с консоли отвечает библиотека Readline, которая отвечает за редактирование командной строки и за хранение истории команд. Она устроена как цикл «считать символ с клавиатуры — найти команду, ему соответствующую — исполнить её — показать результаты». Символ считывается как 8-битный символ (в те времена, когда это писалось, юникод еще не было) и используется как индекс в «таблице диспетчеризации». В этой таблице может быть либо команда (например, «перейти в начало строки»), либо другая такая же таблица (это для поддержки сочетаний из нескольких символов), либо команда «вывести считанный символ». Еще есть макросы (которые просто вставляют во входной поток символы). Все выводимые символы хранятся в буфере редактирования, а когда надо вывести результат на экран, Readline хитро рассчитывает минимальный набор команд

управления курсором, который преобразует текущую отображаемую на экране строку в желаемую. Все внутренние данные хранятся в виде 8-битных символов, но Readline знает (теперь) про юникод и умеет его корректно отображать и корректно считать позиции для многобайтовых символов. Readline может быть расширена добавлением произвольных функций в таблицу диспетчеризации, и Bash этим пользуется, добавляя более 30 своих команд (например, автодополнение).

### 3.2. Синтаксический разбор

Readline возвращает просто строку, введённую пользователем. Первое, что с ней делается — лексический анализ, то есть, в случае с Bash-ем, разделение по словам и их идентификация.

Bash — один из немногих шеллов, написанный на lex + bison, о чём, впрочем, автор несколько сожалеет, говоря, что ручная реализация рекурсивным спуском сильно упростила бы дело. Оригинальная грамматика шелла Борна, которую Bash пытается поддерживать, никому не известна, есть грамматика (контекстно-зависимая), стандартизованная POSIX, её-то и реализует Bash (так что грамматика Bash таки известна и документирована).

Интересно, что подстановка alias-ов выполняется лексером. Правда, для этого парсер сообщает ему, разрешена ли в данный момент подстановка. Лексер же отвечает за кавычки и бэкслеш.

Дополнительные проблемы создаёт подстановка результата выполнения команды и программируемое автодополнение, где тоже могут выполняться произвольные команды прямо в процессе разбора другой команды. Для поддержки таких вещей парсер умеет сохранять своё состояние и корректно восстанавливать его после разбора и исполнения «подкоманды».

Результат работы парсера — одна команда (которая в случае сложных команд типа for может содержать подкоманды), которая передаётся модулю, отвечающему за подстановки.

### 3.3. Подстановки

Подстановки (expansions) работают на уровне слов и могут порождать новые слова и списки слов. Они могут быть довольно сложными.

Ещё бывает подстановка тильды и арифметическая подстановка. Все они выполняются по порядку, то есть подстановщики организованы во что-то вроде конвейера.

Результат подстановки снова разбивается на слова (при этом это делает код, видимо, отличный от лексера). После разбиения происходит замена шаблонов — каждое слово интерпретируется как потенциальный шаблон и пытается сопоставиться с файлом в файловой системе.

### 3.4. Исполнение команд

Команды бывают встроенными (которые модифицируют состояние самого шелла, например, cd) и внешними, которые сами отдельные программы (например, cat). Ещё бывают сложные команды — if, for и т.д.

Каждая команда позволяет перенаправлять свой ввод и вывод (да, даже в for можно направить входной поток из пайпа). Самое сложное в реализации перенаправления — это следить за тем, когда его нужно отменить. Встроенные и внешние команды работают с точки зрения пользователя одинаково, поэтому наивная реализация перенаправления вывода встроенной команды перенаправила бы вывод всего шелла. При этом Bash ещё и следит за файловыми дескрипторами, которые участвуют в редиректе, создавая новые или используя старые при необходимости.

Встроенные команды реализованы как синые функции, принимающие список слов как аргументы, и работают как «обычные» команды, только без запуска отдельного процесса. Единственная тонкость в том, что некоторые команды принимают как аргумент присваивания (например, export), они обрабатывают присваивание по-особому (для этого используются флаги в WORD\_DESC). Обычные присваивания (которые не в export или declare) реализованы на самом деле тоже как команды, но парсятся и обрабатываются немного по-особому. Если присваивание стоит перед обычной командой, то оно передаётся команде и действует до её завершения, если присваивание одно на строке, оно действует на весь шелл.

Внешние команды перед запуском ищутся в файловой системе, при этом шелл смотрит на PATH. Но если в имени команды есть слеши, то не ищет, а исполняет как есть. При этом единожды найдённая команда запоминается в хеш-таблице и дальше сначала ищется в ней. Если команда не нашлась, Bash вызывает функцию, которую можно переопределить, и многие дистрибутивы это делают, предлагая поставить нужный пакет с командой.

Ещё есть некоторые хитрости с управлением процессами, в которых исполняются команды. Есть режим foreground, в котором шелл ждёт завершения процесса с командой, есть background, где шелл запускает команду и тут же читает следующую. При этом Bash умеет переводить команду из одного режима в другой, для чего там есть понятие «Job», как группа процессов, исполняющая команду. Например, пайпы собирают несколько процессов в один Job, который может быть отправлен в фон или в foreground целиком.

### 3.5. Lessons learned

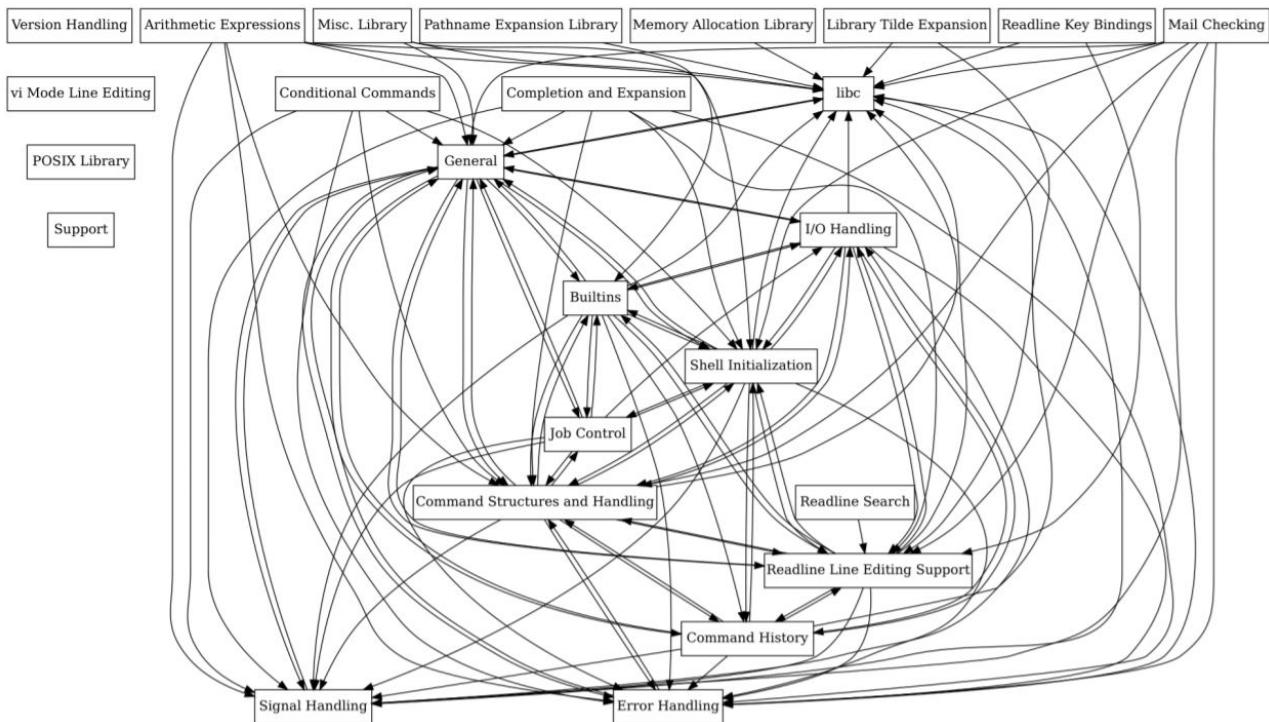
Вот кратко вещи, на которые разработчик Bash Chet Ramey обратил внимание в ходе разработки.

- Хорошие комментарии к коммитам со ссылками на багрепорты с шагами воспроизведения сильно облегчают жизнь.
- Хороший набор тестов — тоже, Bash имеет тысячи тестов, покрывающие практически всю неинтерактивную функциональность.
- Сильно помогли стандарты, как внешние на функциональность шелла, так и внутренние на код.
- Хорошая пользовательская документация важна.
- Переиспользование сильно экономит время.

### 3.6. Как обстоят дела на самом деле

Товарищи из университета Южной Калифорнии исследовали «настоящую» архитектуру Bash с целью получить «базовую» архитектуру, по которой можно было бы оценивать эффективность различных инструментов автоматического восстановления архитектуры.

Один аспирант 80 часов копался в исходниках, после чего отправил результаты автору и тот ещё высказал свои соображения. В итоге получилась вот такая структура системы:

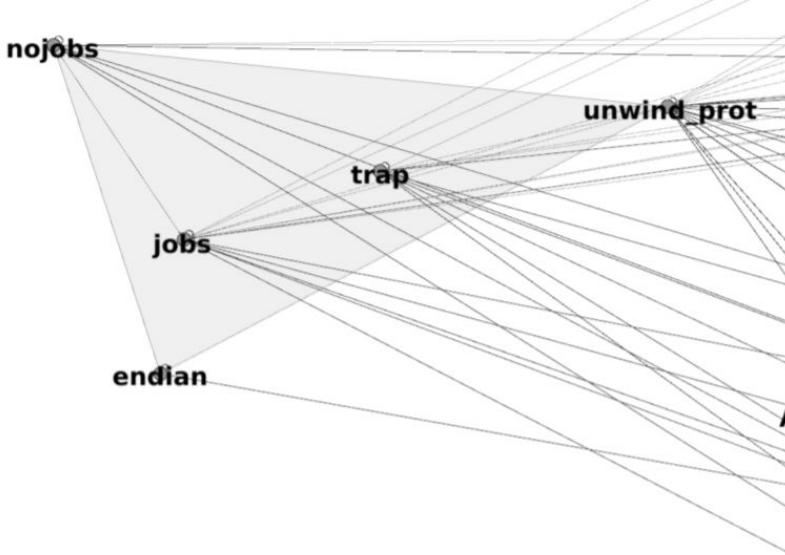


Видно, что зависимостей в коде больше, чем на картинке с концептуальной архитектурой и поток данных на структурной диаграмме совершенно неочевиден. Но некоторые схожести всё-таки есть, например, ввод-вывод реально выделяются в отдельный кластер компонентов.

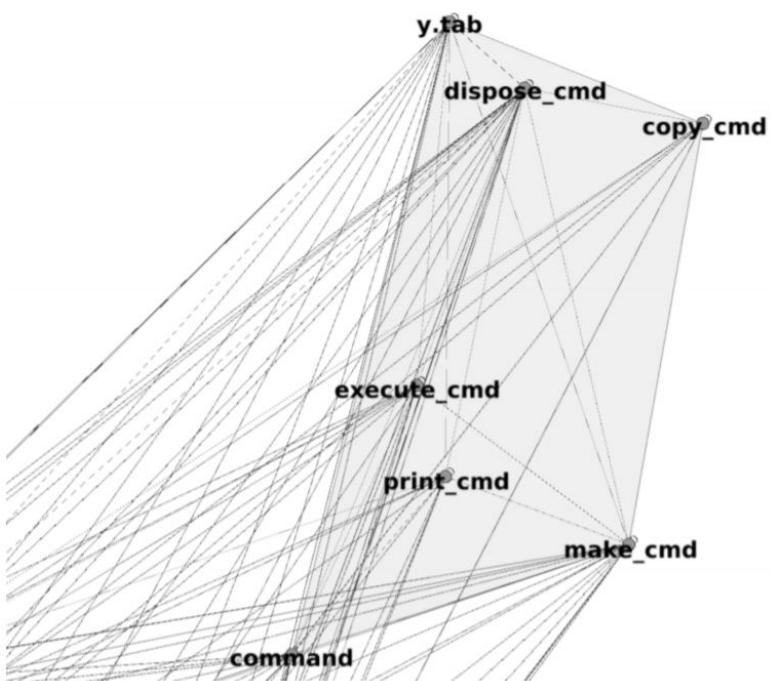
Bash имеет размер порядка 70К строк кода, около 200 отдельных файлов. В ходе восстановления архитектуры было выделено 25 компонентов, из которых 16 относятся к ядру функциональности системы, 9 — вспомогательные. Выяснилось, что структура папок практически не соответствует компонентам, только два компонента имеют свои отдельные папки в исходниках.

Это должно ещё раз проиллюстрировать мысль касательно архитектур реальных приложений — есть архитектура как она проектировалась (*prescriptive architecture*), есть архитектура как она реализована в коде приложения (*descriptive architecture*), и в ходе эволюции приложения эти архитектуры становятся всё больше и больше непохожими друг на друга. Эти расхождения связаны с понятиями «*architectural drift*» (привнесение в архитектуру вещей, которых в исходной архитектуре не было, без нарушения принципов исходной архитектуры) и «*architectural erosion*» (привнесение в реализацию нарушений принципов исходной архитектуры). Для долгоживущих систем архитектурная эрозия

становится довольно критичным фактором, приводящим к тому, что исходное разбиение на компоненты перестаёт быть валидным вообще. Bash в этом плане довольно показателен, поскольку ему много лет. Вот увеличенный вид компоненты управления Job-ами:



Как видим, каждая сущность с этой компоненте больше связана с внешними сущностями, чем с сущностями внутри компоненты, то есть coupling очень высок, а cohesion, судя по всему, очень низок.



Собственно, поэтому важна архитектурная документация и нелишне наличие архитектора, который следил бы за тем, чтобы код и документация не очень расходились. Иначе приложение быстро превратится в гигантский ком кода, где всё зависит от всего и ломается при любом изменении.

## 84. Архитектура компьютерной игры Battle for Wesnoth.