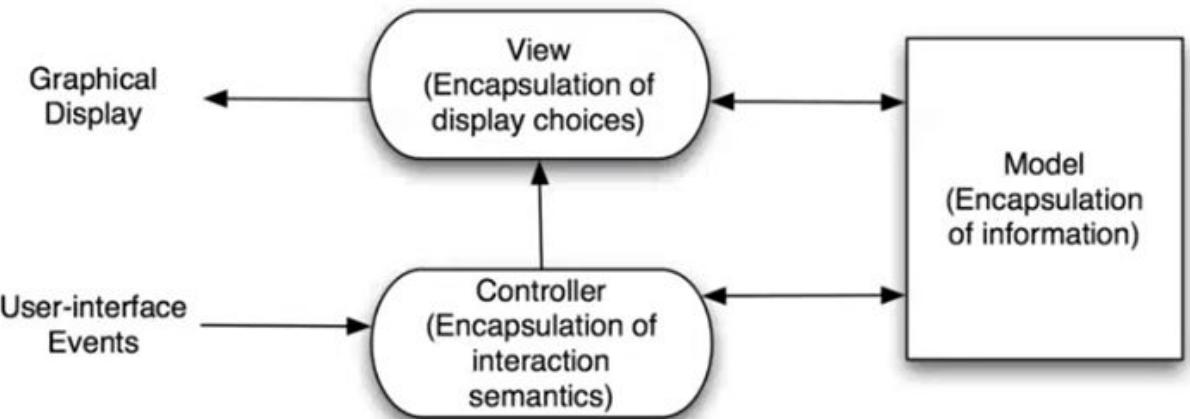


Вопросы



- - Стрелки от вью к модел и от модел к контроллеру?
 - Например, вью читает данные из модели, когда модель сообщила, что что-то изменилось
- Sense-Compute-Control
 - стиль это или **шаблон**. В конспекте оба термина используются
 - Если кроме этого ничего нет, то это стиль.
 - Если что-то сложное есть ещё, то это паттерн
- ??? Но помните, что Dependency Injection имеет и свои недостатки, связанные со сложностью отладки и отсутствием контроля во время компиляции
 - Сложности:
 - мы хз какой будет тип и кудаставить брекпоинт
 - куча всего в стеке вызовов
-
- ~~Зачем неразделяемый конкретный приспособленец? (видимо, чтобы у потомков было общее внешнее состояние?)~~
- 65. Говорящие интерфейсы, функции без побочных эффектов, assertions, замкнутые операции. -- Что хочется от "замкнутых операций"
 - функции возвращали те объекты, с которыми они работают
- 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.
 - Почему не проверяется компилятором? Это не всегда верно? Есть же пример с рефлексией

TODO

- Тупа зафигачить скрины конспектов на пофиг для серча
-
- Структуры паттернов повторить активным повторением

- Перечитать паттерны на Guru Refactoring
- Перечитать арх шаблон и арх стили. В чем их разница?
- ??? Шаблонный метод
- Скипнул при прочтении 2. Единый язык в конспекте; 111 ddd
- Скипнул 3. Модель и реализация 111 ddd
- Скипнул 7. Пример, система грузоперевозок 11 ddd
- Скипнул для 69 билета 3.1. Пример, перевозка грузов в 12 pdf

- Операционный уровень свой для каждого приложения, модель предметной области на всех одна. (Ex. Атака по идеи должна одинаково обсчитываться и в случае просто десктопного приложения, и в распределённом случае, когда есть сервер и клиенты, и всё считается на сервере).

1. Понятие архитектуры, профессия «Архитектор».

Понятие появилось не так давно -- в конце **1990**.

Программа -- Программный продукт (тестирование, документация, сопровождение, обрабатывающая исключения) и программный комплекс (интеграционное тестирование, наложенные интерфейсы между узлами) -- Системный программный продукт. И **архитектура** -- это то, что позволяет создавать большие **системные программные продукты**.

Архитектура, согласно стандарту **ISO/IEC 42010** -- основные понятия или свойства системы в окружающей среде, воплощенной в ее элементах, отношениях и конкретных принципах ее проекта и развития.

Зачем арх-ра?

- В разы сокращает затраты на кодирование
- Это инструмент управления проектом -- оценить сложность через декомпозирование задач, ясность достижений в ходе разработки
- Переиспользование внутренних и третьесторонних компонент
- Анализ качества продукта до его написания

Архитектор

Специалист или группа специалистов, имеющих цель создания и сопровождения архитектуры программных средств, заключающейся в:

- принятии архитектурных решений
- сопровождение архитектуры
- синтезе и документировании решений о структуре
- компонентной составляющей проекта

- оценка требований
- описание и реализация требований программного продукта (вспомни о “башне из слоновой кости”)
- контроль и ревизия реализации проекта
- оценка возможности создания программного проекта
- связующее звено между заказчиком и программистами

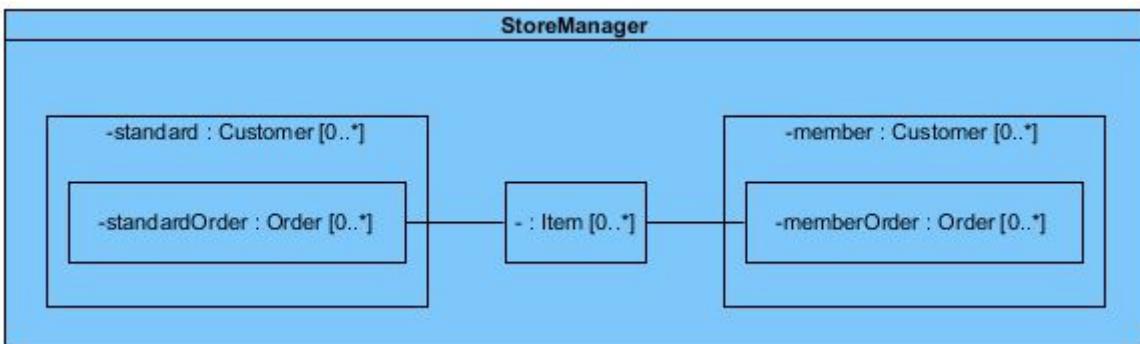
Программист -- специалист с глубиной знаний, архитектор -- с широким кругозором, а также хорошими софт скилами.

2. Архитектурные виды.

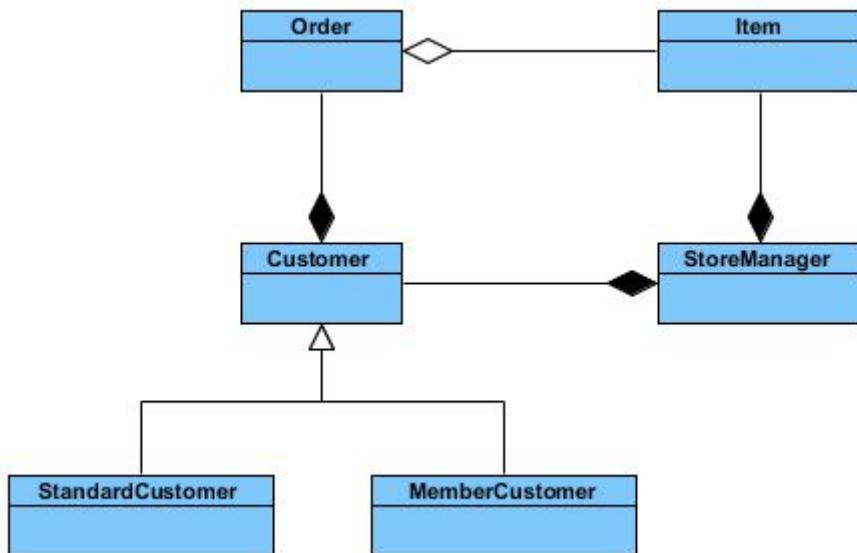
Пример из жизни. При строительстве зданий архитектурными видами могут быть: план водопровода, план этажей. Каждый из которых имеет разное описание с точки зрения формальности и детализированности.

Арх виды согласно стандарту:

1. Контекст -- описывает, что система должна делать, фиксирует окружение системы. Корень иерархии видов, уточняющих систему. Стартовая точка описания архитектуры.
2. Композиция -- описание крупных частей системы и их предназначение. Определение функциональных обязанностей отдельных компонент системы, обеспечение переиспользования, оценка, планирование.
3. Логическая структура -- структура системы в терминах классов и интерфейсов и связи между ними.
4. Зависимости -- определение связей по данным между элементами системы, напр., по передаваемой информации, по потоку управления. Для анализа изменений (что произошло и как быть?), узких мест производительности, планирования (что из этого можно слепить дальше?), интеграционного тестирования (знание что необходимо каждому элементу). Диаграммы компонентов UML, диаграммы пакетов UML.
5. Информационная структура -- определяет персистентный данные, то есть информацию, которую нужно хранить. Определяет схему БД и доступ к данным.
6. Использование шаблонов -- документирование использования **?локальных? (а вот глобальные -- это шо)** паттернов проектирования
7. Интерфейсы -- описание интерфейсов системы, не указанных явно в требованиях к системе, как внешних, так и внутренних. Интерфейс пользователя описывается отдельно в этой точке зрения.
8. Структура системы -- рекурсивное описание внутренних компонентов системы. Используется диаграмма композитных структур UML, диаграммы классов UML, диаграммы пакетов UML



получена из



9. Взаимодействия -- описывает взаимодействиями между сущностями: почему как и когда оно происходит.
10. Динамика состояний -- описание состояний и правил перехода между ними в реактивных системах. Реактивные системы -- системы, основанные на распространении изменений (ООРП, ФРП [хорошо ложится на ФП и параллелизм]). То это системы, 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.
11. Алгоритмы -- описание работы сущностей, методов в деталях.
12. Ресурсы -- описывает используемые ресурсы. Как правило аппаратных или третьесторонние сервисов.

3. Роль архитектуры в жизненном цикле ПО.

Современная точка зрения -- архитектура в центре жизненного цикла ПО

Начало архитектурной активности -- сбор и анализ требований. Это стартовая точка и исходные данные для архитектуры. Требования делятся на:

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

- Производительность системы
- Удобство использования продукта и обучения работы с ним
- Насколько просто сопровождать систему
- Масштабируемость системы -- насколько система справляется с ростом нагрузки
- Надежность
 - Время работы системы до отказа
 - Способность системы восстановиться после отказа
 - Бывает в этом вообще нет смысла (потеря спутника в космосе)
 - Возможность системы корректно завершить работу после сбоя
 - Например, ядерный реактор, банк. системы
- Безопасность
- Ограничения
 - Выбор инструментария, например, iOS
 - Бизнес-ограничения -- со стороны бизнеса, заказчика. Например, закончить проект, дедлайн горит!!

И на архитектуру в большей степени влияют **нефункциональные требования**.

За архитектурой нужно следить на всех стадиях: реализации, тестировании, поддержке. Существуют две проблемы архитектуры:

- Архитектурный дрифт -- появление локальных архитектурных решений в момент реализации, которые не задокументированы при проектировании и которые также обычно не документируются. Это *ad hoc*, вносит хаос.
- Архитектурная эрозия -- отклонения от задокументированной архитектуры. Происходит постоянно (багофиксы, рефакторинг, новая функциональность). Хорошо описанная архитектура и постоянный контроль архитектора позволяет замедлить скорость эрозии.
Но её не избежать. Со временем может потребоваться проводить глобальный рефакторинг архитектуры с переписыванием её на новую. Процесс разработки в этот момент не движется с точки зрения новой функциональности.

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

Сложность

ПО характеризуется двумя сложностями:

- существенная
 - Присуща предметной области. Не избавиться
- случайная

- Вызвана процессом решения задач. Задача архитектуры -- минимизировать случайную сложность. Архитектор управляет сложностью, а не борется с ней.

Основной прием управления сложностью -- это её **сокрытие**.

Сложная программная система, как правило, обладает 3мя свойствами:

- Иерархичность -- элементы системы имеют иерархическое отношение.
- Имеет небольшое кол-во компонент, которые имеют сложные связи друг с другом (что также помогает декомпозировать систему).
- Является эволюцией простой системы.

Декомпозиция

Поэтому для большинства сложных систем есть подходы:

- декомпозиция иерархии системы до элементов, которые можно осмыслить по отдельности и реализовать. В то время как всю систему, может быть, никто не в состоянии осмыслить
- абстрагирование -- сокрытие деталей; рассуждение о типах, а не конкретных элементах; классификация компонент; вынесение общих свойств компонент

Эволюция сложной системы из простой диктует требование проектирования любой системы так, что она перерастет в сложную.

Подходы к декомпозиции.

1. Восходящее проектирование -- изначальное написание отдельных компонент, из которых, как из "кирпичиков", строится вся система. Подходит для исследовательских проектов, где неясно что можно получить в результате, либо если предметная область позволяет комбинировать решения мелких задач.
2. Нисходящее проектирование. От общего к частному. Реализуются компоненты с корня иерархии, наполняясь заглушками. Проверяется не то, что система работает, а то, что делает то что надо и в правильной последовательности. Используется, когда есть четкое видение конечного результата.
 - a. +
 - i. Результат достигается наименьшими усилиями
 - ii. Не пишется ненужный код
 - iii. *Направляет процесс проектирование и разработки*
 - b. -
 - i. Нельзя проверить корректность работоспособности системы (нормально тестировать), пока не реализуются все компоненты

Модульность

Модули -- структурные элементы системы, которые соответствуют подзадачам декомпозированной системы. Модули характеризуются своими *интерфейсами* и *реализацией*. Прежде чем реализовывать модуль нужно четко закрепить контракт, по

которому он будет общаться с другими модулями. То есть сначала система реализуется через интерфейсы с заглушками (пустыми или минимальной реализации).

Модули должны характеризоваться:

- 1 модуль -- 1 функциональность
- Минимальный интерфейс, позволяющий решать данную подзадачу
- Отсутствие знаний о том, как реализованы другие модули
- Четкое место в декомпозированной системе
- Отсутствие побочных эффектов -- работа модуля не должна влиять на другие модули; один результат для одинаковых данных
- Скрытие данных -- у каждого свои данные. Абстрактные данные используются через интерфейс методами модуля.

Интерфейс модуля:

- Синтаксический -- компилятор
- Семантический -- описание использования (Connect, а потом GetData или наоборот?)

Нужен баланс при определении модулей.

- Плохо
 - Один модуль с кучей функциональности
 - стоимость велика, сложно разрабатывать
 - Куча модулей с малой функцией
 - сложно интегрировать
- Хорошо
 - 7+2 сущности может удержать человек. => Это поможет при декомпозиции на модули.

Численные характеристики модульности

- Сопряжение -- мера взаимодействия модуля с другими: насколько часто данный модуль использует другие модули, сколько он должен знать о других модулях, сколько этих модулей в системе.
- Связность -- мера того насколько взаимосвязаны функции внутри модуля и насколько похожие задачи решает модуль. Разная функциональность в одном модуле -- раздели на два модуля.

Цель -- минимизировать сопряжение, максимизировать связность.

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

Объект

Объект, по Thinking of Java (философии Java), -- то, что немного похоже на маленький компьютер, который имеет состояние и который можно попросить сделать то, что он умеет.

Объект имеет три свойства: состояние, поведение, идентичность (ссылки).

Состояние отличает ООП от ФП. С состоянием связано понятие **инвариант**. Объект должен обеспечивать свой инвариант (постоянство состояния) и не давать доступа к нему извне. Например, длина массива -- это инвариант.

Объекты не вызывают методы друг друга напрямую. Они общаются посредством отправки сообщений.

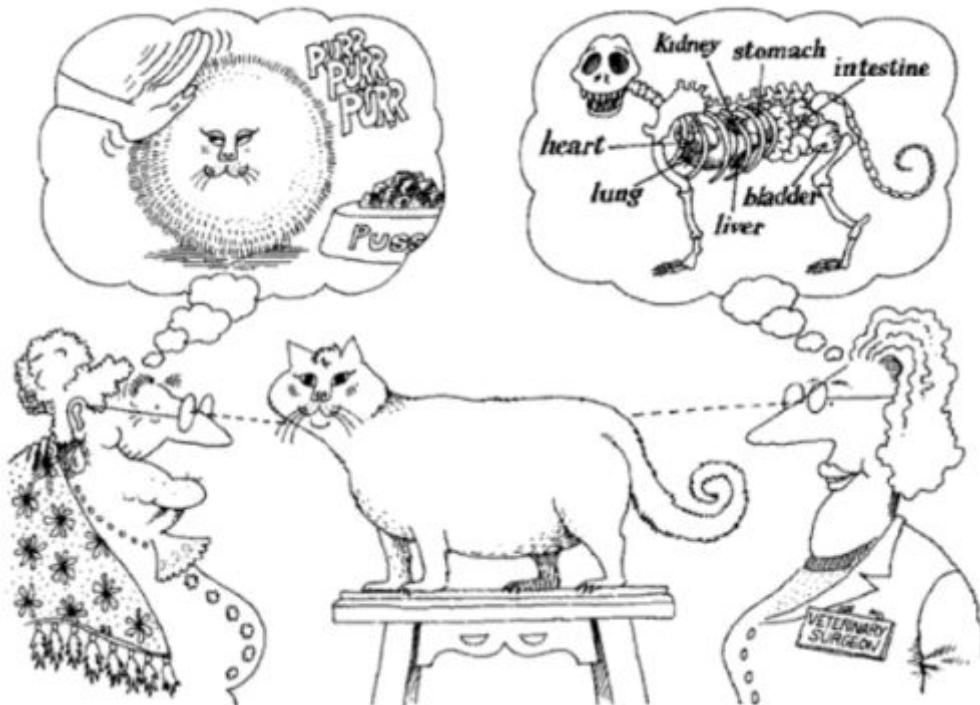
Методы объекта могут быть полиморфны. И то, как будет обработано сообщение, известно только получателю и определяется типом времени выполнения.

Класс

Класс -- это тип объекта. Класс определяет структуру данных, которые *хранят объект*, его поведение (методы) с их реализацией, которые *есть у каждого объекта этого класса*. Класс -- сущность времени компиляции. Объект -- сущность времени выполнения (например, при отладке).

Абстракция

Общее понятие, относящееся не только к ООП. Абстракция выделяет существенные характеристики объекта, отличающего его от остальных, с точки зрения наблюдателя.



Абстракция может иметь несколько взаимозаменяемых реализаций.

На достаточно содержательной абстракции могут основываться другие абстракции. Хороший пример -- математика, в частности алгебра.

Инкапсуляция

Разделяет интерфейс абстракции от её реализации. Обеспечивает инвариантность объекта (защита порчи извне).

Наследование

Это отношение “is-a” (является), реализующее понятие *сабтайпинга* в системе типов.

! Объект типа потомок является одновременно объектом типа предок, поэтому потомок может быть использован везде, где может быть использован предок.

- ! Потомок сохраняет инварианты предка и не вправе их нарушать

Современная архитектурная мысль: наследование -- это НЕ про переиспользование кода. Наследование -- это

- Средства классификации и абстрагирования
- Средство полиморфизма

Для обоих случаев достаточно наследоваться от *интерфейсов*.

Композиция -- это отношение “has-a” (имеет) между типами. Реализуется обычно через private- поля. Гибкое отношение, т.к. работает на этапе выполнения, а не компиляции.

Реализует переиспользование кода.

Полиморфизм -- это предоставление единого интерфейса для сущностей разных типов.

6. Принципы выделения абстракций предметной области.

Действия

при построении объектной модели предметной области:

1. Выделение объектов и их атрибутов через общение с экспертами, изучение предметной области, здравый смысл -- частые **существительные** скорее всего будут классами.
2. Определение действий, которые могут быть выполнены над каждым объектом. То есть назначение ответственности. Действия -- это **глаголы**.
3. Определение связей между объектами, задавая вопросы “кто про это знает”, “где это используется” и т.д.
4. Определение интерфейсов объектов, когда фиксируется и *формализуется* всё, что определено на предыдущих этапах.

Результат действий -- первоначальный набросок архитектуры, обычно через диаграмму классов UML.

Источники абстракций (помимо предметной области):

1. Изоляция сложности -- сложные алгоритмы, сложные структуры данных, целые сложные подсистемы необходимо абстрагировать, чтобы облегчить их использование и обеспечить возможность легко изменять реализацию. *Интерфейсы абстракции нужно тщательно проектировать и упрощать*
2. Изоляция возможных изменений -- прятать все, что с большой вероятностью изменится и что можно будет переписать. Конечно, хорошая архитектура обеспечивает легкое изменение системы, но вот причины возможных изменений:
 - a. бизнес-требования. Например, алгоритмы предметной области могут меняться довольно часто.
 - b. зависимость от оборудование и операционной системы. Например, проблема 2000 года (или Y2K). ПО должно работать многие-многие годы.
 - c. Сложные аспекты проектирования. Сложные архитектурные решения скорее всего плохи. Их нужно легко изменять
 - d. Ввод-вывод, формат сохраняемых файлов, форматы пакетов, используемые команды -- могут изменяться
 - e. Нестандартные особенности языка. ПО должно пережить изменения компилятора
 - f. Зависимость от третьихсторонних компонент. Они могут перестать поддерживаться или стать очень дорогими.
3. Изоляция служебной функциональности, то есть изоляция сущностей, необходимых для работы других сущностей. Например, изолированные сущности используются фабриками, медиаторами, репозиториями, хранилищами.
Классы, отвечающие за бизнес-логику системы должны содержать логику без деталей реализации используемых сущностей.

7. Принципы SOLID.

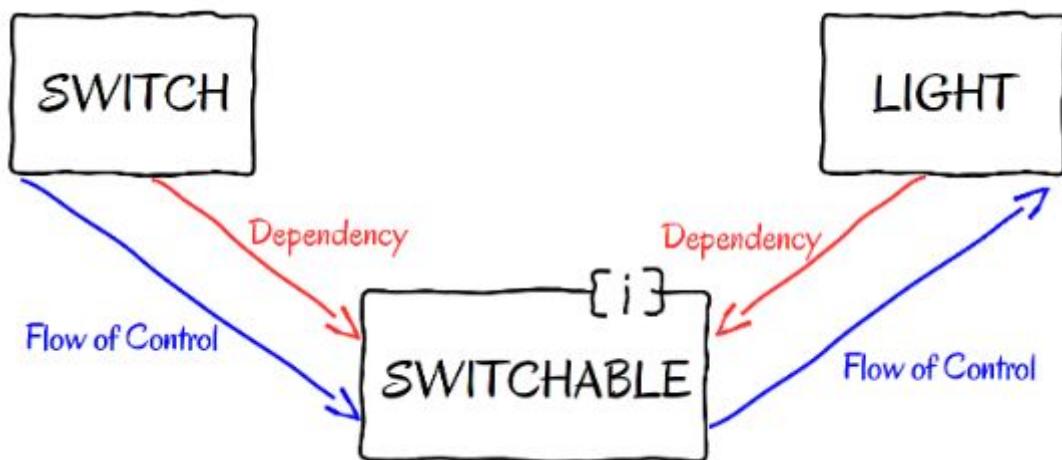
- **S**ingle responsibility principle. Один класс -- одна функциональность. Новые люди проще погружаются в проект с такими классами, такой проект проще сопровождать и развивать.
Описание класса в одно предложение помогает посмотреть на его связность.
- **O**pen/closed principle. Абстракция открыта для расширения, но закрыта для изменений. Если интерфейс абстракции определен, реализован (стабилизирован) и используется, то его менять нельзя. Можно только расширить необходимые абстракции через наследование или через специальные точки для расширения (Слава Богу Архитектуры).
Если же изменять уже созданные абстракции, то это приведет к нарушению принципа **S**, а также, например, может испортить жизнь клиентам.
- **L**iskov substitution principle



Принцип подстановки Барбары Лисков. Это определение понятия наследования. Функции, использующие базовые классы должны иметь возможность использовать классы потомки. Класс потомок реализует интерфейсы предка (синтаксически и Семантически), а также поддерживает инвариант предка.

- **I**nterface segregation principle. Принцип разделения интерфейсов. Клиенты не должны зависеть от методов, которые они НЕ используют.
Абстракция не должна обеспечивать различными группами методов разных клиентов. Одна абстракция -- одна группа методов, кластеризованных по смыслу.
- **D**ependency inversion principle. Принцип инверсии зависимостей. Модули верхних уровней не должны зависеть от модулей нижних уровней, оба типа модулей должны зависеть от абстракции.
Верхний уровень использует абстракцию, которую реализует нижний уровень. Реализуется через Dependency Injection, либо конструктор.

Например, выключатель и лампочка.



Наивно -- в выключатель засунуть класс лампочки => нужно перекомпилировать, нельзя провести мок-тестирование.
DIP -- в выключатель через "has-a" включить абстракцию switchable, реализацию которой выключатель не знает, но может включить, например, и лампочку.

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

Закон Деметры.

“Не разговаривай с незнакомцами”. Объект А не должен получать непосредственный доступ к объекту С, если А имеет доступ к В, а В имеет доступ к С.
 “Крушение поезда”.

НЕ нужно выставлять напоказ внутреннюю структуру данных, не входящих в абстракцию.

“Крушение поезда”. Это раскрывает внутреннюю структуру *ДАННЫХ* объекта, что не даёт построить хорошую абстракцию.

Не путать с LINQ, где паровозик есть, но этот паровозик везет один и тот же объект, не раскрывая внутренней структуры.

Ещё пример -- паттерн строитель.

Принципы хорошего объектно-ориентированного кода.

1. Абстракция.

a. Изменять непонятные поля непонятными константами -- плооооох

- `currentFont.size = 16`

b. Изменять объект через контракт так, как нам удобно -- нойссс

`currentFont.setSizeInPoints(sizeInPoints)`

`currentFont.setSizeInPixels(sizeInPixels)`

2. Не забывать про принцип единственности ответственности

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

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

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

3. Создавать объекты, в которых будет одно поле и геттер+сеттер -- Хорошо! Ибо Скрывается внутренняя структура, которую можно после поменять, а также появляется помощь компилятора, следящего за типами.

Объединение данных, которые можно кластеризовать, в один класс -- то же самое.

Конечно, больше строк кода. Но это цена архитектуры, которую нужно решать в каждом случае.

```
public class Employee {  
    public Employee(  
        FullName name,  
        String address,  
        String workPhone,  
        String homePhone,  
        TaxId taxIdNumber,  
        JobClassification jobClass  
    ) { ... }  
  
    public FullName getName() { ... }  
    public String getAddress() { ... }  
    public String getWorkPhone() { ... }  
    public String getHomePhone() { ... }  
    public TaxId getTaxIdNumber() { ... }  
    public JobClassification getJobClassification() { ... }  
}
```

4. Писать абстракцию, как можно лучше отражающую предметную область (по DDD проект должен быть лучше, чем хочет заказчик).
Сеттеры, например, помогают понять как меняется объект.

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

5. Абстракция не должна делать много того, чего не нужно клиенту. Она должна быть легка для понимания и удобной для использования.

Удобство понимания > удобства использования

```
public class EmployeeRoster implements MyList<Employee> {  
    public void addEmployee(Employee employee) { ... }  
    public void removeEmployee(Employee employee) { ... }  
    public Employee nextItemInList() { ... }  
    public Employee firstItem() { ... }  
    public Employee lastItem() { ... }  
}  


---



```
public class EmployeeRoster {
 public void addEmployee(Employee employee) { ... }
 public void removeEmployee(Employee employee) { ... }
 public Employee nextEmployee() { ... }
 public Employee firstEmployee() { ... }
 public Employee lastEmployee() { ... }
}
```


```

Рекомендации по проектированию классов и интерфейсов

- A. Каждый класс должен быть таким, что его работу можно описать одним предложением. Соблюдайте принцип единственности ответственности.
- B. Реализовывать методы-антагонисты (add/remove, например). Может, они понадобятся в будущем
- C. Разделять методы **команд** от методов **запросов**. Команды меняют объект и возвращают результат. Запросы только возвращают результат.
 - a. Помогает пониманию
 - b. Облегчает параллельность, т.к. синхронизировать нужно только команды.
- D. Возвращать **Option** вместо **null**.
- E. Некорректные состояния делать невыразимыми в системе типов. То есть, например, оперировать своим типом FullName вместо Зёх строковых полей.
- F. Абстракция должна работать правильно в максимальном числе случаев. В этом поможет компилятор, ведь всё, что не проверяется компиляцией, может быть использовано неправильно.
 - a. Например, конструктор вместо метода init().
- G. Рефакторинг может привести к деградации абстракции. Следите за этим, когда гонитесь за скоростью и удобством! Могут пострадать хорошие абстракции.

Инкапсуляция

Обычно понимают “скрытие реализации”, Но это, вообще говоря, свойство кода, относящегося к одной задаче, лежать рядом. Ну, для ООП верно всё предыдущее предложение (*обеспечивается классами*).

Инкапсуляция -- это механизм защиты инвариантов объекта. =>

1. “Меньше знаешь -- крепче спишь” нужно реализовывать с умом, то есть не делать классы с одним методом.
2. public-полей не бывает. Геттеры/сеттеры вообще могут выручить, когда объект для хранения станет полноценным объектом с поведением.

Рекомендации относительно инкапсуляции:

- A. Объекты не должны знать о других объектах. То есть не нужно делать предположений, которые влияют на реализацию абстракции. Например, “тут не будет проверки на null, т.к. тот объект не вызывает меня с null”
- B. Читабельность **>>** писабельности.
- C. Опасаться программирования через интерфейс, *выдвижении предположений о том, как работает абстракция*. То есть семантических нарушений инкапсуляции. Не должно быть: “не вызываю метод ConnectToBD, т.к. он вызовется в методе GetRow”.
- D. Protected, *internal* поля тоже запрещены. Предок не может контролировать потомка. Да и в сборке нет гарантии, что инварианты не сломаются (не вами,

вами с замутненным сознанием). Должно быть два полноценных контракта (конечно, поддерживающих инвариант):

- a. для потомков
- b. для внешних объектов

Наследование, полиморфизм

Агрегация, композиция **>>** наследования.

- - Да, при агрегации, композиции, нужно упаковывать вызовы внешнего кода в методы данного объекта,
+ но зато потомок не имеет лишней функциональности, не должен поддерживать инварианты предка.
- + Также для изменения поведения программы при наследовании нужно перекомпилировать её. При агр., комп., программа может меняться во время выполнения.
- Агр., комп. не реализует полиморфизм (если только сам объект не полиморфен)

Т.о. наследование нужно для обеспечения полиморфизма через наследование от интерфейсов.

???

Но наследование юзается в паттерне “Шаблонный метод” ???

А также в оконных библиотеках, но лишь потому, что раньше не было лямбда-функций. И поэтому наследование было единственным вариантом.

- Корректность наследования можно проверить через принцип Барбары Лисков.

* Существуют ключевые слова для запрещения наследования (final, sealed).

Рекомендации для наследования:

1. Не делать базовый класс с одним потомком “на будущее расширение”. Если что, можно будет легко отрефакторить
2. Множественное наследование плохо. Больше 7 -- ужасно. 3-4 -- подозрительно (исключение, абстрактное синтаксическое дерево компилятора).
3. Лишние переопределяемые методы, которые потомок не может содержательно переопределить.
4. Новые методы с теми же названиями, что и у родителя. Возможно, потомок -- это вообще отдельная сущность??

Фишечка: если есть switch, цепочка if-else, то это индикатор необходимости виртуальных методов. Нужный метод вызывает в рантайме. Это облегчает расширение.

Например, алгебраические операции.

```

abstract class Operation {
    private int left;
    private int right;

    protected int getLeft() { return left; }
    protected int getRight() { return right; }
    abstract public int eval();
}

class Plus extends Operation {
    @Override public int eval() {
        return getLeft() + getRight();
    }
}

class Minus extends Operation {
    @Override public int eval() {
        return getLeft() - getRight();
    }
}

```

Вопрос инициализации

1. Инициализировать всё, что можно инициализировать. Использовать для этого конструктор. Не ждать, что юзер вызовет `init()`.
2. Не вызывать виртуальные методы в конструкторе, т.к. сначала идёт инициализация предка, в которой может быть вызван метод ещё не инициализированного своими инвариантами потомка.
3. Делать `private`-конструкторы для объектов, которые *не должны быть созданы или одиночек*. Есть ключевое слово `abstract` в Java и C#.
Protected конструкторы для абстрактных классов, чтобы не смущать пользователя, но дать возможность инициализироваться потомкам.
4. Глубокое копирование предпочтительнее поверхностного. Ибо юзер не ожидает того, что изменение одного объекта приводит к изменениям другого. Да, глубокое сложнее, особенно если нужно идентифицировать объекты.
Если идентифицировать не надо, то есть хак -- сериализовать объект в массив байт, затем десериализовать.
5. Одиночки юзать с осторожностью, ибо это глобальные переменные.
 - a. Проблемы мультипоточности.
 - b. Требует постоянного создания Mock-объектов при юнит-тестировании.

Мутабельность

Порождает проблему синхронизации в многопоточных приложениях.

При написании не мутабельного класса помнить:

- Не реализовывать методы изменения состояния. Если очень нужно, то эти методы должны возвращать копию объекта, а *не модифицировать состояние*.
- Делать все поля константными.
- Делать невозможным наследование от класса

- Не давать вовне ссылок на константные поля. Ссылку изменить нельзя, а вот объект по константной ссылке -- сколько угодно.

! все что может быть не мутабельным, должно быть не мутабельным.

Оптимизация

- а) Не оптимизировать, пока не получили абсолютно четкого работающего решения.
- б) Не доводить до пессимизации, когда пишется очевидно плохой код.

Общие рекомендации

01. “Failfast”: падать всегда и везде -- при разработке и на релизной версии.
 - а. Ловить всевозможные ошибки.
 - б. “защитное программирование” -- не доверяем внешнему миру полностью
02. Документация открытого API. Документация внутренней работы для других программистов.
 - а. не забывать про: постусловия, предусловия, бросаемые исключения, потокобезопасность, ассипмптотику
03. Использовать проверки на этапе компиляции по максимуму. То есть во всю юзать систему типов. Проверяться в рантайме по минимуму.
04. Continuous Integration, воспроизводимые сборки, запускающие юнит-тестов при каждом коммите в репозиторий.
05. Написание как можно большего числа unit-тестов с интересными случаями
06. Не боятся все переписать. Код хоть и выбрасывается, но знания о предметной области остаются.

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

Моделирование

Модель -- упрощенное представление объекта, абстрагирование от его сложности и описание существенных свойств. Модель помогает донести информацию, поэтому важно строить модель для каждого конкретного случая по своему, а не тупо фигачить диаграммы. Также нужно контролировать детализацию модели.

“Все модели ошибочны, некоторые из них полезны”: то есть модель не описывает объект полностью, она описывает существенные его свойства.

Моделирование в ПО. Помогает проанализировать продукт ещё до написания строчки кода и даже протестировать его, а также понять. Часто полезны неформальные модели, обсуждающиеся в ходе работы над проектом у доски. Формальные, например, UML, важны для документирования ПО.

! Модели ПО моделируют не только само ПО, но и окружение, в котором оно создается.

Архитектура -- набор основных решений, принятых для данной системы.

Архитектурная модель -- некоторый *артефакт*, отражающие некоторые или все эти решения.

1 арихтектура ----- [*] архитекторных моделей.

Архитектурное моделирование -- процесс уточнения и документирования архитектурных решений.

Для избежания “архитектурного паралича” нужно думать:

1. Какие архитектурные решения нуждаются в моделировании
2. Насколько подробно
3. И как формально

Моделировать только важное (для клиентов). Цена за архитектурные решения НЕ должна превышать их преимуществ:

- Облегчают понимание и проектирование системы. Помогают увидеть путь развития архитектуры и явные ошибки
- Документирование архитектуры при архитектурном описании. Помогает понимать проект другим разработчикам.
- Способ коммуникации между разработчиками
- Легкое донесение информации заказчику по сравнению с идеями, выраженными в словах или коде
- Исходники для генерации кода (не рекомендуется использовать, т.к. все равно придется переписывать, ибо модель не полна на 100%, либо как минимум нужно подвести под стайлгайд)

Виды моделей (+ визуальные модели)

- A. Естественные языки
 - a. Максимально гибок, выразителен, *не требует специальных знаний*
 - b. Нельзя автоматически анализировать, многословен, *не строг*.
- B. Неформальные графические модели
 - a. Наглядны, не требуют специальных знаний, красивые
 - b. Неформальны, нельзя автоматически анализировать, *нестрогость, неоднозначность, воспринимаются специалистами как формальные*
- C. Формальные графические модели
 - a. Стандартизированы, формальны
 - b. Мало строгой семантики, хоть и есть строгий синтаксис
- D. Формальные текстовые языки
 - a. Возможен анализ *утверждений о системе* на уровне модели, *формальность, строгость*
 - i. Для областей с требованием высокой надежности и безопасности: встроенные системы, системы реального

времени, программно-аппаратные системы, аппаратные системы

b. Многословность, сложность обучения

Пример -- AADL (Architecture Analysis & Design Language)

Чаще всего требования описываются словесно, неформально или полуформально (на-

пример, через “user story” в Scrum и других Agile-методологиях). Есть полуформальные

визуальные языки описания требований (например, диаграмма случаев использования

UML, но она требует и текстовых описаний тоже), есть более-менее формальные модели, например, диаграмма требований SysML.

Метафора визуализации

Метафора визуализации -- договоренность о том, как будут представляться сущности в модели. В UML классы -- это прямоугольники. В нотации Буча: классы -- облака с пунктирной линией, объекты -- облака сплошной линии

И ещё пара терминов

Точка зрения на систему -- это то, какие аспекты системы мы должны моделировать и для кого это нужно. Для разработчиков (что делать-то?), менеджеров (что уже сделано), заказчика (сколько сделанное стоит)? Для всех своя модель.

Семантический разрыв -- неспособность модели полностью специфицировать моделируемую систему; разрыв между информацией в модели и информацией в программе реализующей модель, которую компьютер может исполнить.

Часто используются одноразовые модели, которые даже нельзя понять без помощи их авторов.

P.S. существуют исполнимые графические модели -- это графические языки.

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

Язык UML

Unified modeling language -- набор визуальных языков (14) моделирования ПО , объединенных единым стандартом и единым описанием синтаксиса и описывающих систему с разных точек зрения.

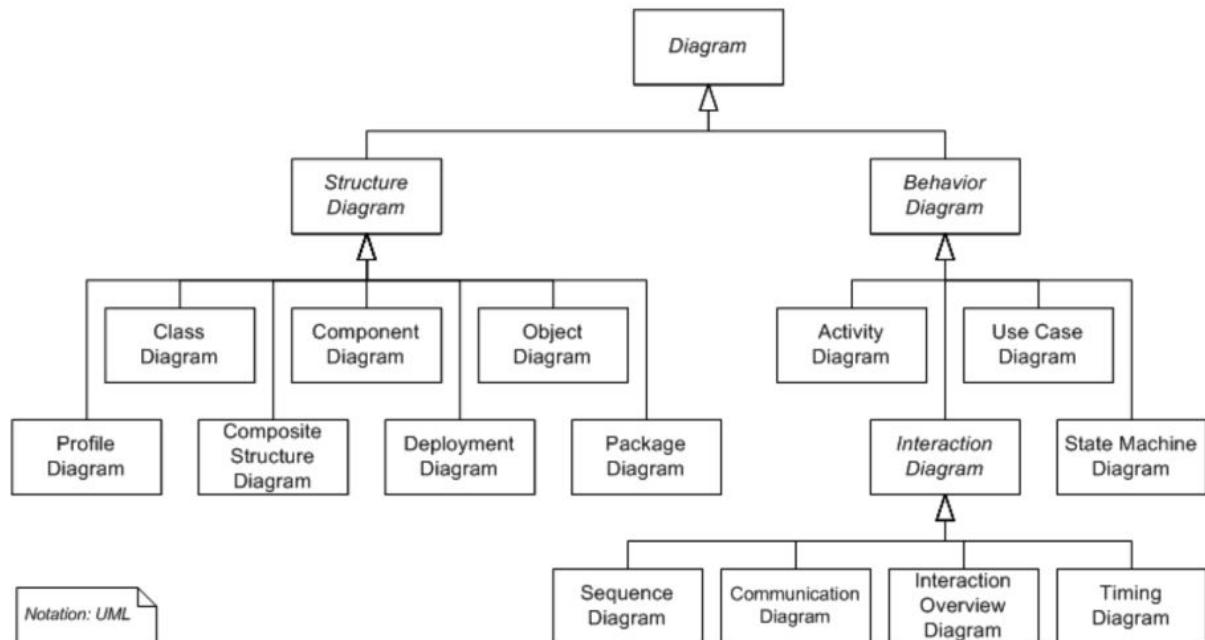
UML используется при проектировании структуры системы: поясняются, уточняются и документируются архитектурные решения.

Два механизма расширений UML:

- профили -- уточнение UML для предметной области
- метамоделирование -- редактирование UML стандарта путем добавления и удаления элементов

Виды языков UML делятся на два типа:

- структурные -- структура системы времени компиляции
- поведенческие -- поведение системы во время выполнения



UML-диаграммы допускают опускать информацию, например:

- атрибуты (поля)
- операции (методы)
- ассоциации
- ??? Что-то кроме диаграмм классов

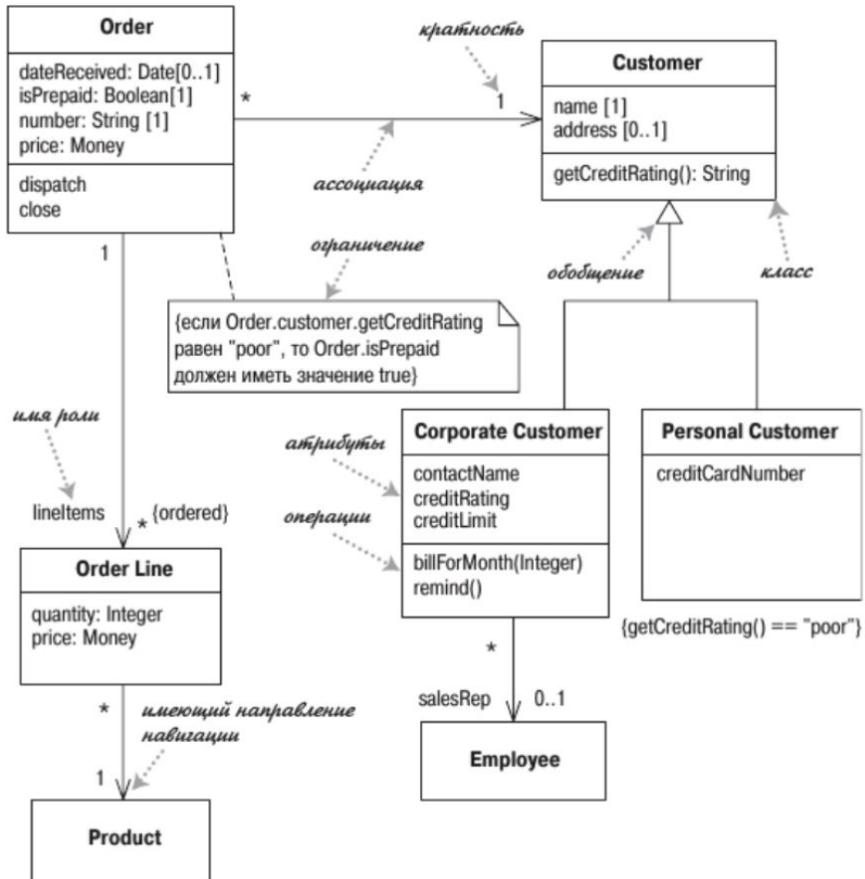
Самая популярная диаграмма -- диаграмма классов.

Проектирование структуры системы

См. билет с моделированием.

Диаграммы классов

Показывают структуру системы в терминах классов, интерфейсов и отношений между ними.



Атрибуты семантически равны ассоциациям.

Атрибуты используются, когда поле --

- это стандартный тип-значение
- тип из сторонней библиотеки

Ассоциации -- когда нужно показать связи данного архитектурной модели (когда поля должны быть типами создаваемых нами типов)

ЗАМЕЧАНИЕ: ассоциация может иметь класс-ассоциацию. Класс-ассоциация дает возможность определить дополнительное ограничение, согласно которому двум участвующим в ассоциации объектам может соответствовать только один экземпляр класс-ассоциации. Пару примеров:

Посмотрим на две диаграммы, изображенные на рис. 5.14. Форма этих диаграмм практически одинакова. Хотя можно себе представить компанию (Company), играющую различные роли (Role) в одном и том же контракте (Contract), но трудно вообразить личность (Person), имеющую различные уровни компетенции в одном и том же навыке (Skill); действительно, скорее всего, это можно считать ошибкой.

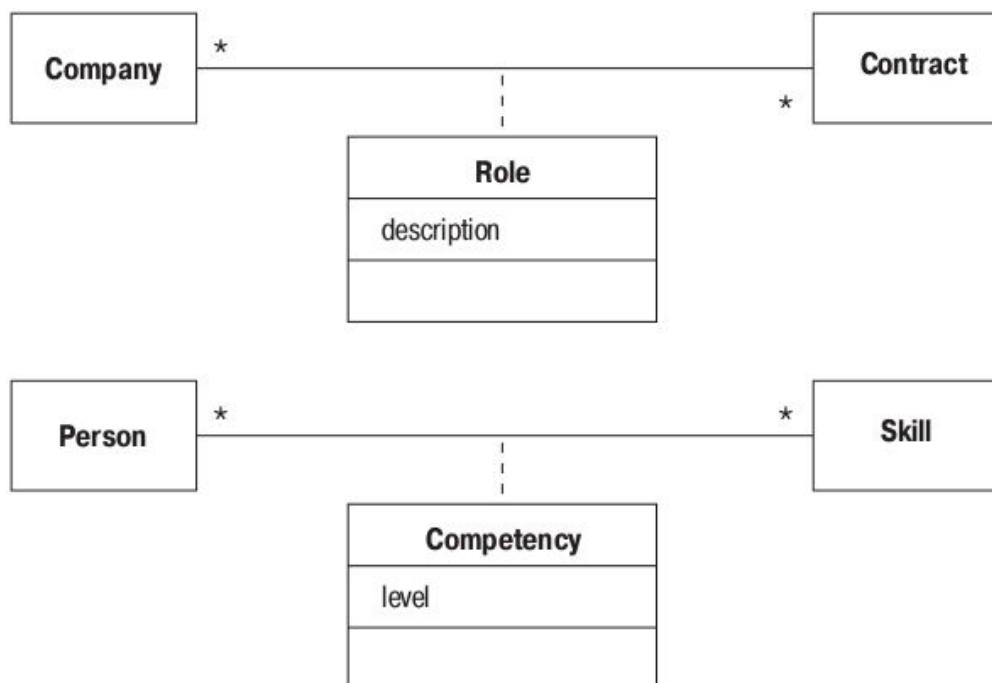


Рис. 5.14. Хитрости класса-ассоциации (класс Role, возможно, не должен быть классом-ассоциацией)

Диаграммы и исходный код двунаправленной ассоциации



Если изменился объект с одной стороны двунаправленной ассоциации, то нужно проинформировать об этом объект на другом конце.

```

class Car {
    public Person Owner {
        get { return _owner; }
        set {
            if (_owner != null)
            {
                _owner.friendCars().Remove(this);
            }
            _owner = value;
            if (_owner != null)
            {
                _owner.friendCars().Add(this);
            }
        }
    }
    private Person _owner;
}

class Person {
    public IList Cars {
        get { return ArrayList.ReadOnly(_cars); }
    }

    public void AddCar(Car arg) {
        arg.Owner = this;
    }

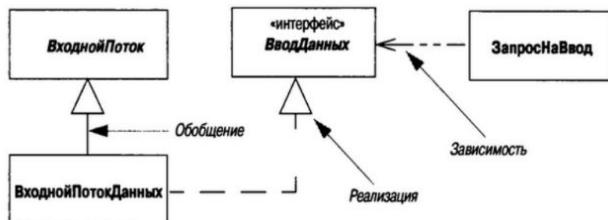
    private IList _cars = new ArrayList();
    internal IList friendCars() {
        // должен быть использован только Car.Owner
        return _cars;
    }
}
  
```

Интерфейсы

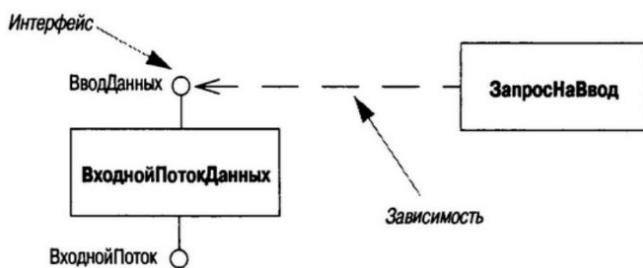
<<some word>> -- стереотип в UML.

Интерфейсы рисуются как:

- классы с <<interface>>, если важно упомянуть методы



- “в леденцовой нотации”, если интерфейсов много, либо не важны методы



Зависимости

Один класс что-то знает о другом. Пунктирная стрелка

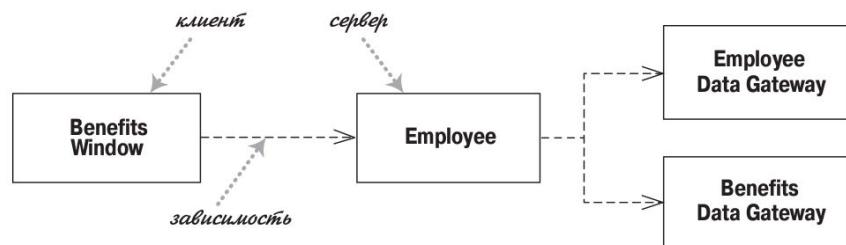


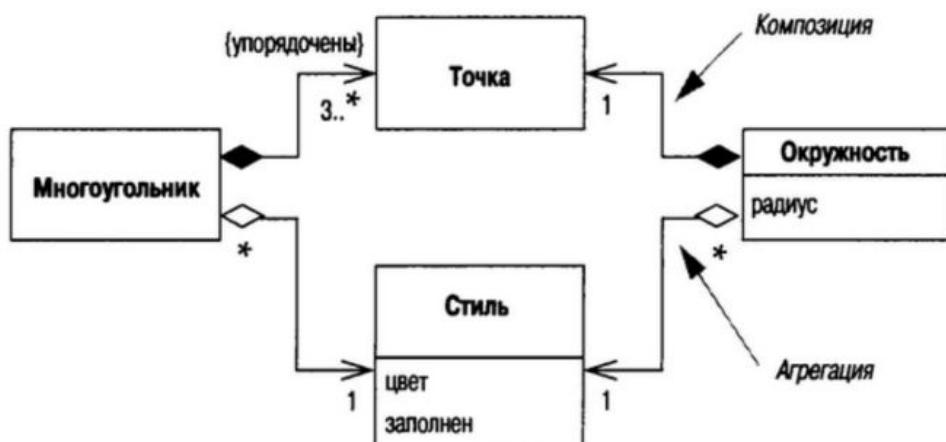
Рис. 3.7. Пример зависимостей

Виды зависимостей:

- call -- источник вызывает операцию цели
- create -- источник создает экземпляр цели
- derive -- в источнике значение считается из цели
- refine -- уточнение. Возможно даже между разными диаграммами (семантическими уровнями) -- предметная область и реализация
- trace -- влияние. Возможно даже между разными семантическими уровнями -- случай использования-реализующие классы
- instantiate -- источник является экземпляром цели (источник класс, значит, целевой класс -- это метакласс [класс, экземпляры которого тоже классы])
- realize -- источник является реализацией цели
- use -- для реализации источника требуется цель
- substitute -- источник может быть заменен целью

Агрегация и композиция

Разница -- время жизни и владения объектом.



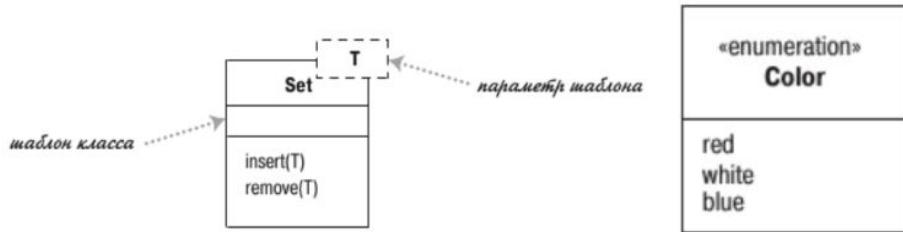
© М.Ф.

Пример:

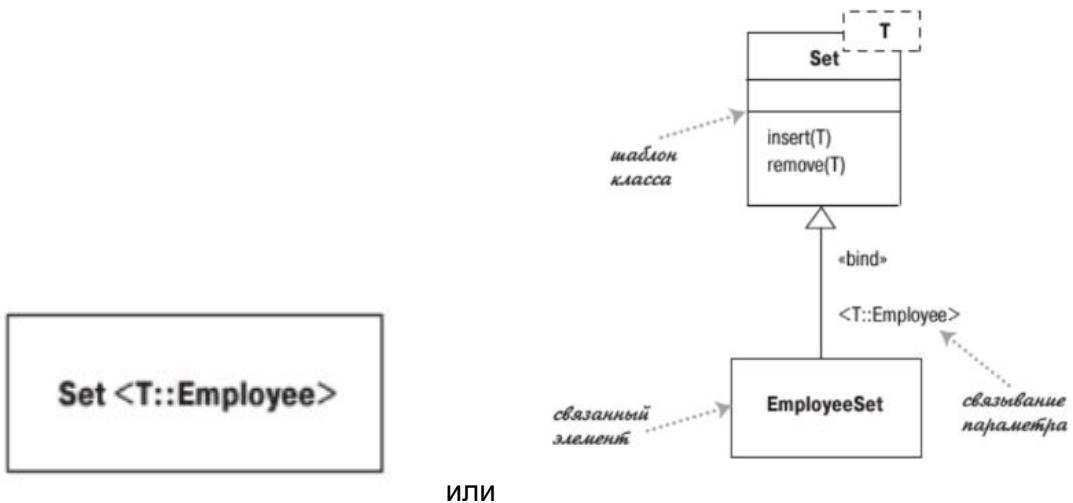
Многоугольник и окружность владеют своими точками, но не владеют стилями — у каждой геометрической фигуры должен быть стиль, но стили существуют независимо и могут переиспользоваться между фигурами. Точки между фигурами переиспользованы быть не могут.

Шаблоны и перечисления

Синтаксис:



Инстанцирование шаблона:



ИЛИ

Во 2ом случае -- хотим добавить инфу о новых методах.

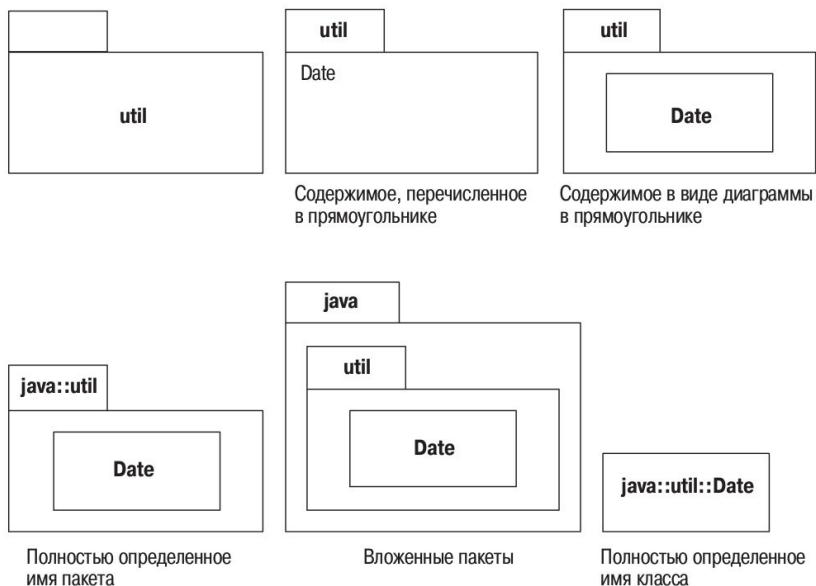
11. Диаграммы объектов, диаграммы пакетов UML.

Диаграмма пакетов

Пакет в UML -- объединение любой UML-конструкции.

Пакеты иерархичны, разбиваются на подпакеты и классы.

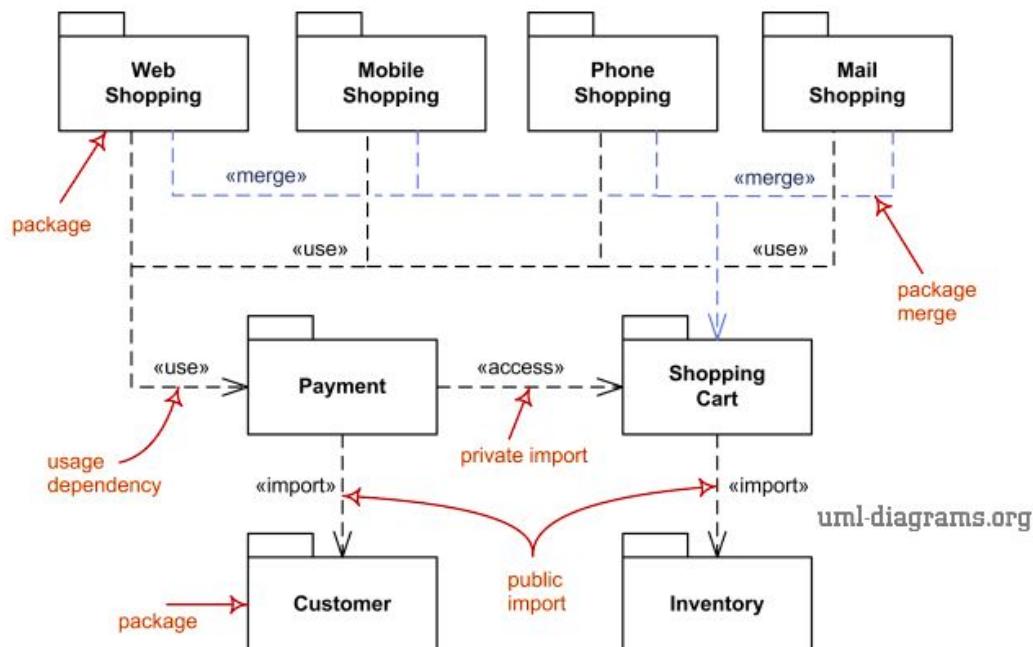
Пример, пространства имён в .NET.



Можно даже **диаграмму классов внутри** пакета

Полностью определенное имя -- идентификатор класса.

Для пакетов также доступны зависимости.



- Merge -- Слияние пакетов похоже на обобщение в том смысле, что исходный элемент концептуально добавляет характеристики целевого элемента к своим собственным характеристикам, в результате чего получается элемент, сочетающий в себе характеристики обоих.
 - access -- If the package import is **public**, the imported elements will be added to the namespace and made visible outside the namespace, while if it is **private** they will still be added to the namespace but without being visible outside (<https://www.uml-diagrams.org/package-import.html>).
 - import -- If the package import is **public**, the imported elements will be added to the namespace and made visible outside the namespace, while if it is **private** they will still be added to the namespace but without being visible outside.

Просто зависимость означает, что пакет как-то связан с другим пакетом (так же, как зависимость между классами), 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.

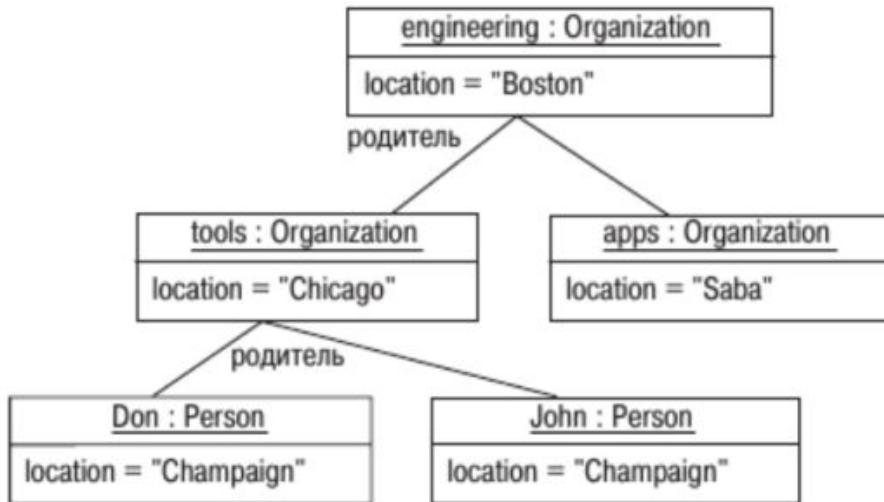
На самом деле, часто отношение реализации рисуют в том случае, если смысл пакета-предка в архитектуре — содержать интерфейсы или абстрактные классы, которые должны реализовывать другие пакеты.

Диаграмма объектов

Визуализация объектов в памяти в момент времени выполнения и *отношения между ними*.

Зачем?

- Поясняет диаграмму классов, которая, например, неявно скрывает в себе дерево.
- Погружение в предметную область перед созданием диаграммы классов



12. Диаграммы компонентов, диаграммы развертывания UML.

Диаграмма компонентов

Точного определения термина “компонент” не существует, но все интуитивно представляют, что это такое — нечто структурно связанное и больше, чем класс. Компонентами могут быть пакеты, пространства имён, сборки, .dll/.so-файлы, отдельные веб-сервисы в распределённом приложении и т.д. В общем, это именно то, из чего состоит высокоуровневая архитектура приложения.

Важно то, что компоненты представляют элементы, которые можно независимо друг от друга купить и обновить. В результате разделение системы на компоненты является в большей мере маркетинговым решением, чем техническим.

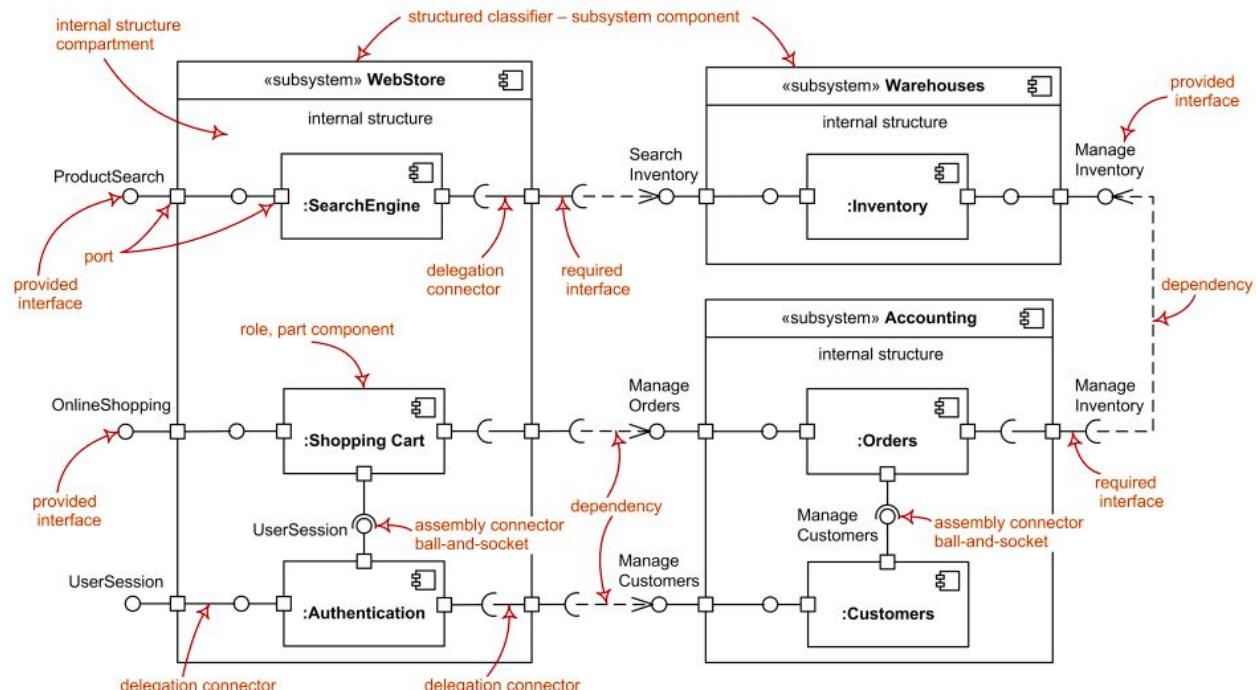
"Компоненты – это не технология. Технические специалисты считают их трудными для понимания. Компоненты – это скорее стиль отношения клиентов к программному обеспечению. Они хотят иметь возможность

покупать необходимое им программное обеспечение частями, а также иметь возможность обновлять его, как они обновляют свою стереосистему. Они хотят, чтобы новые компоненты работали так же, как и прежние, и обновлять их согласно своим планам, а не по указанию производителей. Они хотят, чтобы системы различных производителей могли работать вместе и были взаимозаменяемыми. Это очень разумные требования. Одна загвоздка: их трудно выполнить.

Ральф Джонсон (Ralph Johnson)"

Высокоуровневое представление системы.

Живут актуальными очень долго в процессе рефакторинга, т.к. их не нужно перерисовывать, как, например, диаграммы классов. Изменяются тогда, когда изменяется глобальная архитектура проекта.



The major elements of UML component diagram - component, provided interface, required interface, port, connectors.

Видно, что компоненты могут быть вложенными друг в друга, что они могут иметь интерфейсы (так же, как и классы, но на диаграммах компонентов чаще всего используется

только “леденцовую” нотацию). У компонентов есть порты, которые могут предоставлять или потреблять интерфейсы, но порты часто не рисуются, а подразумеваются, поскольку

обычно порт имеет только один интерфейс. Порты бывают полезны для изображения дег

легирования — что компонент просто перенаправляет запросы вложенному компоненту.

Слова `<<subsystem>>` и “internal structure” опциональны (и обычно не пишутся).

component diagram:

provided interface -- реализованный интерфейс,
required interface -- требуемый интерфейс,

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

The **provided interfaces** of a port describe requests to the classifier that other classifiers may make through this port. The **required interfaces** of a port describe the requests that may be made from the classifier to its environment through the port.

port,

A port may specify the services an encapsulated classifier provides to its environment as well as the services that an encapsulated classifier requires of its environment. Any port is service port by default, which is specified by default true value of isService attribute of the port.
(more -- <https://www.uml-diagrams.org/port.html>)

connector,

Connector is feature which specifies a link that enables communication between two or more instances playing some roles within a ???structured classifie???

delegation connector -- связь извне вовнутрь компонента.

assembly connector -- связь внутри компонента

Связи между компонентами тут означают зависимости по сборке, каждый компонент — это отдельный проект, который собирается в отдельный .dll/.so-файл.

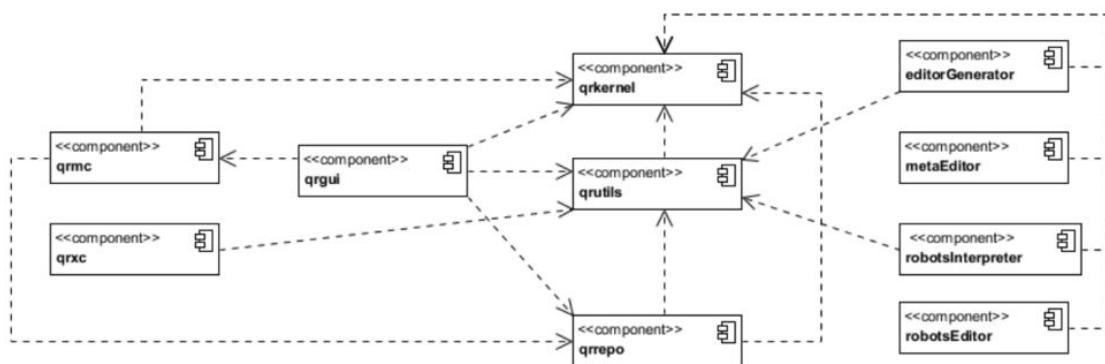


Диаграмма развертывания

Показывают на каком физическом оборудовании запускается та или иная часть системы.

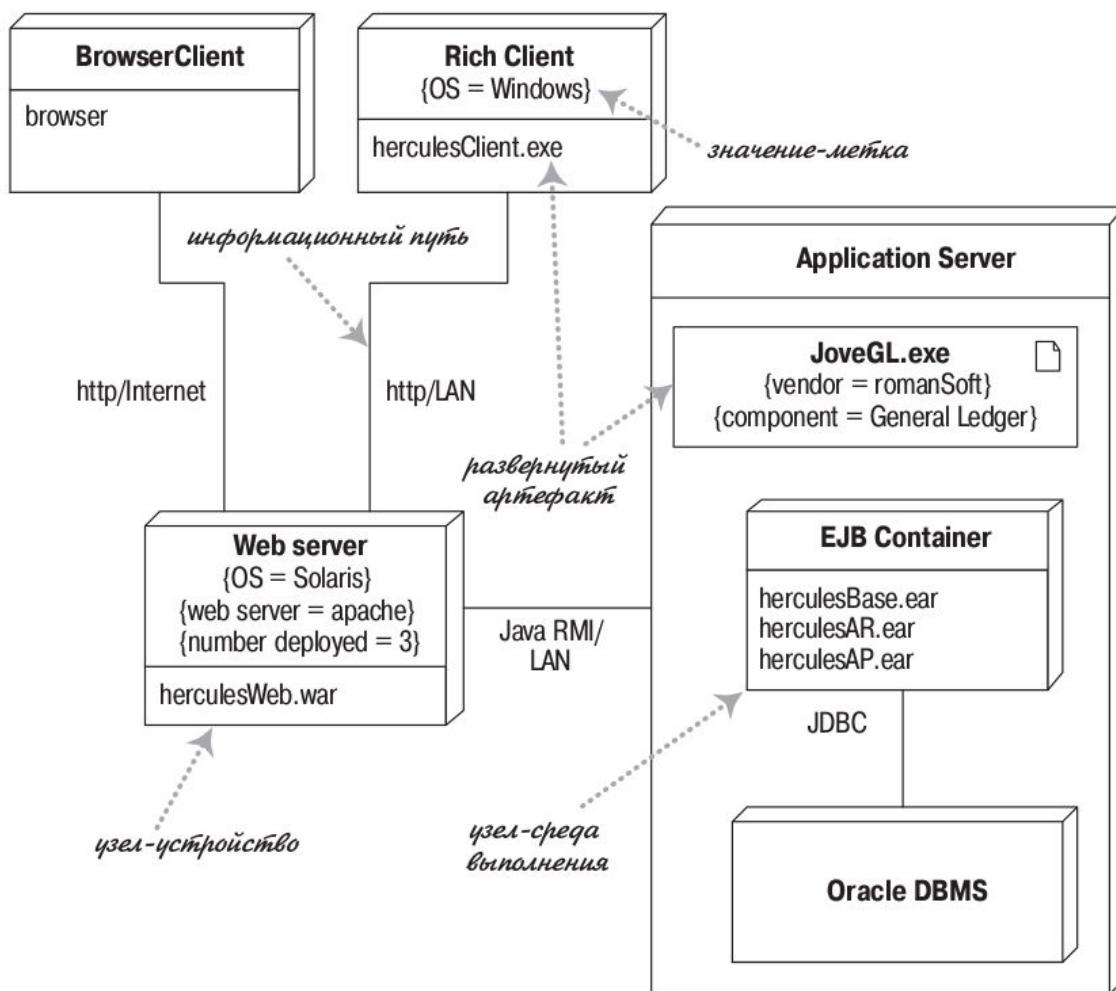


Рис. 8.1. Пример диаграммы развертывания

На рис. 8.1 показан пример простой диаграммы развертывания. Главными элементами диаграммы являются узлы, связанные информационными путями. **Узел** (node) – это то, что может содержать программное обеспечение. Узлы бывают двух типов. **Устройство** (device) – это физическое оборудование: компьютер или устройство, связанное с системой. **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. Такими файлами могут быть исполняемые файлы (такие как файлы *.exe*, двоичные файлы, файлы DLL, файлы JAR, сборки или сценарии) или файлы данных, конфигурационные файлы, HTML-документы и т. д. Перечень артефактов внутри узла указывает на то, что на данном узле артефакт разворачивается в запускаемую систему.

Артефакты можно изображать в виде прямоугольников классов или перечислять их имена внутри узла. Если вы показываете эти элементы в виде прямоугольников классов, то можете добавить значок документа или ключевое слово «*artifact*». Можно сопровождать узлы или артефакты значениями в виде меток, чтобы указать различную интересную информацию об узле, например поставщика, операционную систему, местоположение – в общем, все, что придет вам в голову.

Часто у вас будет множество физических узлов для решения одной и той же логической задачи. Можно отобразить этот факт, нарисовав множество прямоугольников узлов или поставив число в виде значения-метки. На рис. 8.1 я обозначил три физических веб-сервера с помощью метки *number deployed* (количество развернутых), но это не стандартная метка.

Артефакты часто являются реализацией компонентов. Это можно показать, задав значения-метки внутри прямоугольников артефактов.

Информационные пути между узлами представляют обмен информацией в системе. Можно сопровождать эти пути информацией об используемых информационных протоколах.

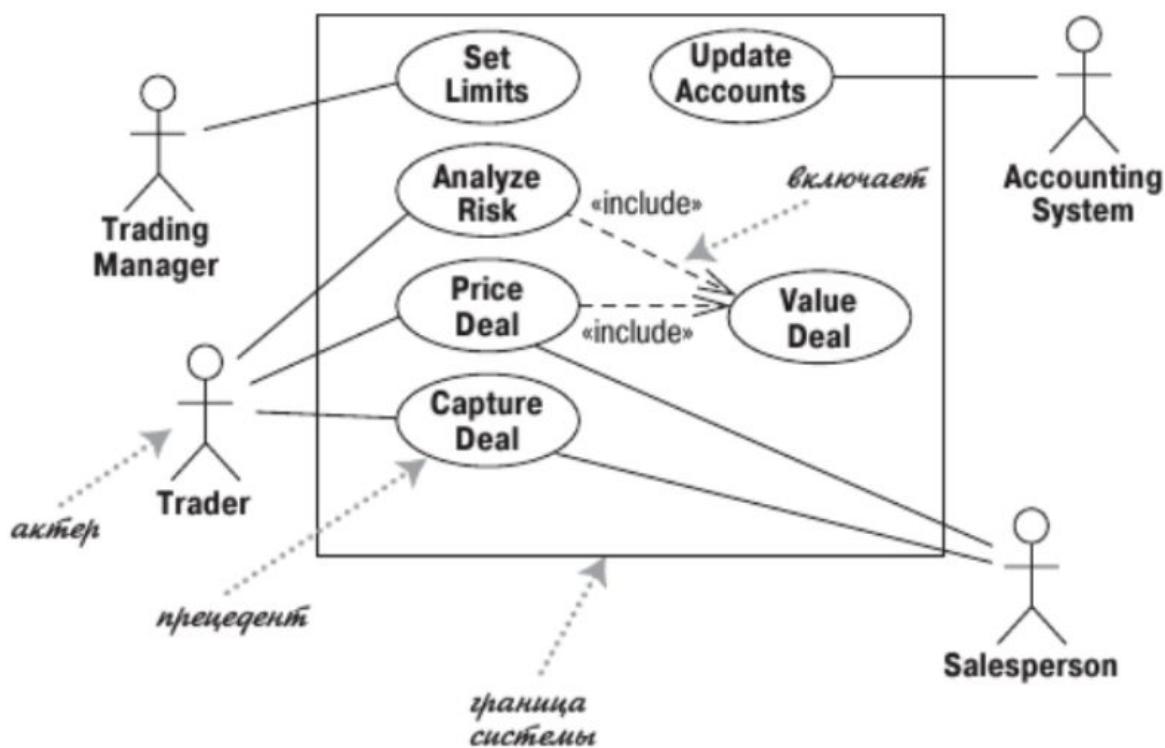
Ещё одно полезное применение диаграммы развёртывания — обратное проектирование, когда система у заказчика уже есть, но никто не знает, как она устроена, а вам надо для неё чего-то написать. Как правило, даже если документация по коду совсем не сохранилась, у админов есть документация по сопровождению, или они просто знают, какую машину надо перезапустить, если повис такой-то сервис. Если отобразить эти знания на диаграмме развёртывания, получим первое приближение высокоуровневой архитектуры системы.

13. Диаграмма случаев использования UML.

/диаграмма прецедентов/use case диаграмма

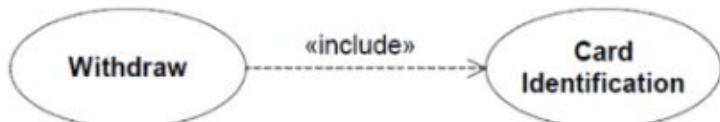
Имеет *границу системы* и 2 сущности:

- Акторы (клиенты -- люди, использующее нас ПО)
- Случаи использования (прецедент) -- цель акторов. НЕ функциональность системы, а цель использования системы. "Перевести деньги", а не "залогиниться в банке".
 - Это 2-3 слова, поэтому нужны сценарии использования -- описание последовательности действий для реализации прецедента
 - Часто неформальный язык
 - user story из Scrum
 - Диаграммы активностей UML



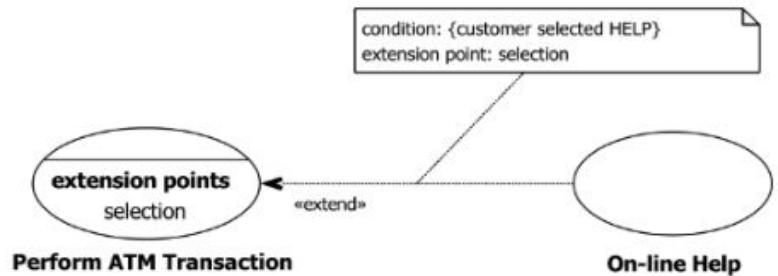
Прецеденты могут иметь отношения:

- *include* -- источник включает в себя цель



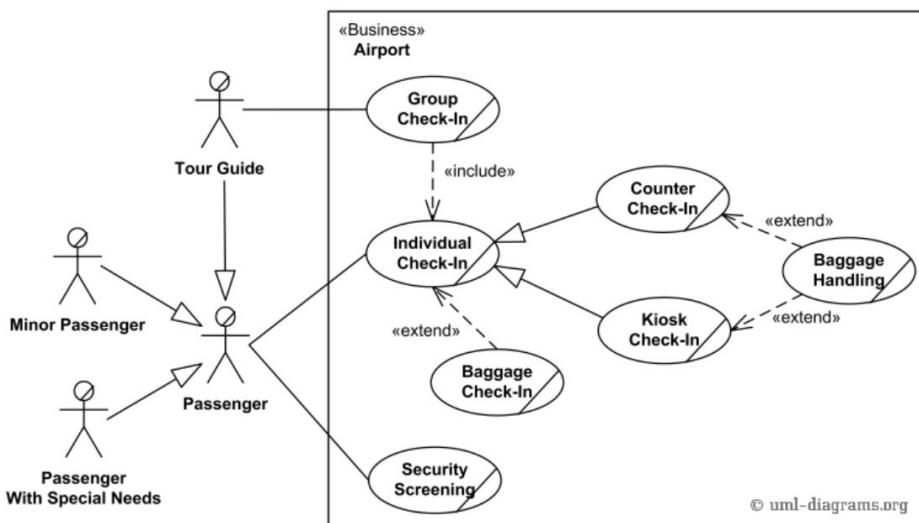
© OMG, UML 2.5 Specification

- *extend* -- источник расширяет цель



© OMG, UML 2.5 Specification

Пример сложнее:



Сценарии использования

Use Case Name: Request a chemical	ID: UC-2	Priority: High
Actor: Lawn Chemical Applicator (LCA)		
Description: The Lawn Chemical Applicator (LCA) specifies the lawn chemical needed for a job by entering its name or ID number. The system satisfies the request by reserving the quantity requested or the quantity available and notifying the Chemical Supply Warehouse of the pick-up.		
Trigger: A Lawn Chemical Applicator (LCA) needs a chemical for a job.		
Type: <input checked="" type="checkbox"/> External <input type="checkbox"/> Temporal		
Preconditions:		
1. The LCA identity is authenticated. 2. The LCA has necessary training and credentials on file. 3. The Chemical Supply datastore is up-to-date and on-line.		
Normal Course:		
1.0 Request a lawn chemical from the chemical supply warehouse. 1. The LCA specifies a chemical needed and the quantity needed 2. The system lists chemical and quantity on hand from Chemical Supply datastore a. If the quantity on hand is less than the quantity needed, the LCA specifies the quantity he will take b. Purchasing is notified of chemical shortage 3. The system gives the LCA a Chemical Pick-up Authorization for the quantity requested 4. The system notifies the Chemical Supply Warehouse of the chemical pick-up 5. The system stores the Lawn Chemical Request in the Chemical Request datastore		
Postconditions:		
1. The Lawn Chemical Request is stored in the Chemical Management System. 2. The Chemical Pick-up Authorization is produced for the LCA. 3. The Chemical Supply Warehouse is notified of the chemical pick-up. 4. Purchasing is notified of chemical outage.		
Exceptions:		
E1: Chemical is no longer approved for use (occurs at step 1) 1. The system displays message "That chemical is no longer approved for use" 2. The system asks the LCA if he wants to request another chemical or to exit 3a. The LCA asks to request another chemical 4a. The system starts Normal Course again 3b. The LCA asks to exit 4b. The system terminates the use case		

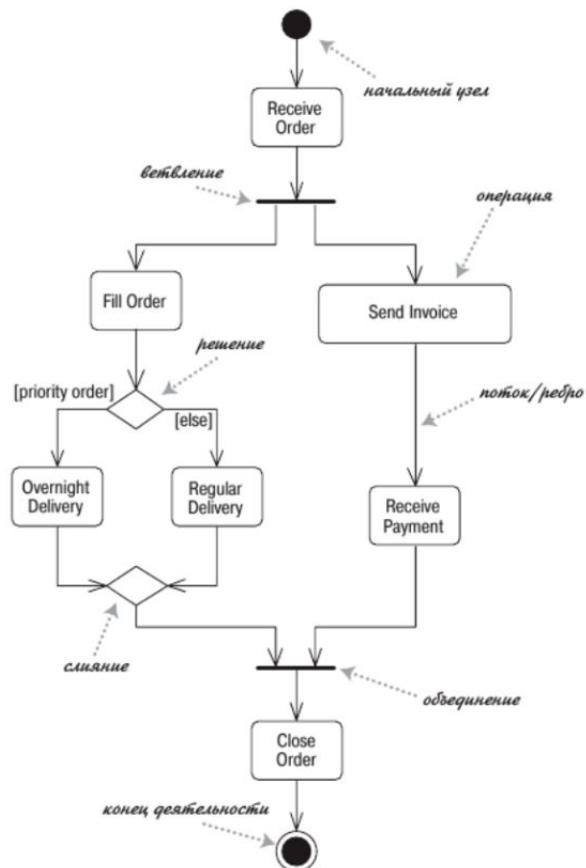
14. Диаграмма активностей UML.

Для моделирования бизнес-процессов. Бизнес-процесс -- любой процесс работы в организации.

Хороша для визуализации сценария использования с точки зрения пользователя.

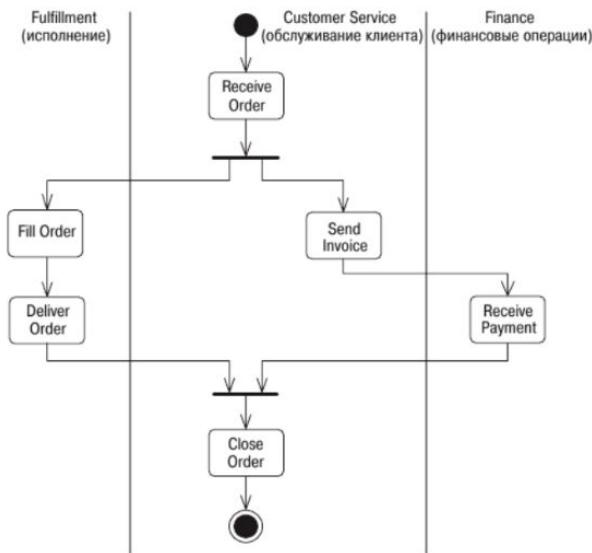
Имеет семантику исполнения (а в BPMN исполнители даже могут выполнять исполняемую семантику BPMN).

A) Синтаксис



- Решение -- взаимоисключающий switch/case
- Слияние
- Ветвление -- fork
- Объединение (ждём все потоки)
- Операция
- Поток исполнения
- Конец деятельности -- хоть один дошёл, завершаемся

Б) Разделение ответственности по отделам организации

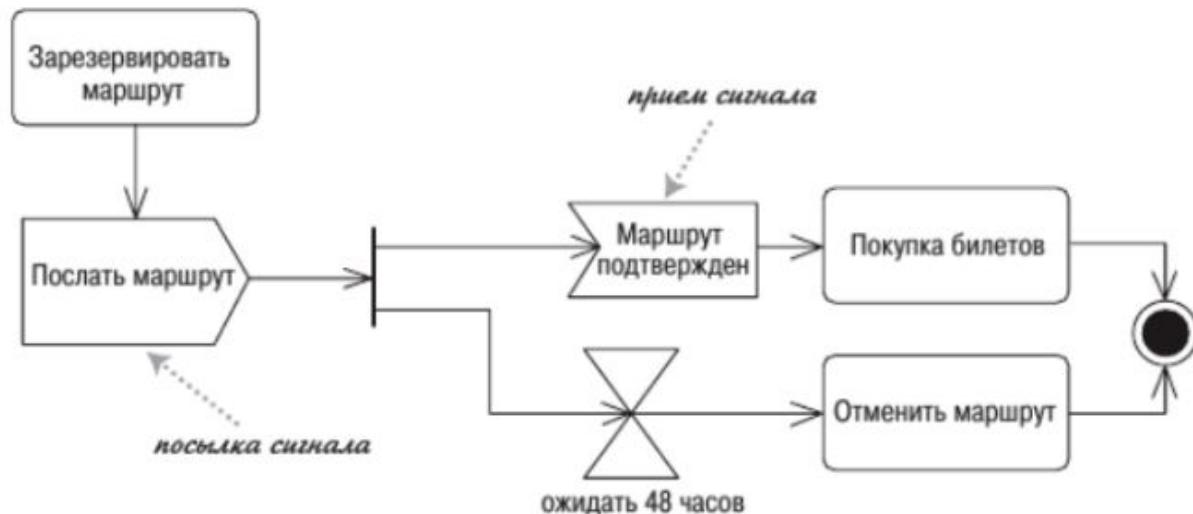


В) Асинхронное выполнение через посылку информации за границы моделируемого бизнес-процесса

После исполнения блока “посылка сигнала” управление возвращается (асинхронный вызов всё-таки).

“Прием сигнала” не исполняется, пока сигнал не придёт.

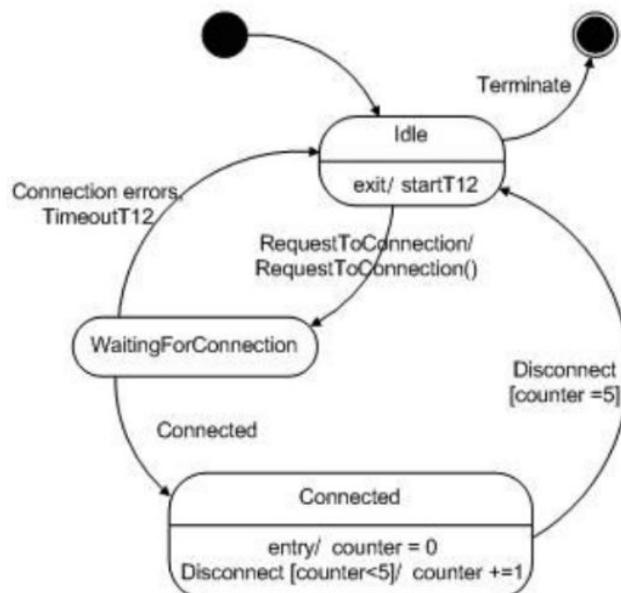
“Таймер” -- прием сигнала по умолчанию.



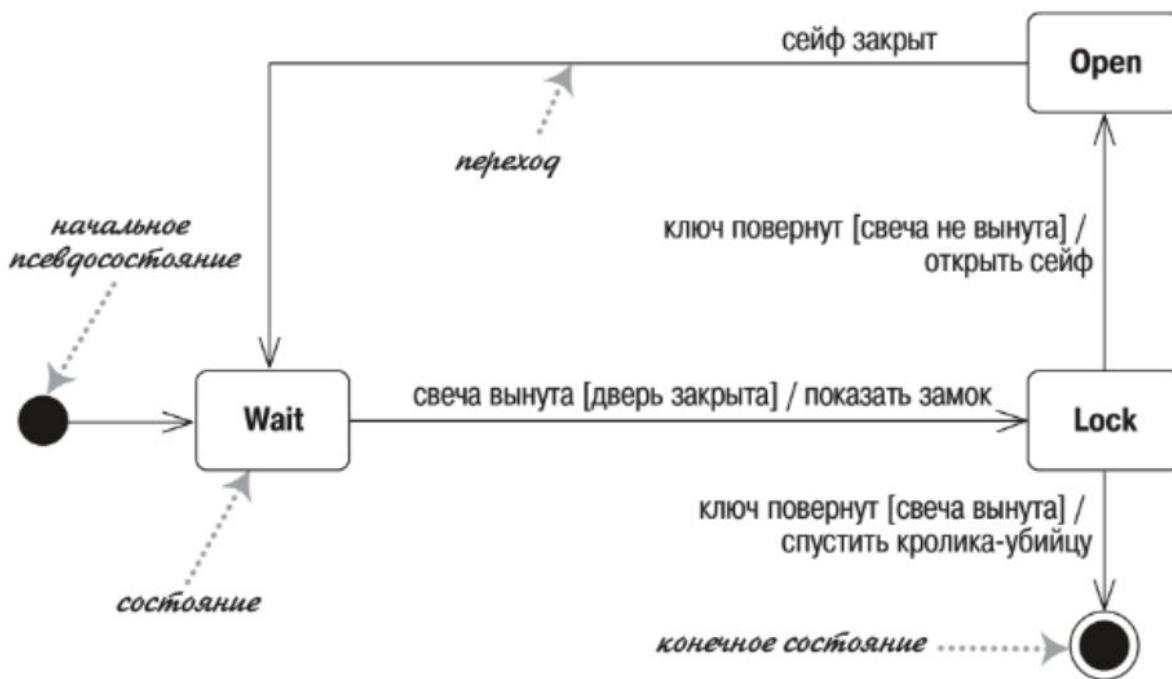
15. Диаграммы конечных автоматов UML.

/Диаграмма состояний
5.pdf

Одна из двух диаграмм, имеющих исполняемую семантику. Используется для моделирования реактивных систем, то есть систем, которые имеют состояния и при изменениях переходят из одного состояния в другое, возможно, ещё и делая полезную работу. Пример реактивной системы -- сеть с передачей пакетов (соединенено, соединение не установлено, передаются пакеты).



Синтаксис:

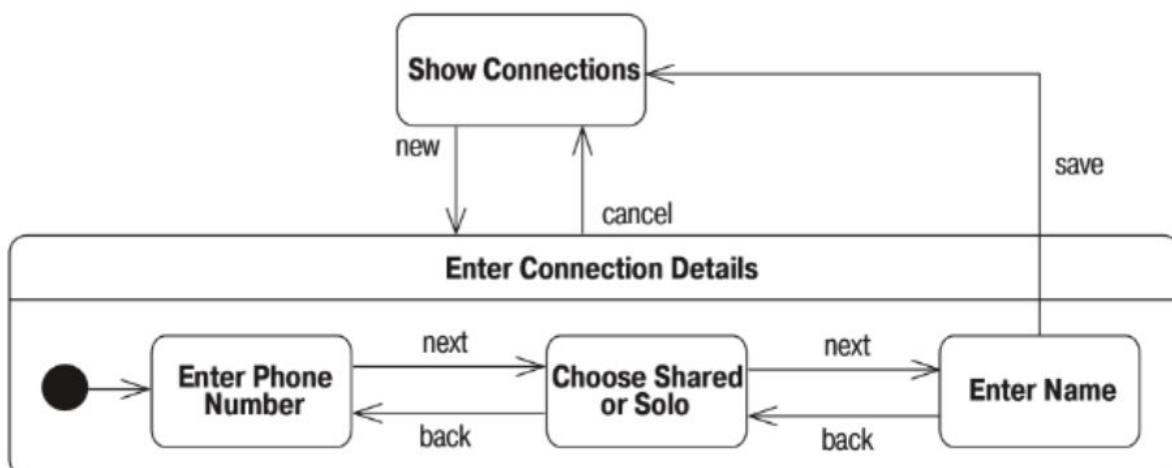


- Прямоугольники со скруглением -- состояние, имеющее имя и (опционально) действия, выполняемые в состоянии в зависимости от происходящих событий.
- На стрелках -- переходы между состояниями, имеют опционально стражников (guard) -- условие, при котором происходит изменение состояния. Над переходом пишется событие, которое инициирует переход. Условия взаимоисключающие. Недетерминированные автоматы считаются некорректными.
- [<trigger> ',' <trigger>]* '[' '<guard>''] [/<behavior-expression>]]
- Событие -- что-то внешнее, на что реагирует система
- Есть псеводостояния начала и конца (из начала переход мгновенный, в конец -- заканчивается выполнение).
- Entry Activity
- Exit Activity
- Do activity -- постоянно, пока мы в состоянии
- внутренний переход -- без срабатывания entry и exit (Может, конечно, быть полноценным переходом)

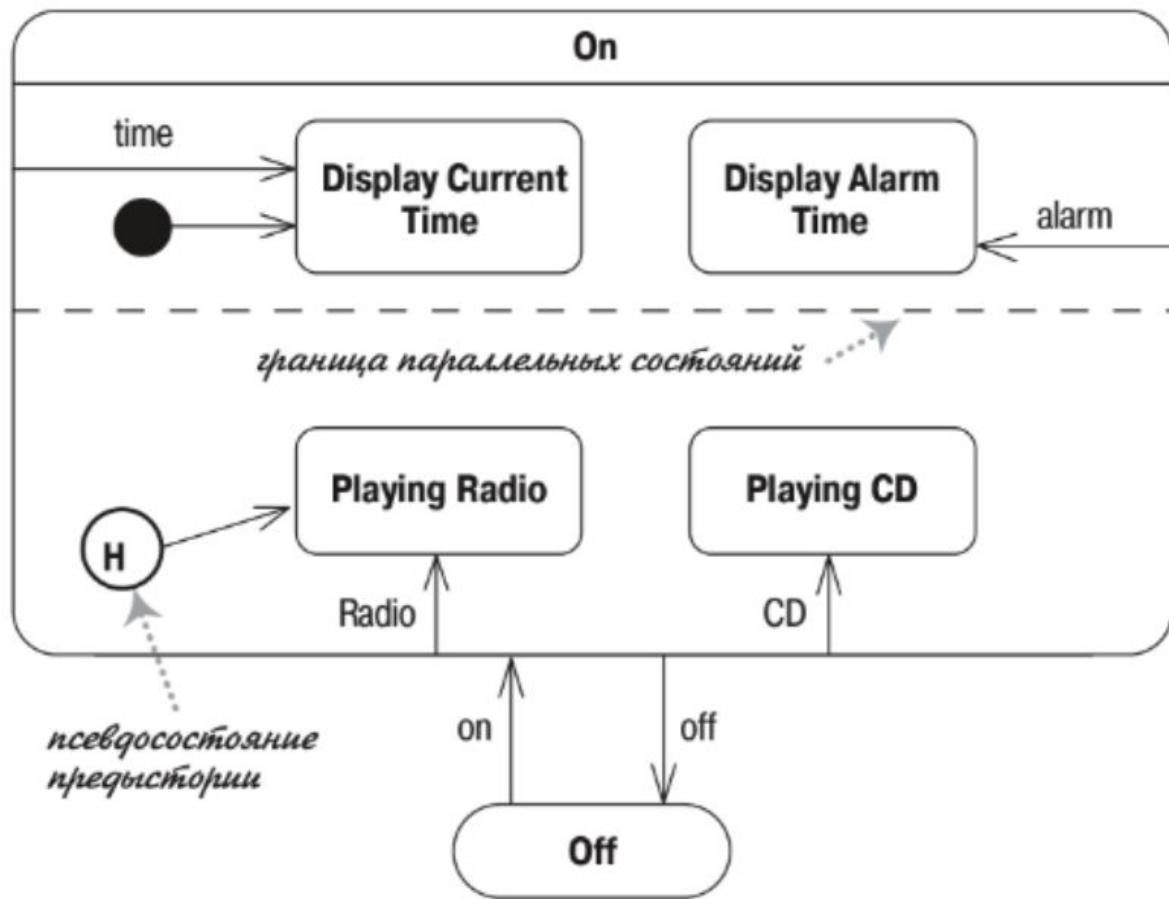
Семантические различия Активностей vs Конечных автоматов

- Активностей не останавливаются на действиях, указанных в диаграмме. Конечных автоматов -- на каждом состоянии объект находится какое-то время своей жизни
- Активности описывают один метод объекта. Состояний -- весь объект целиком (состояния объекта -- это его поля)
- Активности -- полезная работа в действиях, состояний -- полезная работа, обычно, в переходах

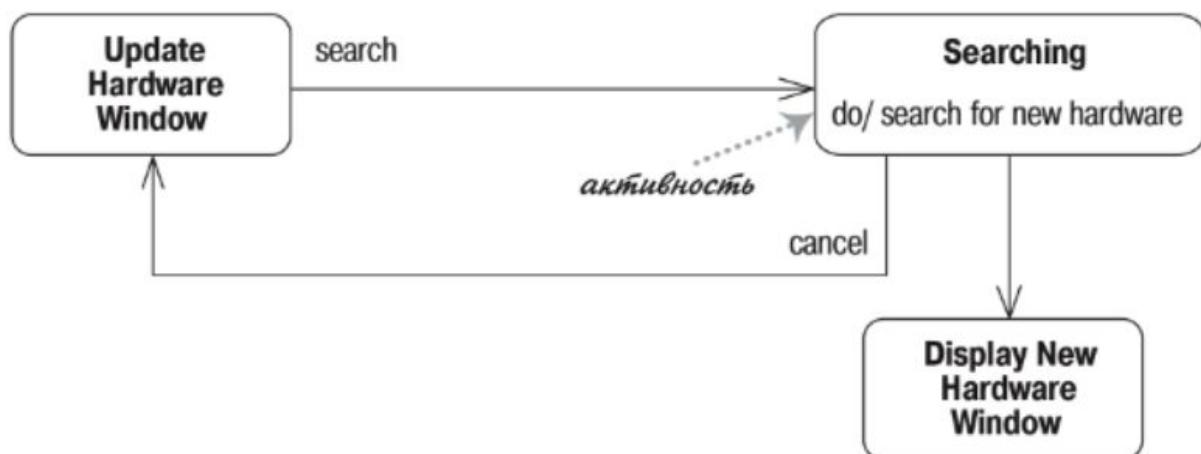
Пример перехода “из всех состояний” + вход/выход из объемлющего



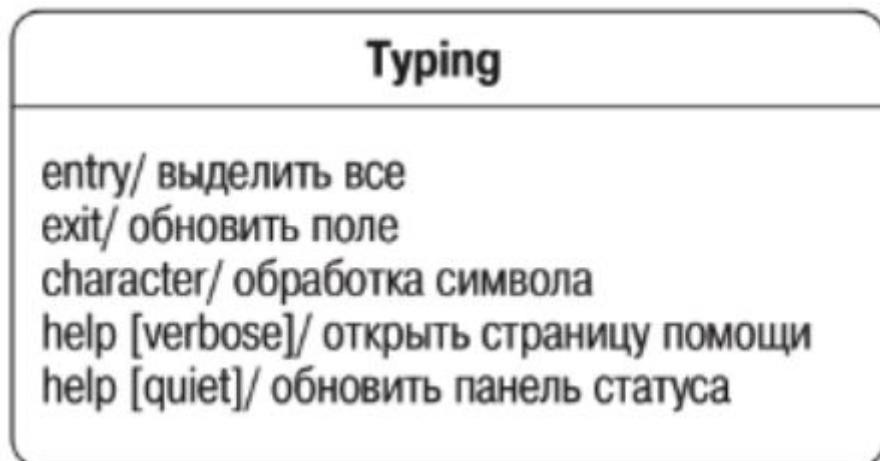
Псевдосостояние истории, параллельные состояния, переход из объемлющего во вложенные состояния



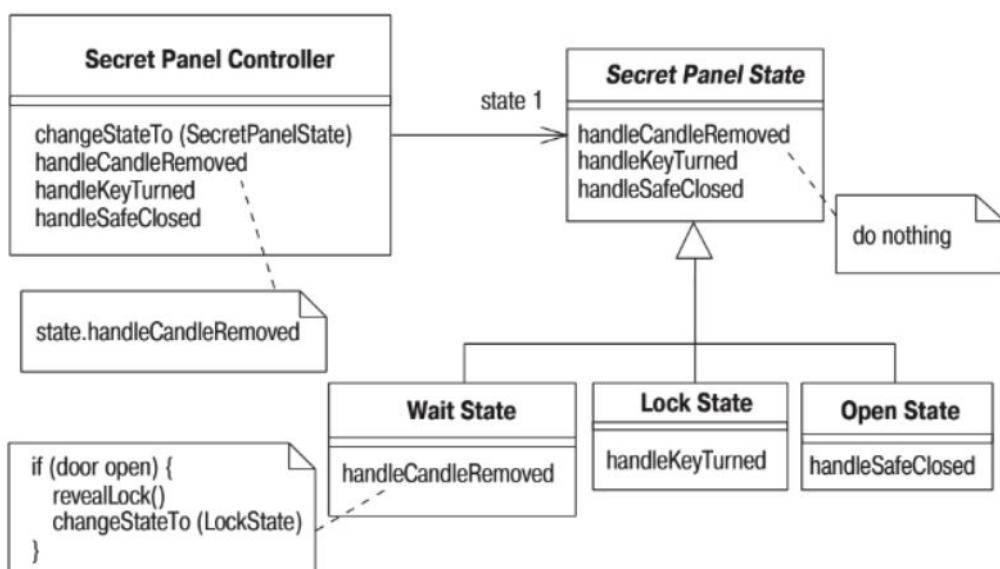
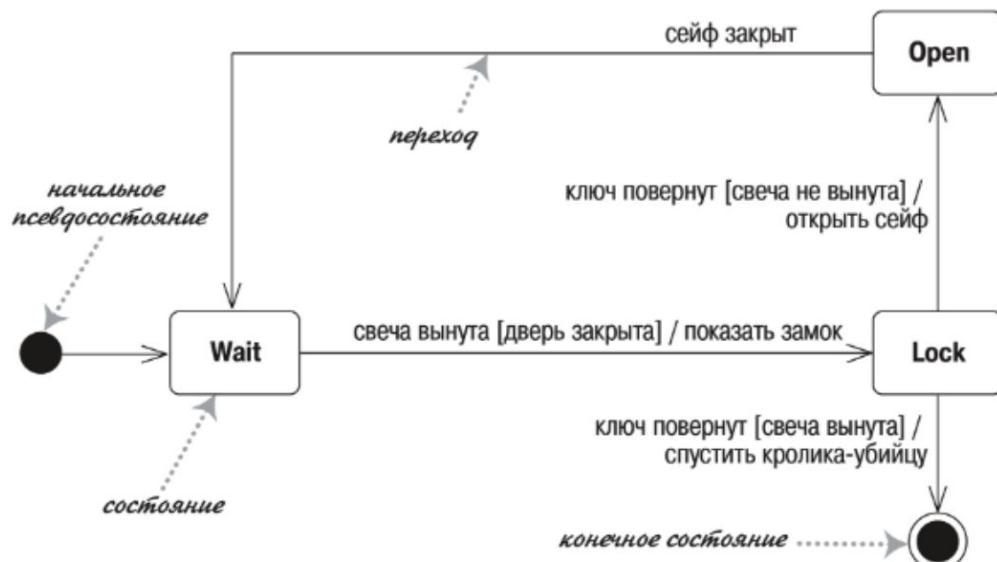
Активности внутри состояния



Внутренние переходы, exit/entry

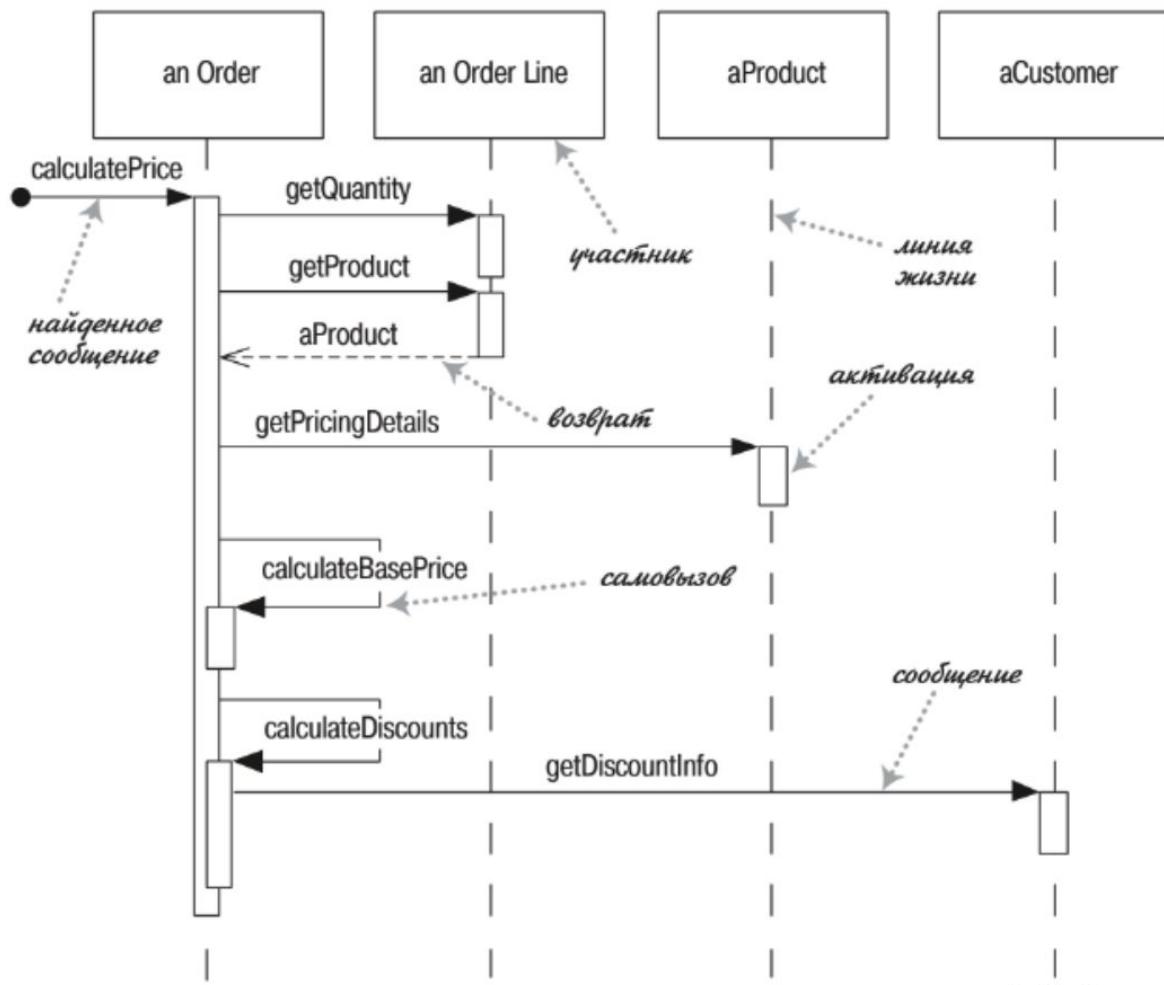


Реализация через паттерн “Состояние” >> гигантского switch



16. Диаграммы последовательностей UML.

Используются для визуализации взаимодействия между объектами -- передача сообщений, возврат значений, время жизни объекта.



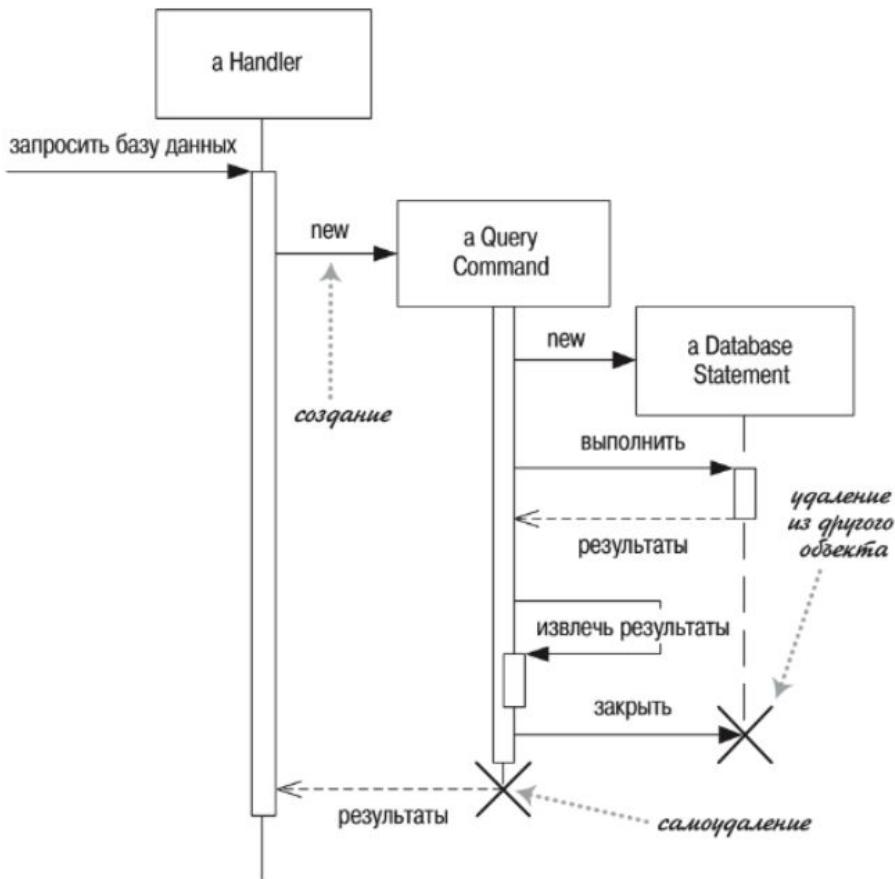
- Линия жизни -- когда объект в памяти.
- Активация -- когда на стеке вызовов лежит метод объекта.
- В сообщениях могут передаваться параметры (сами объекты тоже могут передаваться)

Внимание! Тут не класс, а **объекты**!

ГДЕ??

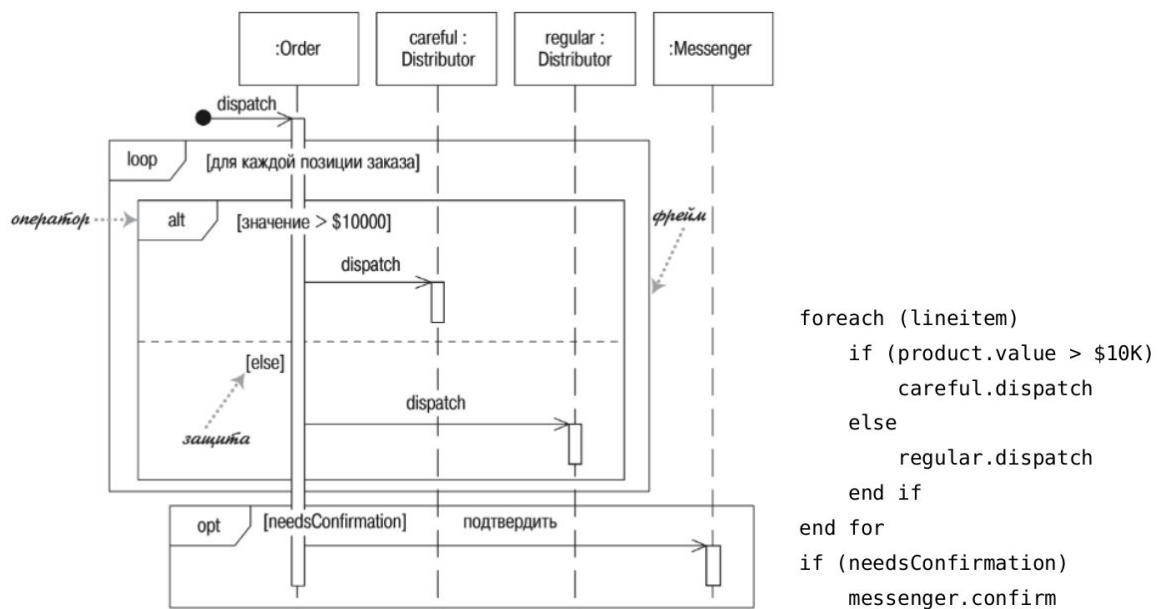
- В многопоточных/асинхронных системах
- Телекоммуникационных системах
- На этапе анализа предметной области, чтобы понять взаимодействие объектов
- Для составления плана тестирования: что когда и кому передавать
- Для отладки -- составление логов работы системы

Можно создавать/удалять (самим или GC) объекты:



Фреймы

Можно зацикливать, ставить условия и switch/case:



Фреймы ухудшают читабельность. Для описания алгоритмов:

Диаграмма активностей от диаграмм обзора взаимодействий >> диаграммы последовательностей

В UML 1 использовались маркеры итераций и защиты. В качестве маркера итерации (iteration marker) выступал символ *, добавленный к имени сообщения. Для обозначения тела итерации можно добавить текст в квадратных скобках. Защита (guard) – это условное выражение, размещеннное в квадратных скобках и означающее, что сообщение посыпается, только когда защита принимает истинное значение. Эти обозначения исключены из UML 2, но они все еще встречаются в диаграммах взаимодействия.

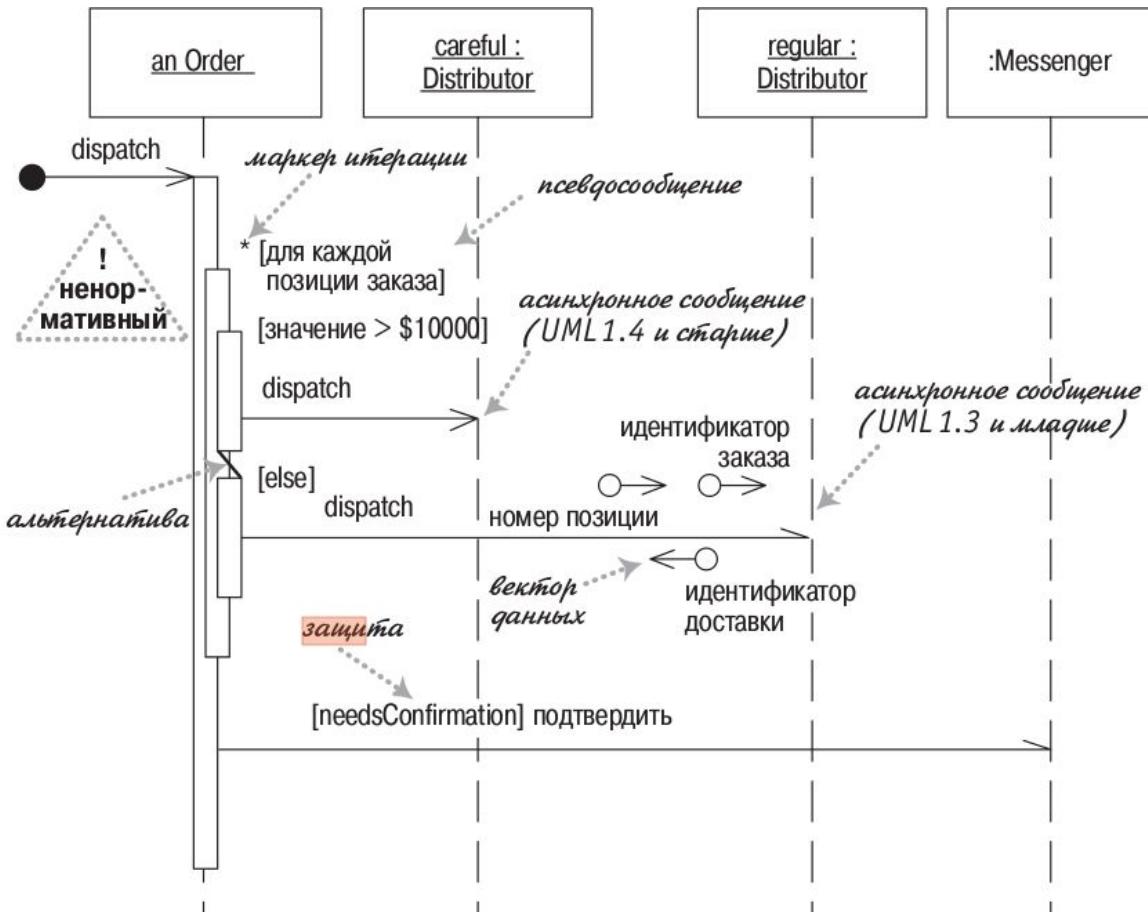


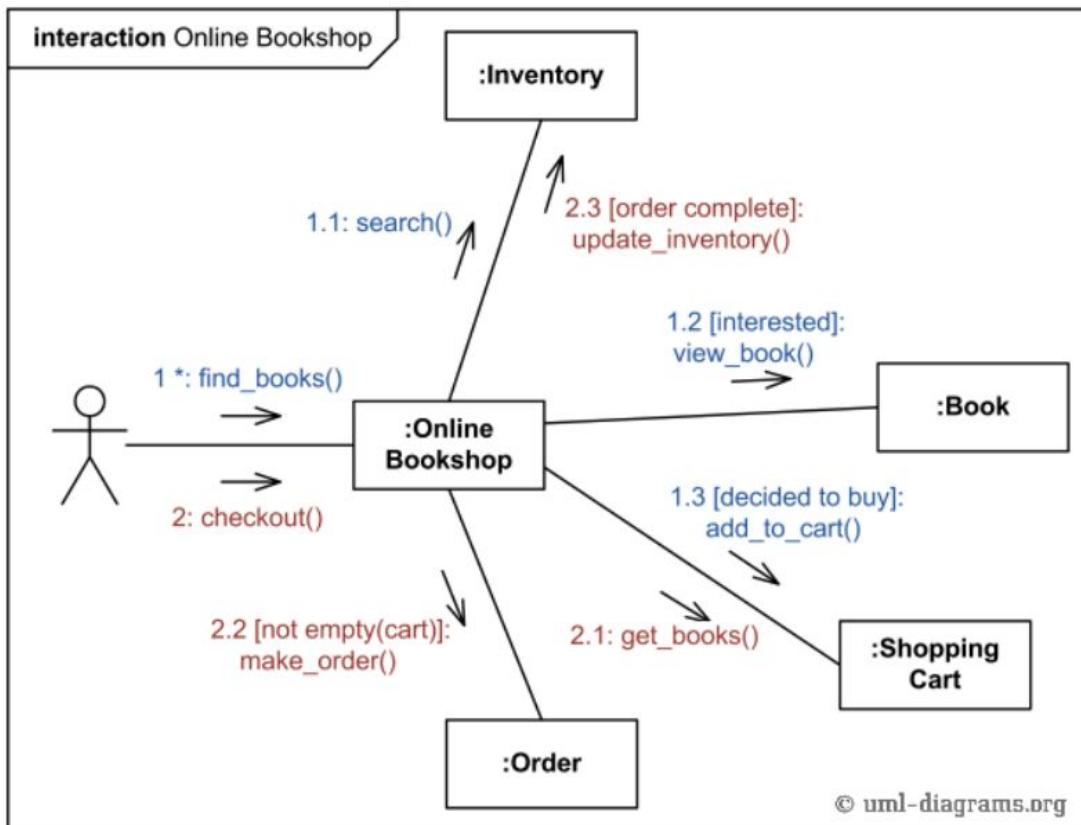
Рис. 4.5. Старые соглашения для условной логики

17. Диаграммы коммуникации UML.

Визуализирует взаимодействие между объектами, последовательность обмена сообщениями, один сценарий использования.

НО на двумерной плоскости, а не как диаграммы последовательностей. Вложенная нумерация показывает из какого метода вызван метод, расположенный ниже по иерархии.

- + компактность
- порядок действий не очевиден.



Последости vs коммуникаций:

- коммун-ций, если важны связи
- послед-сти, если важно проследить за порядком действий

До двоеточия пишется имя объекта, если оно важно, после — имя типа, если оно важно.

Коммуникационные диаграммы не имеют точно определенных нотаций для управляющей логики. Они допускают применение маркеров итерации и защиты (стр. 86), но не позволяют полностью определить алгоритм управления. Не существует также специальных обозначений для создания и удаления объектов, но ключевые слова «create» и «delete» соответствуют общепринятым соглашениям.

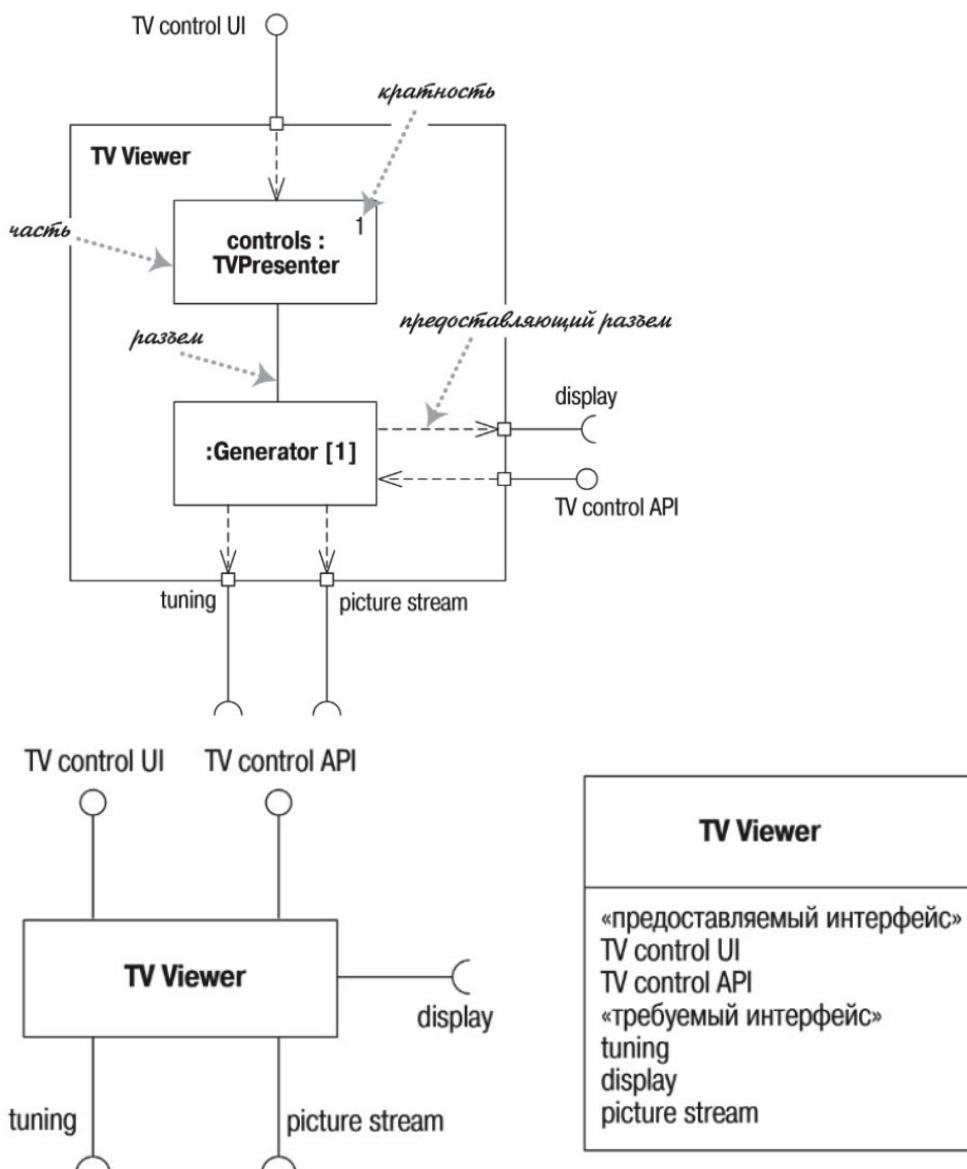
18. Диаграммы составных структур, коопераций, временные диаграммы.

Как в диаграмме компонентов, тут тоже визуализируются крупные блоки, их интерфейсы, порты, связи.

НО Диаграмм компонент показывает структуру системы времени компиляции. А диаграмм составных структур -- внутри блоков имеется не другие компоненты, а роли. Роли -- это почти то же, что и объекты. На диаграмме может быть несколько ролей одного типа.

Синтаксис составных структур

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



Диаграммы коопераций

Что-то между диаграмм объектов и диаграмм классов. Вместо объектов, классов -- роли. Роли -- сущности, на место которых может быть подставлен объект.

Можно увидеть какие роли играют классы на диаграмме классов, указав кооперацию:

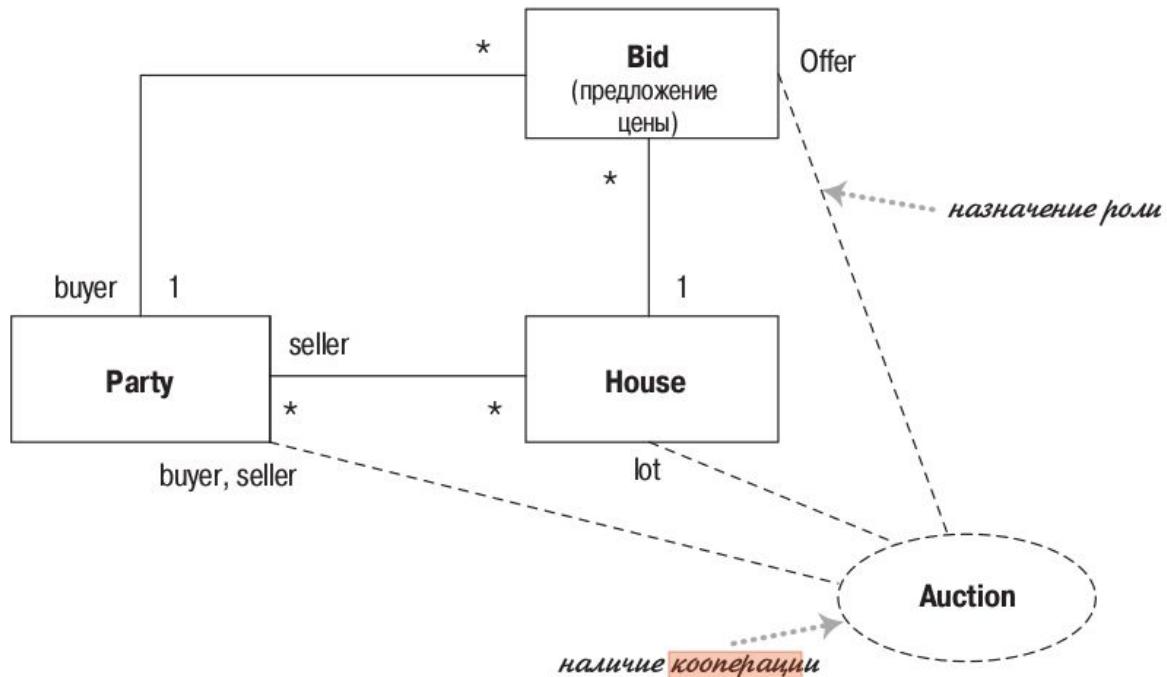
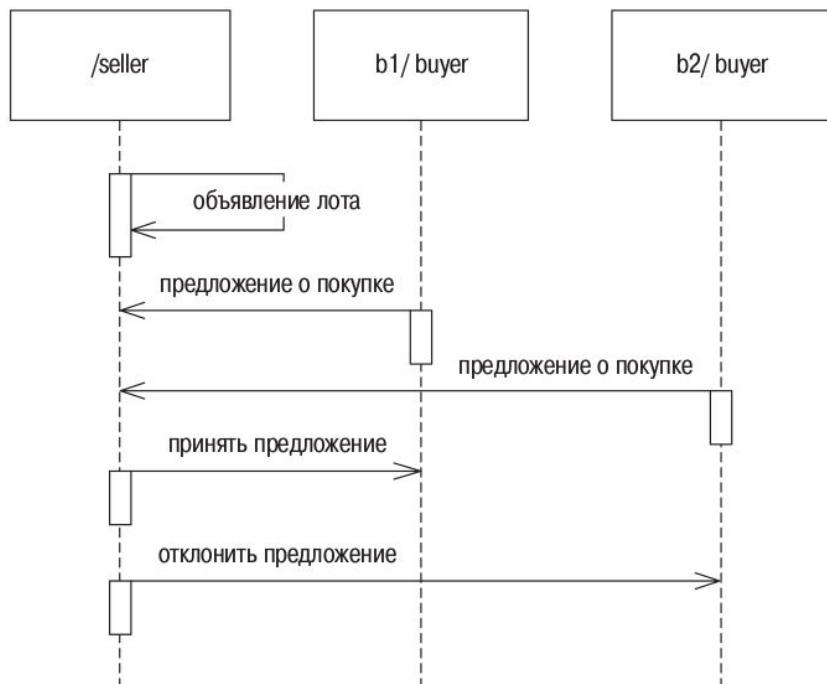


Рис. 15.3. Наличие кооперации

Можно описывать в терминах классов, а можно в терминах диаграм послед-сти:



В кооперации схема именования выглядит следующим образом: имя участника / имя роли : имя класса. Как всегда, все эти элементы необязательны.

Временные диаграммы

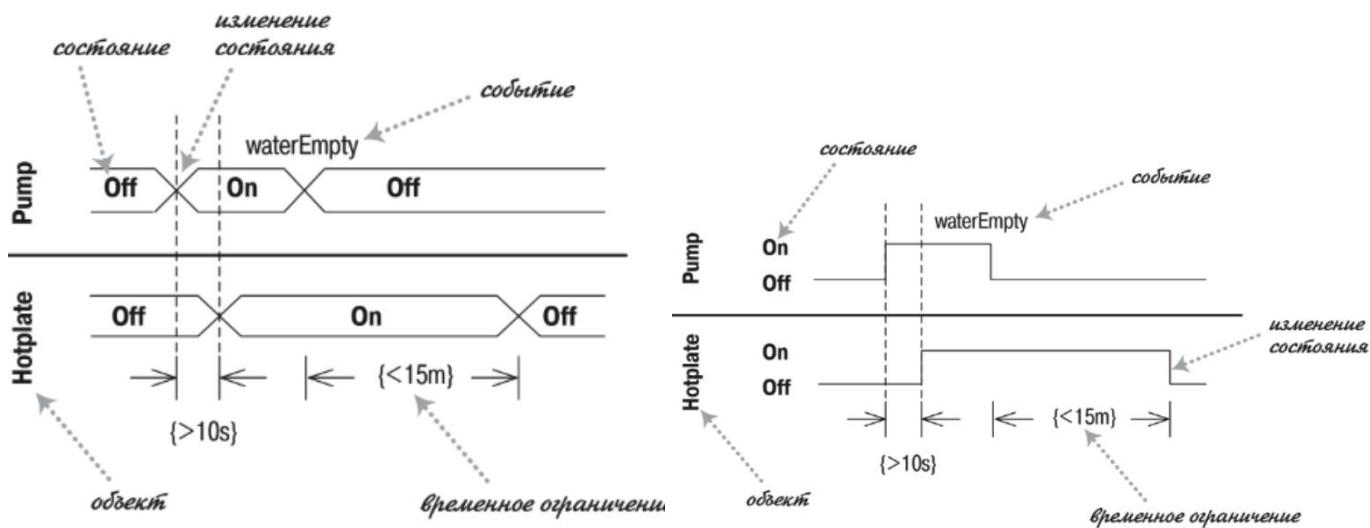
- Для кого?

Проектирование Систем Реального Времени.

- Зачем?

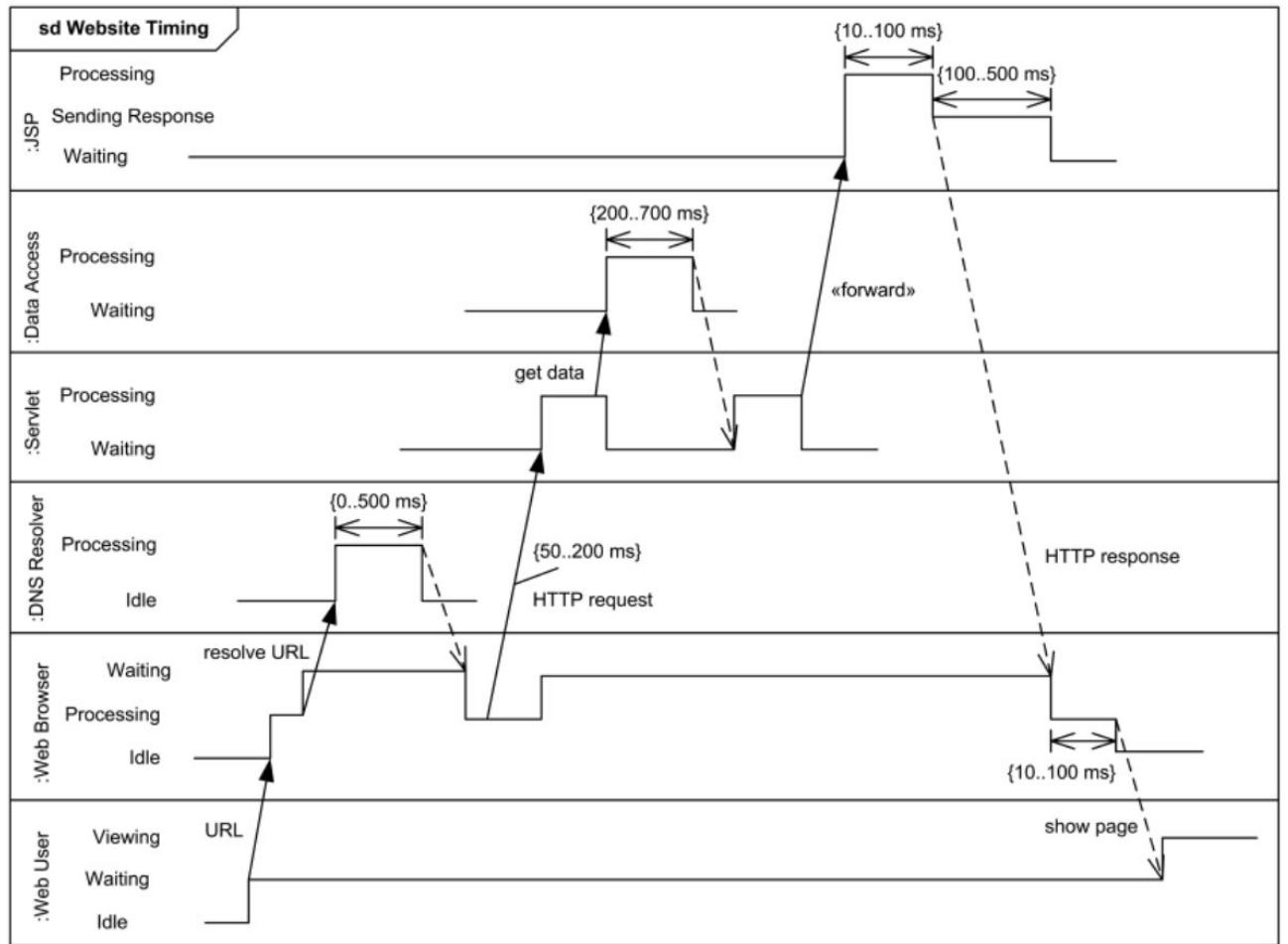
Визуализация событий с четкими временными ограничениями

Синтаксис



- + Позволяет проследить во времени за последовательностью действий, как диаграмма последовательностей
- + Видно внутреннее состояние объекта, в отличие от диаграмм последовательностей

Ещё пример:



19. Диаграммы обзора взаимодействия, диаграммы потоков данных.

Обзоры взаимодействия

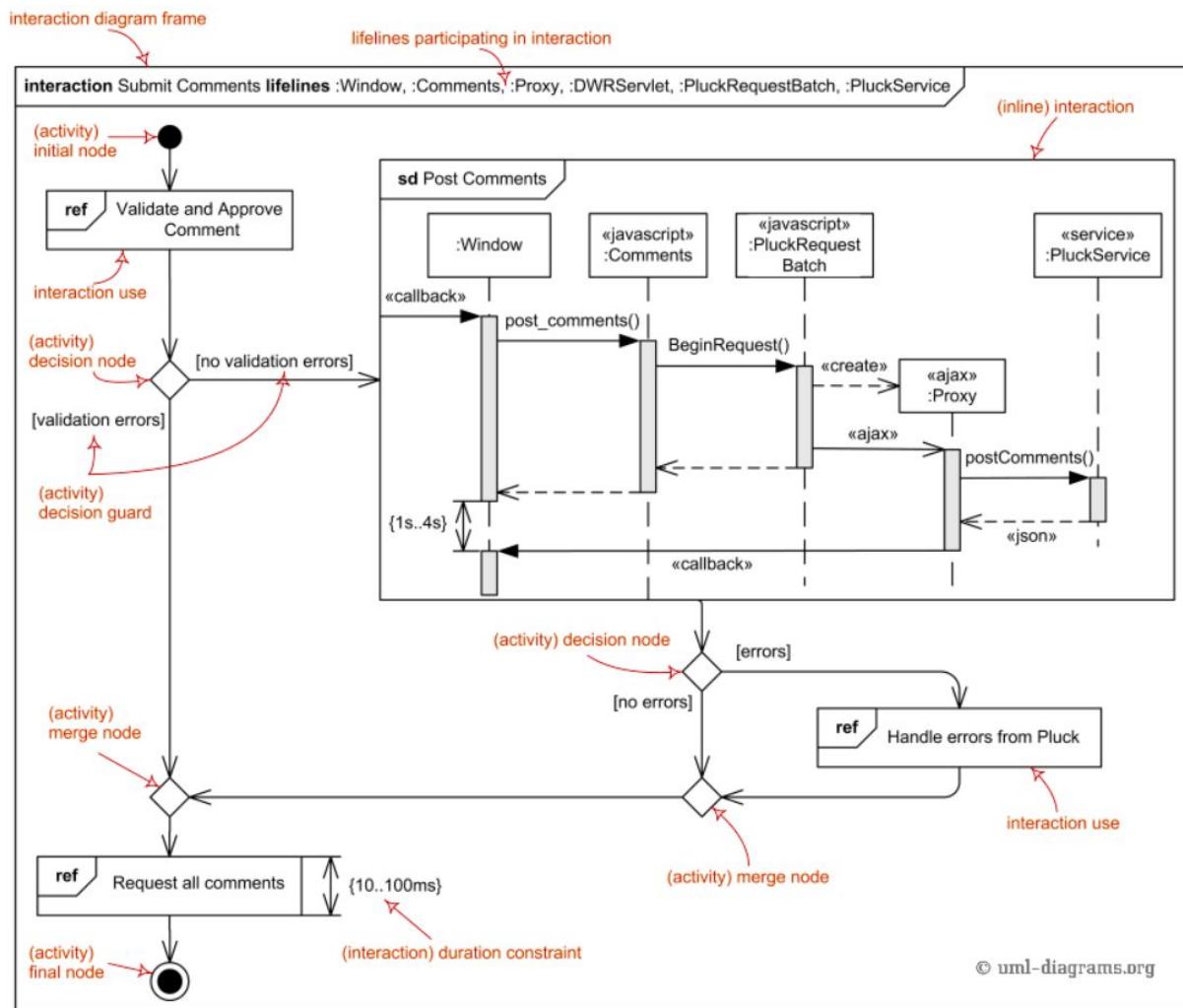
Это объединение диагр последовательности и диагр активности.

- Зачем?

Описание сложных алгоритмов, где куча ветвлений и циклов.

Могут указываться временные ограничения.

Синтаксис:



(sd) Interaction -- отображенная диаграмма.

(ref) Interaction use is an **interaction fragment** which allows to use (or call) another interaction. Large and complex sequence diagrams could be simplified with interaction uses. It is also common to reuse some interaction between several other interactions.

The notation within the rectangular frame is one of the form of:

- **sequence diagram**,
- **communication diagram**,
- **timing diagram**, or
- **interaction overview diagram**.

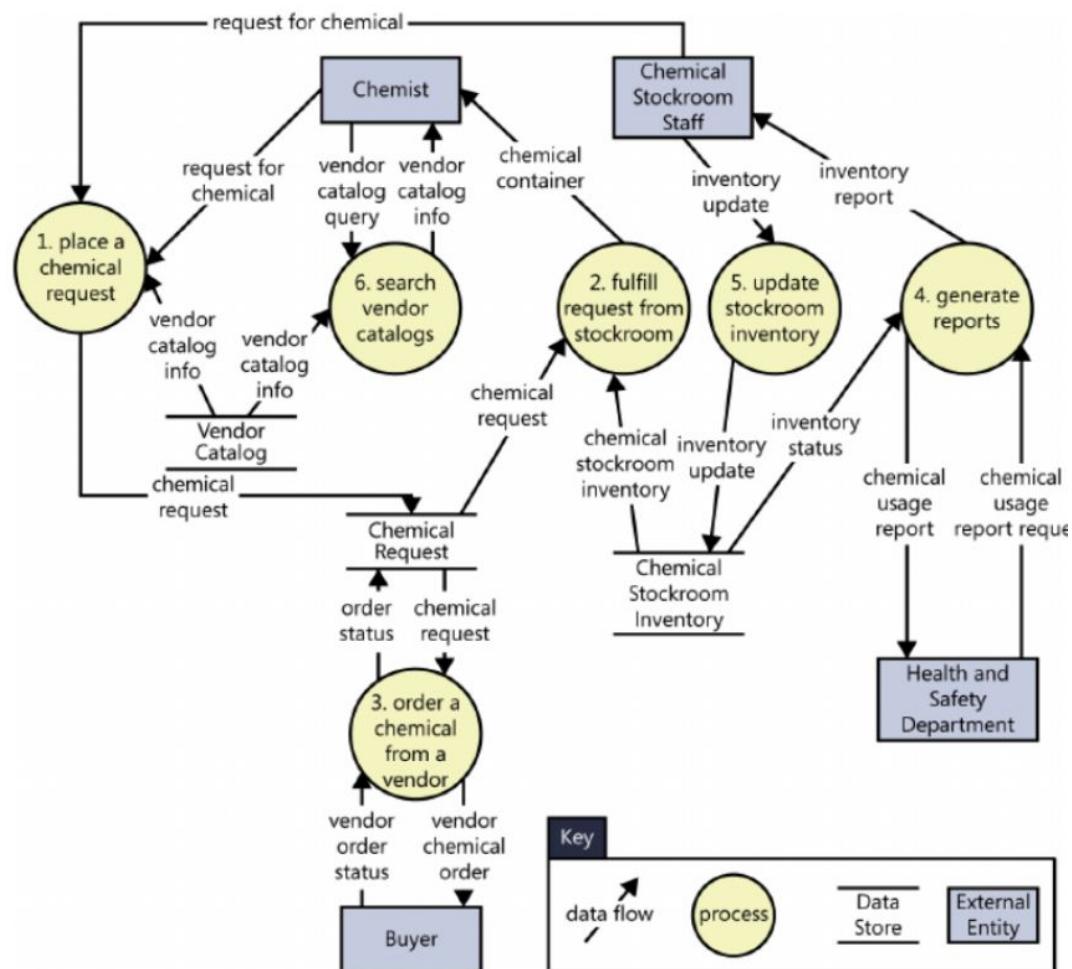
Потоков Данных

!!! НЕ выходят в UML и НЕ имеют в UML аналогов.

Первый набросок архитектуры, когда есть бизнес-процесс, где данные уже как-то ходят.

Да и вообще полезно для погружения в систему, т.к. поток данных интуитивен, но сложен.

Синтаксис



Процесс -- то, что обрабатывает данные

Хранилище данных -- хранение и извлечение.

Внешняя сущность -- то, что потребляет и поставляет данные

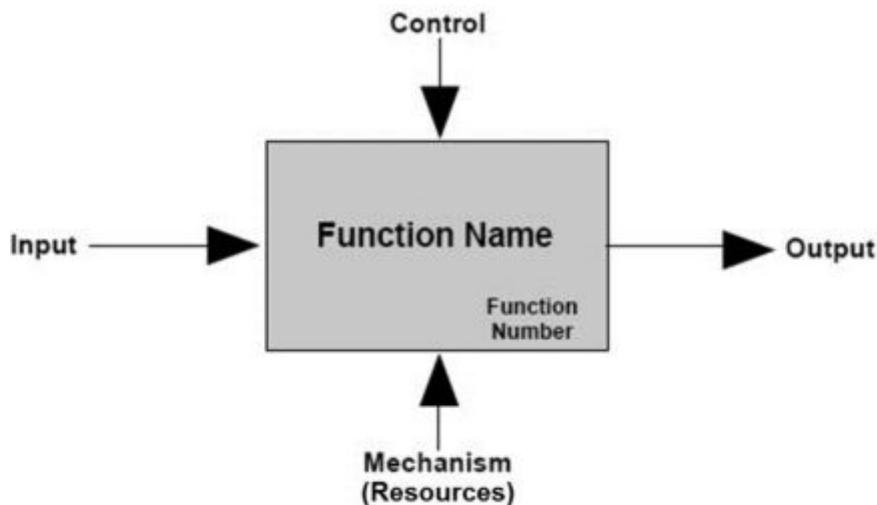
Поток данных -- по нему ходят данные.

20. Диаграммы IDEF0, характеристик. Feature tree

Контекстная диаграмма IDEF0

Для структурной декомпозиции системы. Представляется в виде шагов со входами и выходом, соединенных друг с другом.

Блок диаграммы (шаг):



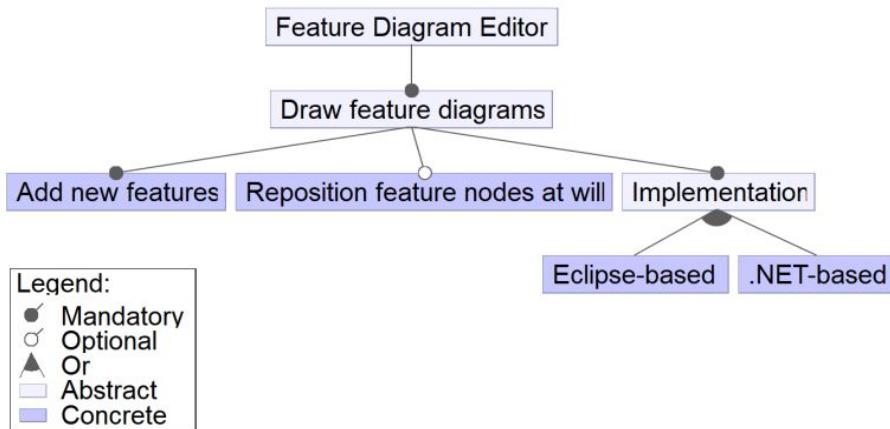
Каждый блок может быть представлен IDEF0, но входы и выходы должны совпасть.

Входы -- информация извне, выход -- полезный результат.

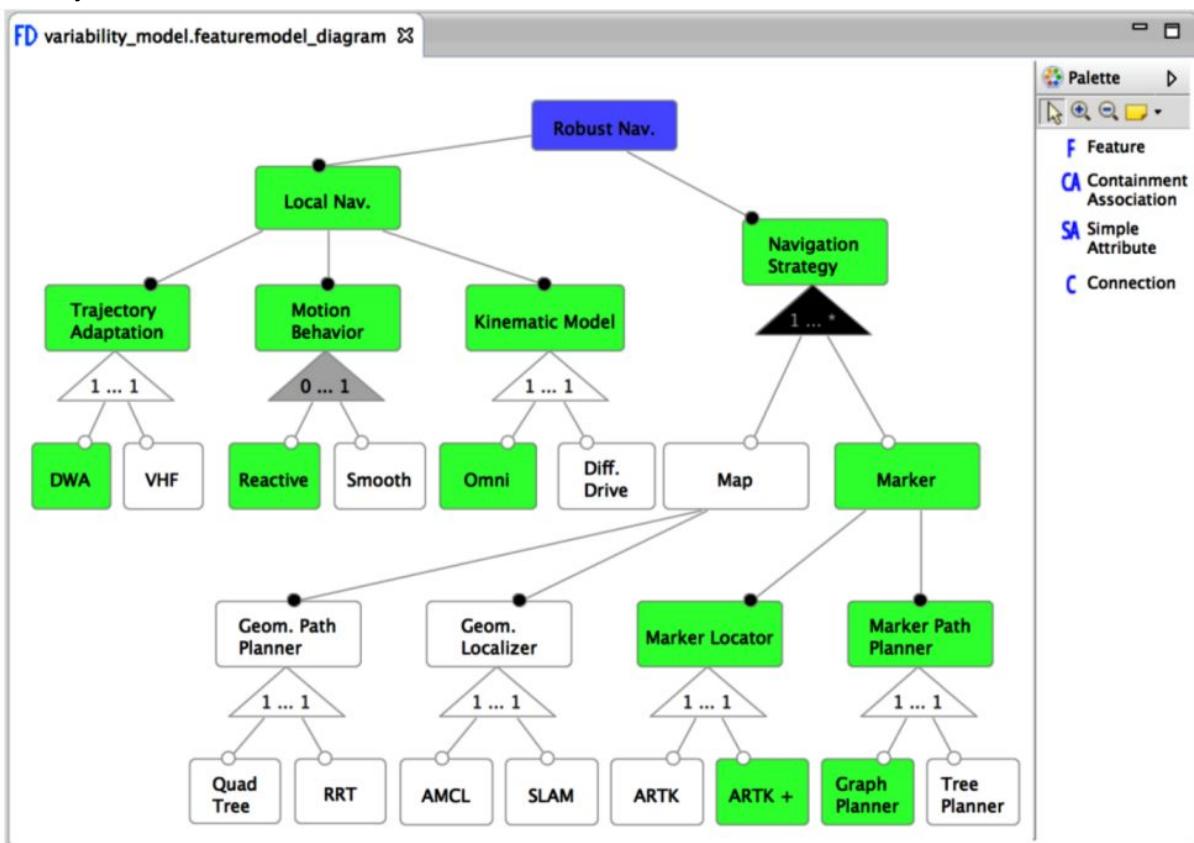
- Эти диаграммы любят аналитики для анализа бизнес-процессов.

Диаграмма характеристик

Нотация очень простая — характеристики бывают абстрактными (то есть теми, которые надо ещё декомпозировать перед тем как реализовать) и конкретными (пригодными к реализации), отношения между характеристиками бывают “или” (включающее и исключающее), обязательность и необязательность (то есть, без необязательной характеристики продукт вполне может существовать).

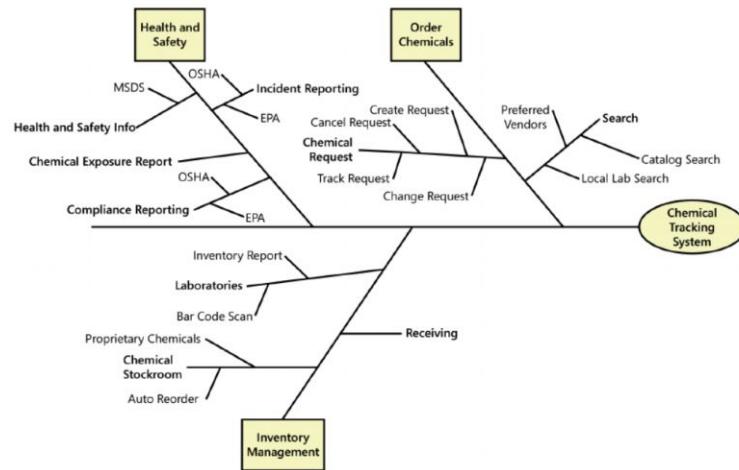


- + Позволяет описывать всевозможные потенциальные характеристики системы (не все нужны каждому новому продукту), а затем выбирать подходящие под новую задачу.



Дерево характеристик (feature tree) ((fishbone diagram))

Декомпозиция системы на фичи:



Последовательность анализа

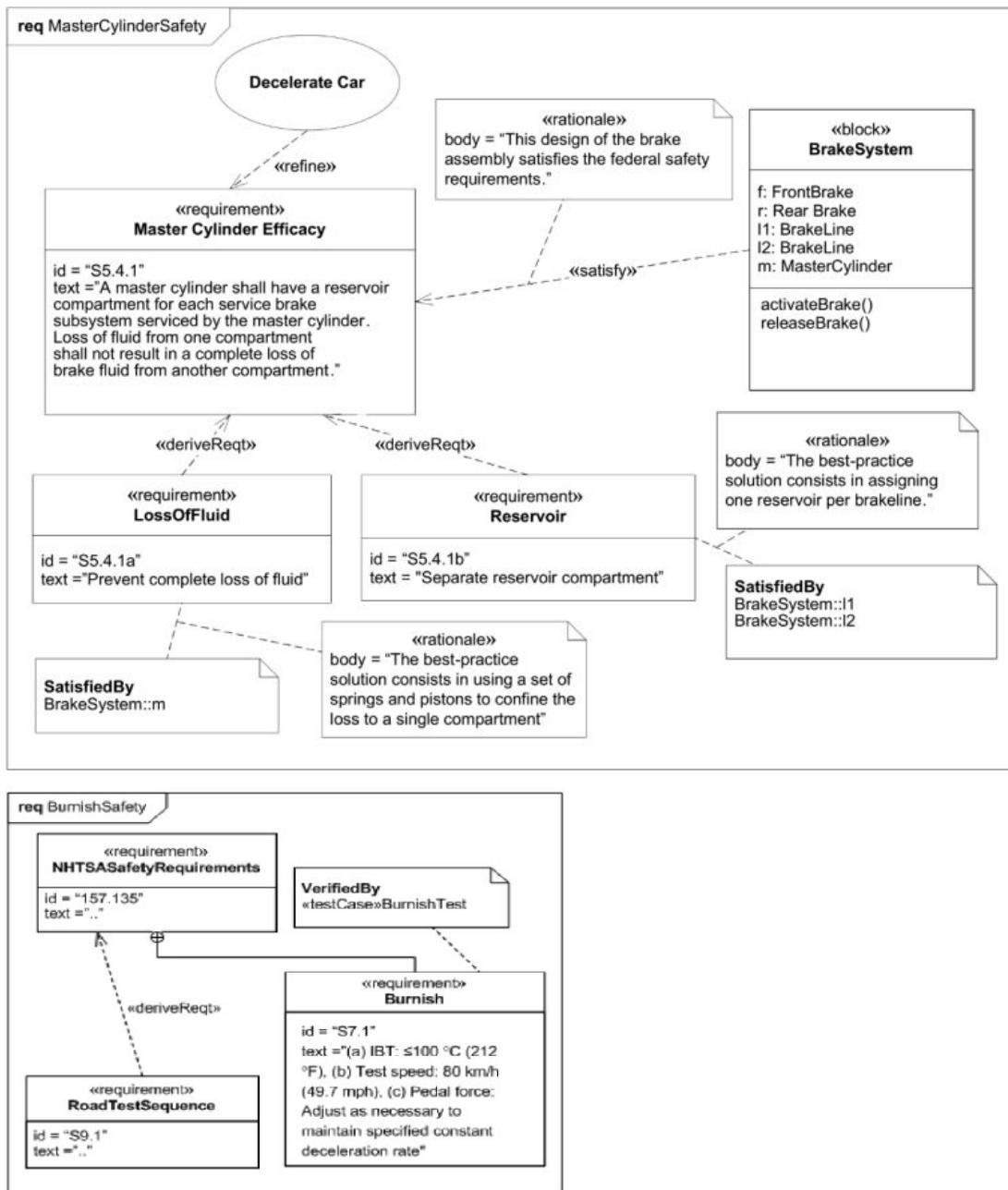
Feature tree, диаграмма активностей, диаграмма характеристик, т.д.

21. Моделирование требований в SysML.

SysML изначально -- расширение языка для больших ПО, включающих как программные так и аппаратные компоненты, *людей*, другие системы.

Диаграмма требований.

- Требование
- Уточнения требований
- Реализация требований
- Аргументирование решений
- Тестирование
 - Сценарий тестирования может представляться диаграммой активностей



22. ?! Язык BPMN.

(Business Process Model and Notation)

Язык для более продвинутого анализа бизнес-требований и моделирования и визуализации бизнес-процессов. Для анализа множества бизнес-процессов: аутсорсинг; внутренние процессы, включающие процессы с аутсорсерами.

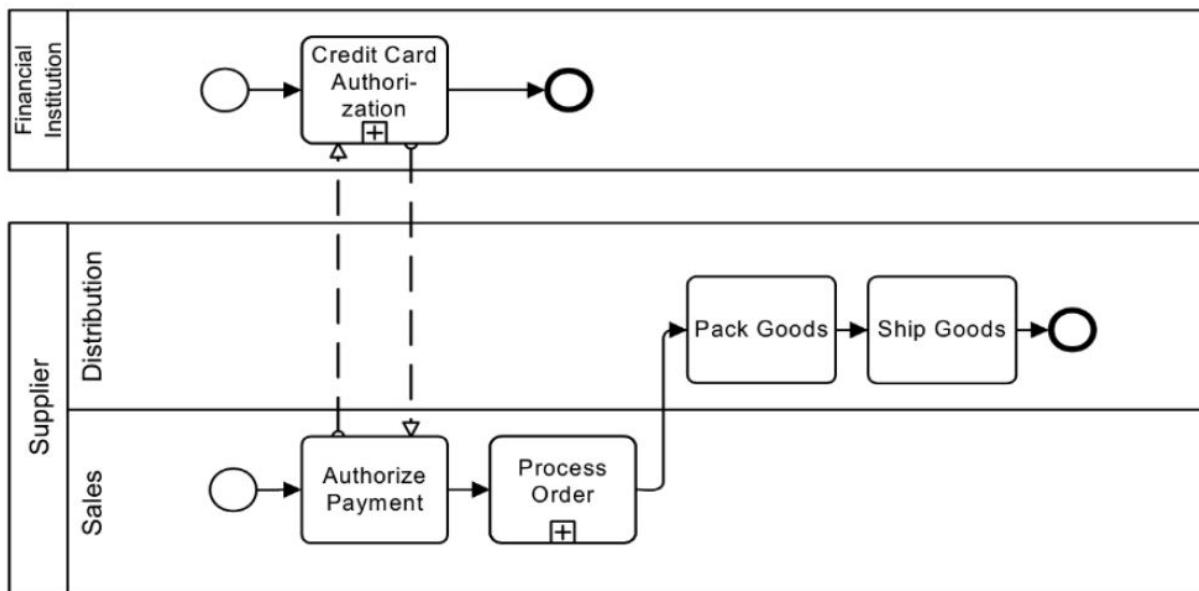
Нужен НЕ архитектор, а бизнес-аналитик, с которым общается архитектор.

Можно считать более продвинутой диаграммой активностей, в отличие от которой может:

- визуализировать несколько процессов
- качественно поддерживает исключения
- бОльший набор блоков-ветвлений
- бОльший набор блоков ожиданий событий

Может быть реально исполнен на движках, способных читать исполняемый язык BPEL (Business Process Execution Language), являющимся XML-документом, сконвертированным из языка BPMN.

Диаграмма бизнес-процессов.



- + -- раскрывается в другой детализированный бизнес-процесс
- Видно две организации
 - В одной организации бизнес-процесс поделена на два отдела, но все же один поток управления
- Общаются путём отправки сообщений

(<https://ru.wikipedia.org/wiki/BPMN>):

Категории элементов BPMN

❖ Объекты потока управления

➤ События

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

События

изображаются окружностью и означают какое-либо произшествие в мире. События инициируют действия или являются их результатами. Согласно расположению в процессе события могут быть классифицированы на начальные (англ. *start*), промежуточные (*intermediate*) и завершающие (*end*). Начиная с BPMN 1.1 различают события обработки и генерации. Ниже представлена категоризация событий по типам.

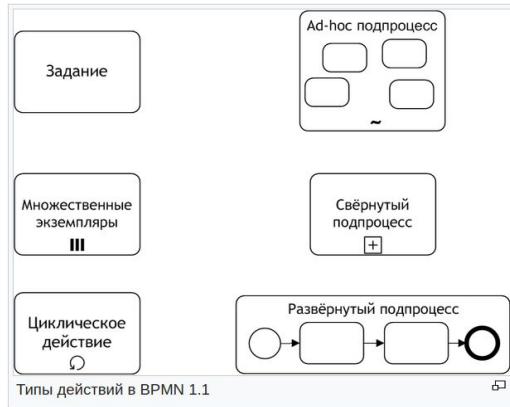
- Простые события (*plain events*) это нетипизированные события, использующиеся, чаще всего, для того, чтобы показать начало или окончание процесса.
- События-сообщения (*message events*) показывают получение и отправку сообщений в ходе выполнения процесса.

- События-таймеры (*timer events*) моделируют события, регулярно происходящие во времени. Также позволяют моделировать моменты времени, периоды и тайм-ауты.
- События-ошибки (*error events*) позволяют смоделировать генерацию и обработку ошибок в процессе. Ошибки могут иметь различные типы.
- События-отмены (*cancel events*) инициируют или реагируют на отмену транзакции.
- События-компенсации (*compensation events*) инициируют компенсацию или выполняют действия по компенсации.
- События-условия (*conditional events*) позволяют интегрировать бизнес правила в процесс.
- События-сигналы (*signal events*) рассылают и принимают сигналы между несколькими процессами. Один сигнал может обрабатываться несколькими получателями. Таким образом, события-сигналы позволяют реализовать широковещательную рассылку сообщений.
- Составные события (*multiple events*) моделирует генерацию и моделирование одного события из множества.
- События-ссылки (*link events*) используются как межстраничные соединения. Пара соответствующих ссылок эквивалентна потоку управления.
- События-остановы (*terminate events*) приводят к немедленному завершению всего бизнес процесса (во всей диаграмме).

➤ Действия

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

- Задание (*task*) — это единица работы, элементарное действие в процессе.
- Множественные экземпляры (*multiple instances*) действия показывают, что одно действие выполняется многократно, по одному разу для каждого объекта. Например, для каждого объекта в заказе клиента выполняется один экземпляр действия. Экземпляры действия могут выполняться параллельно или последовательно.
- Циклическое действие (*loop activity*) выполняется, пока условие цикла верно. Условие цикла может проверяться до или после выполнения действия.
- Развёрнутый подпроцесс (*expanded subprocess*) является сложным действием и содержит внутри себя собственную диаграмму бизнес-процессов.
- Свёрнутый подпроцесс (*collapsed subprocess*) также является составным действием, но скрывает детали реализации процесса.
- Ad-hoc-подпроцесс (*ad-hoc subprocess*) содержит задания. Задания выполняются до тех пор, пока не выполнено условие завершения подпроцесса.



➤ Логические операторы (развилки)



Логические операторы (развилки)

изображаются ромбами и представляют точки принятия решений в процессе. С помощью логических операторов организуется ветвление и синхронизация потоков управления в модели процесса.

- Оператор **исключающего «или»**, управляемый данными (англ. *data-based exclusive gateway*). Если оператор используется для ветвления, то поток управления направляется лишь по одной исходящей ветви.
 - **СИНХР.** Если оператор используется для синхронизации, то он ожидает завершения выполнения одной входящей ветви и активирует выходной поток.
- Оператор исключающего «или», управляемый событиями (*event-based exclusive gateway*) направляет поток управления лишь по той исходящей ветви, на которой первой произошло событие. После оператора данного типа могут следовать только события или действия-обработчики сообщений.
- Оператор **включающего «или»** (*inclusive gateway*) активирует одну или более исходящих ветвей, в случае, когда осуществляется ветвление (**ТО ЕСТЬ** если условия выполняются, то активирует несколько).

- **СИНХР.** Если оператор используется для синхронизации, то он ожидает завершения выполнения всех активированных ветвей и активирует выходной поток.
- Оператор «и» (*parallel gateway*), использующийся для ветвления, разделяет один поток управления на несколько параллельных. При этом все исходящие ветви активируются одновременно.
 - **СИНХР.** Если оператор используется для синхронизации, то он ожидает завершения выполнения всех входящих ветвей и лишь затем активирует выходной поток.
- Сложный оператор (*complex gateway*) имеет несколько условий, в зависимости от выполнения которых активируются исходящие ветви. Оператор затрудняет понимание диаграммы, так как условия, определяющие семантику оператора, графически не выражены на диаграмме. Вследствие этого использование оператора нежелательно.

❖ **Соединяющие объекты (соединение объектов потока управления)**

➤ Поток управления

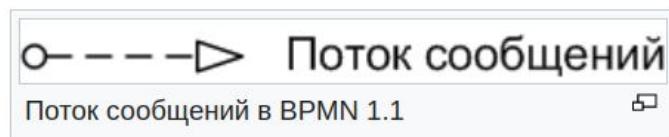
- Поток управления задаёт порядок выполнения действий. Если линия потока управления перечеркнута диагональной чертой со стороны узла, из которого она исходит, то она обозначает поток, выполняемый по умолчанию

➤ Поток сообщения

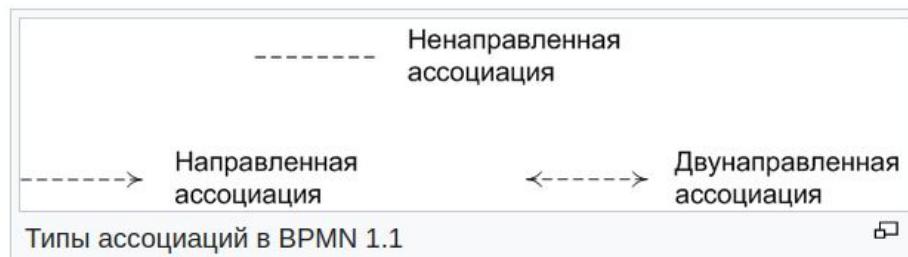
- изображается штриховой линией, оканчивающейся открытой стрелкой. Поток сообщений показывает, какими сообщениями обмениваются участники.

➤ Ассоциации

- используются для ассоциирования артефактов (данных или текстовых аннотаций) с объектами потока управления



Поток сообщений изображается какими сообщ

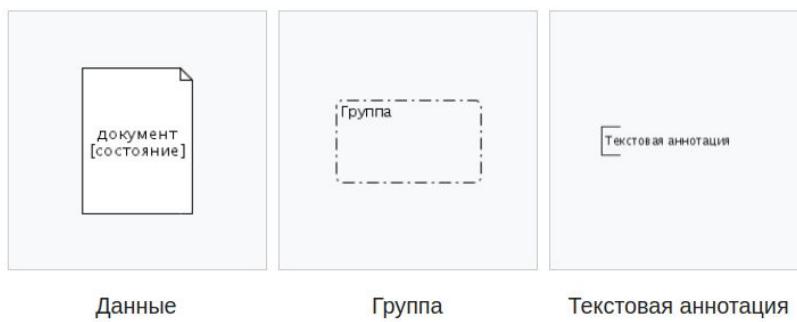


❖ Роли

- Роли — визуальный механизм организации различных действий в категории со сходной функциональностью
 - 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.

❖ Артефакты

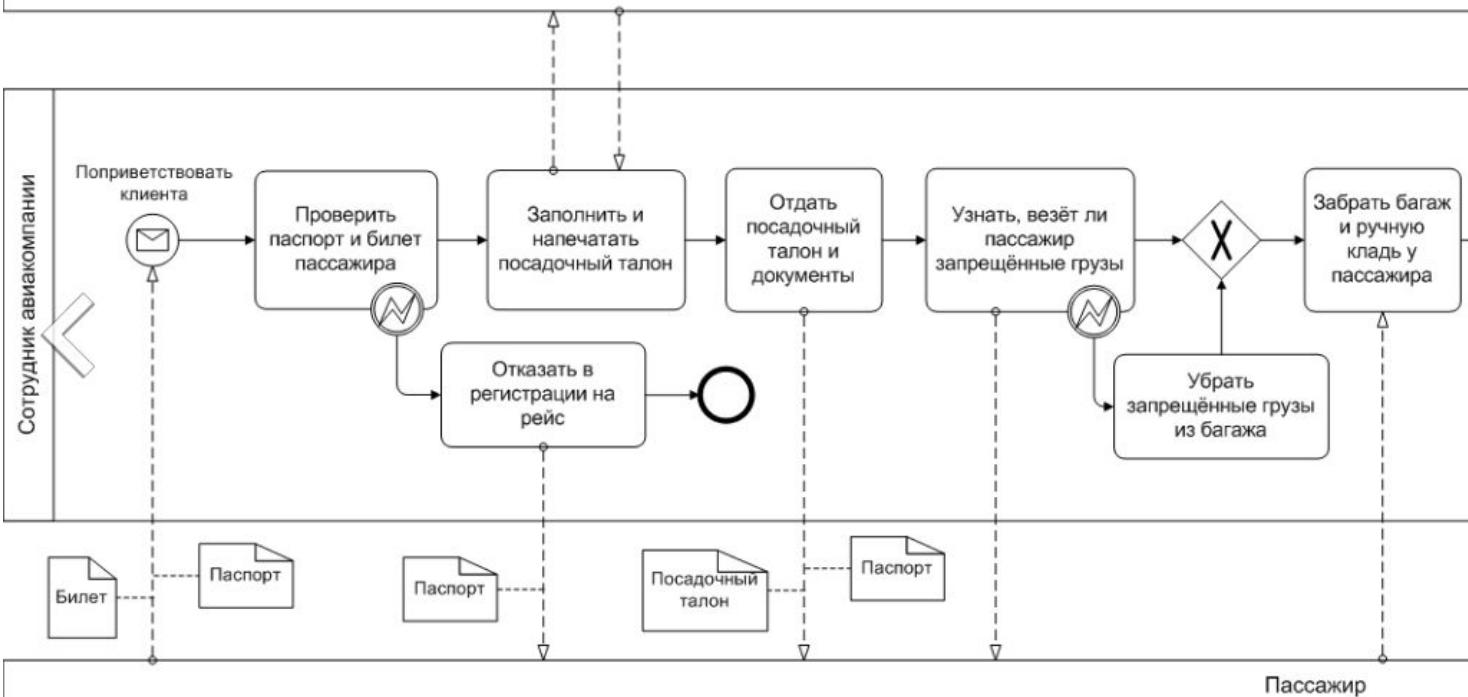
- Данные
 - показывают читателю, какие данные необходимы действиям для выполнения и какие данные действия производят.
- Группа
 - Группа позволяет объединять различные действия, но не влияет на поток управления в диаграмме.
- Текстовая аннотация
 - используются для уточнения значения элементов диаграммы и повышения её информативности.



Пример

https://ru.wikipedia.org/wiki/BPMN#/media/%D0%A4%D0%B0%D0%B9%D0%BB:Check-in_BPMN_example.png:

Информационная система авиакомпаний



и компаний

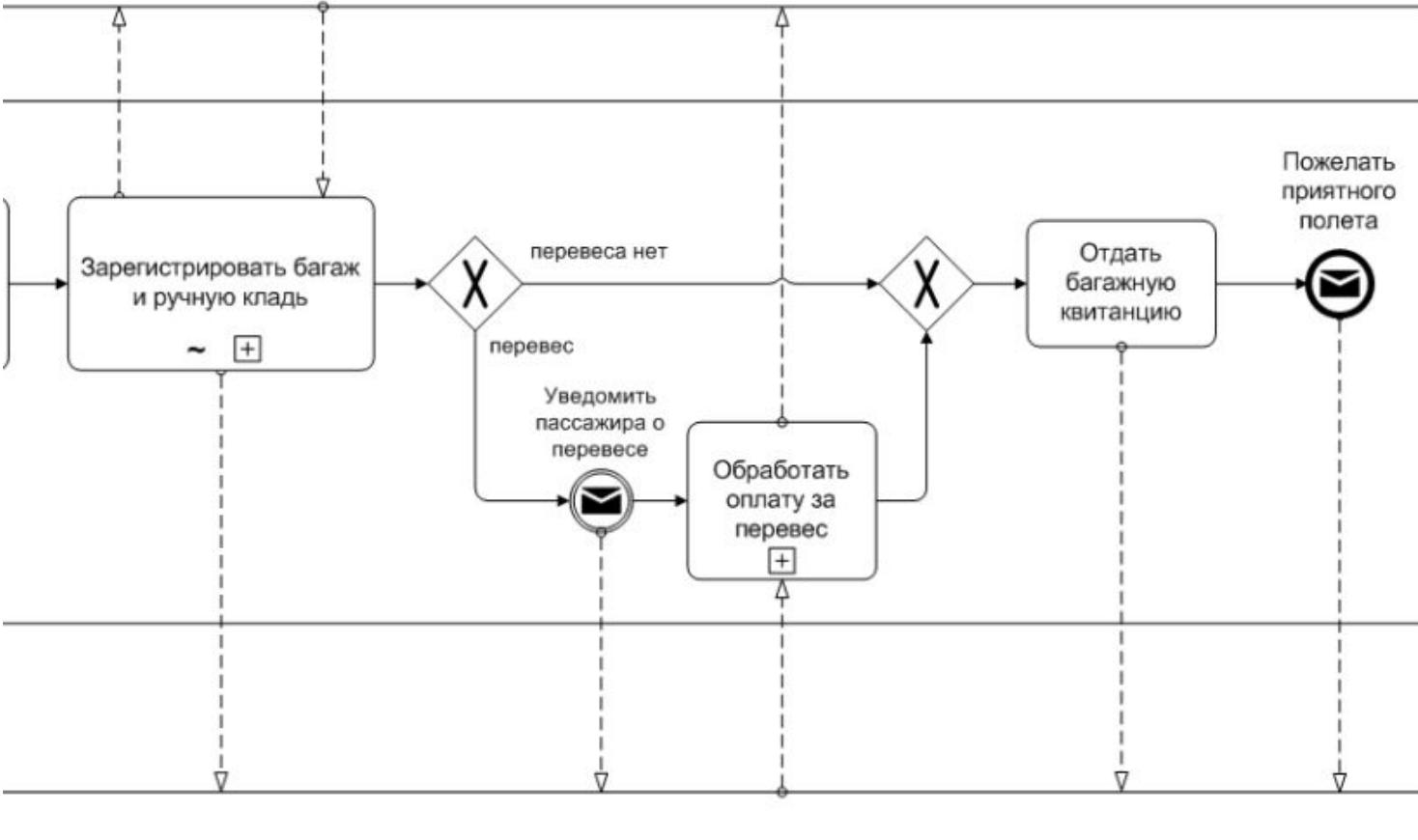
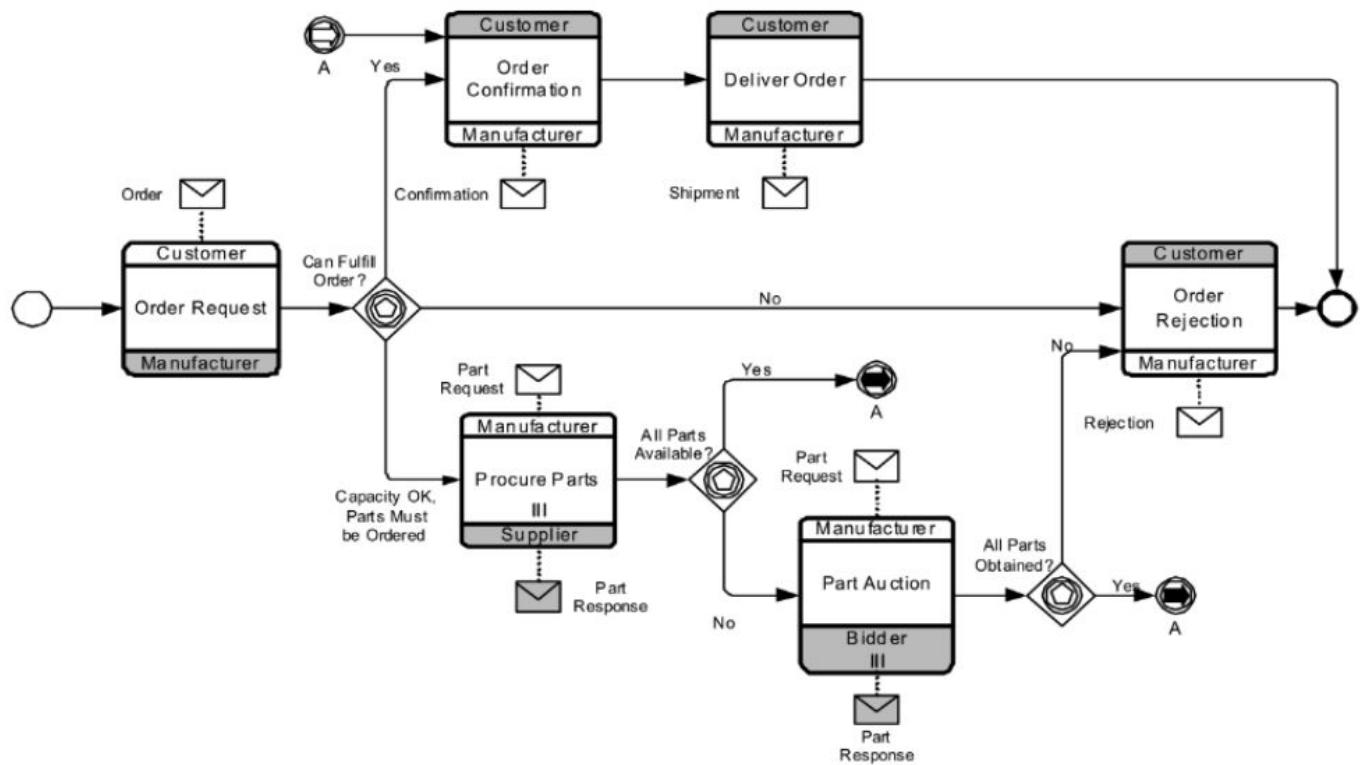


Диаграмма хореографии

Показывает исключительно взаимодействия между бизнес-процессами. Рисуются не то, кто что делает, а то, кто когда с кем общается..



Скругленные квадратики -- точки общения между процессами. Белые объект -- источник общения. Серый -- тот, кто отвечает. Конверт -- сообщение или документ. События и условия указываются, действия -- нет

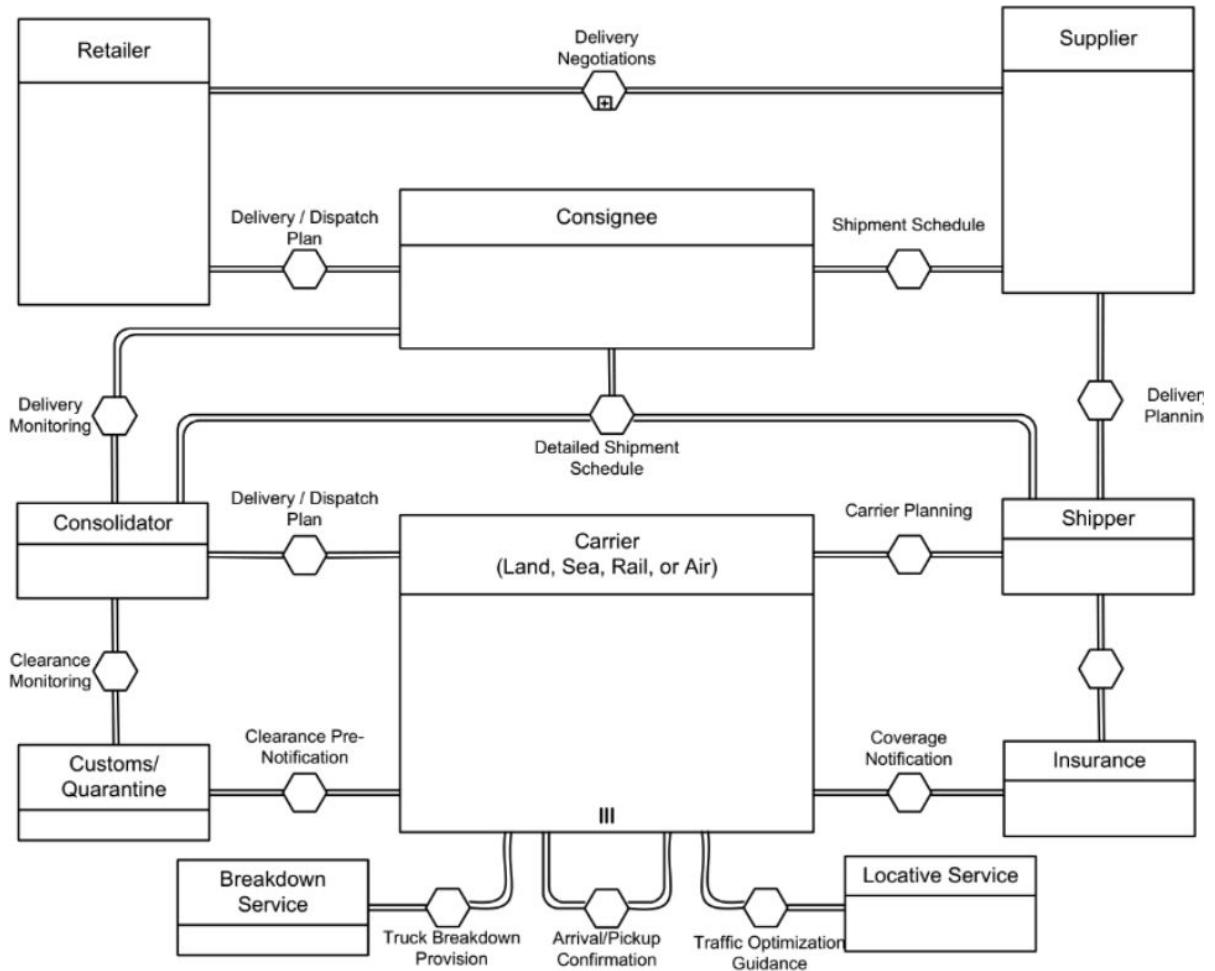
Зачем??

Если указание действий на дорожках на диаграмме приведет к хаосу из стрелочек.

Диаграмма диалогов

Показывает кто с кем в ПРИНЦИПЕ может общаться.

Прямоугольники -- участники общения. Быть угольники и двойные линии -- общение между участниками. "+" -- обснений много и они детализируются диаграммой хореографии или бизнес-процессом без действий. З палочки внутри участника -- участников может быть несколько.

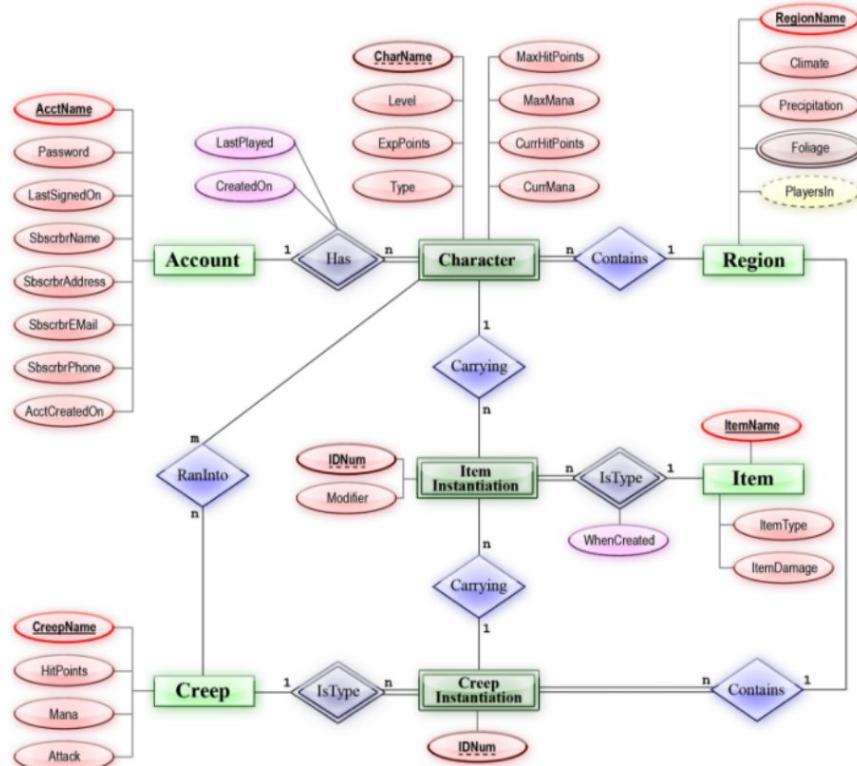


23. Моделирование данных: диаграммы «Сущность-связь»

При моделировании предметной области присутствует этап анализа данных. Анализируются сущности и их свойства. В обязанностях архитектора присутствует задача построения концептуальной модели базы данных. Для этого существуют средства визуализации:

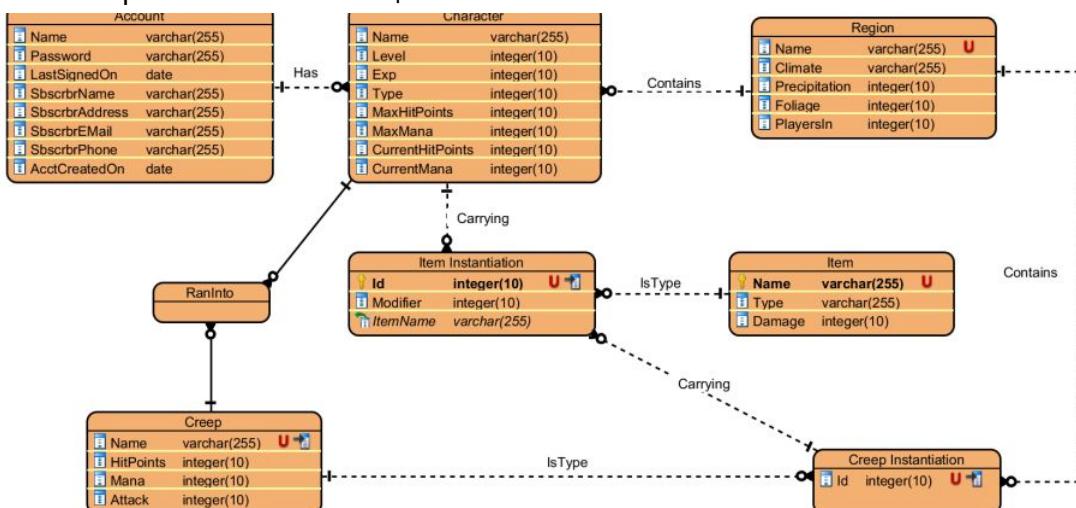
- Чена --

- не используется, т.к. громоздко



- Нотация “вороньей лапки” --

- воронья лапка на конце множественной связи. Более компактна



24. Концептуальное моделирование, диаграммы ORM.

Object role modeling (**HE** object relational mapping).

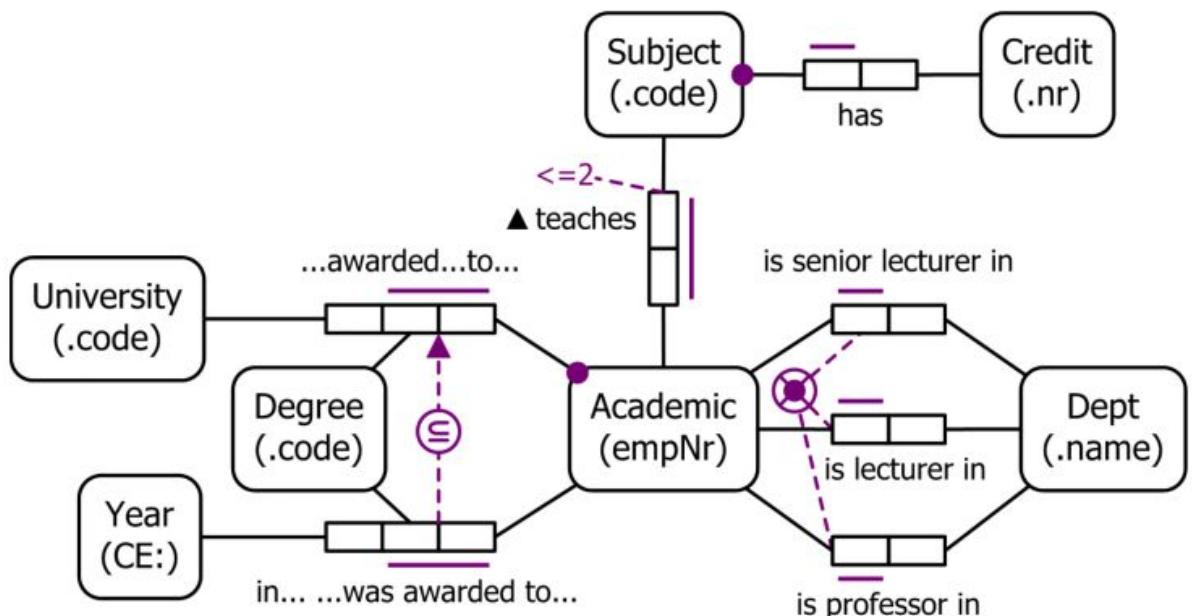
Зачем?

Для концептуального моделирования данных на начальном этапе анализа предметной области, когда знаний о ней ещё мало.

Есть только сущности и связи

Атрибутов нет =>

- более громоздка чем ER-diagram,
- но устойчивее к изменениям, т.к. просто можно добавить новую сущность, создав новую сущность-связь. Тогда как в ER-diag пришлось бы рисовать новую таблицу, изменять старую, добавлять связи



Синтаксис

1. Сущность с её идентификатором
2. Связи, играющие роль таблиц. Сущности играют *роли* в связях, устанавливаясь в свой прямоугольник.
3. Линии рядом со связью указывают на уникальность *сочетаний* сущностей
4. Связи читаются как предложения
5. Связи имеют ограничения

25. Понятие и примеры CASE-систем.

Сейчас CASE-система -- инструмент для работы с *визуальными языками*. Это не просто графические редакторы с формами. Это инструмент, понимающий визуальный язык и способный сообщать об ошибках при визуализировании: *от наличия "класс" в палитре до поиска антипаттернов*.

CASE-системы способны генерировать код по визуальной модели.

Функциональность CASE-инструментов:

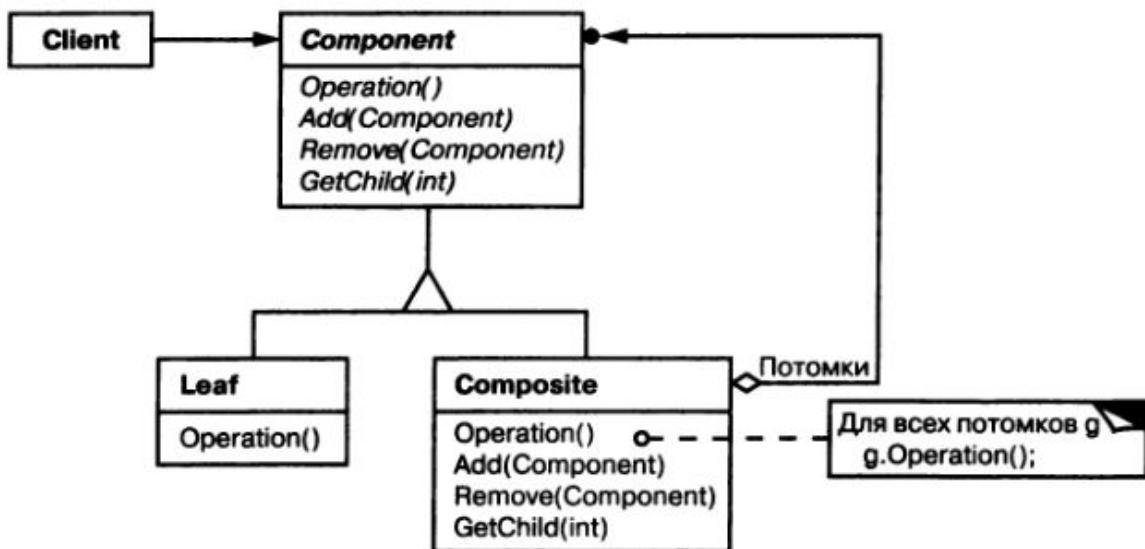
- Набор визуальных редакторов:
 - UML, ER, BPMN и т.п.
- Репозиторий
 - Хранилище информации о моделях.
 - Доступный интерфейс, предоставляющий доступ для внешних нужд
- Генерация кода
 - Обычно заглушками методов
 - По диаграммам конечных автоматов единицы могут и без заглушек
- Текстовый редактор
 - Редактирование сгенеренного кода
 - Документация
- Средства обратного проектирования
 - Код по диаграммам
 - Диаграммы по коду
 - Visual Studio хитрит, т.к. код уже есть, а диаграммы отображают этот код
- Средства *верификации* и анализа моделей -- очень редки
- Средства отладки и тестирования -- очень редки
 - Конечные автоматы
 - Диаграммы активностей
- API для получения данных о модели
- Средства контроля версий
 - Выложить бинарники на git можно, но git не умеет следить за изменениями диаграмм
- Шаблоны и примеры

Примеры

- Рисовалки
 - Dia for linux
 - Microsoft Visio for Win
- Полноценный
 - Visual Paradigm

26. Паттерн «Компоновщик».

Структура



Участники

Участники

- **Component (Graphic)** – компонент:
 - объявляет интерфейс для компонуемых объектов;
 - предоставляет подходящую реализацию операций по умолчанию, общую для всех классов;
 - объявляет интерфейс для доступа к потомкам и управления ими;
 - определяет интерфейс для доступа к родителю компонента в рекурсивной структуре и при необходимости реализует его. Описанная возможность необязательна;
- **Leaf (Rectangle, Line, Text, и т.п.)** – лист:
 - представляет листовые узлы композиции и не имеет потомков;
 - определяет поведение примитивных объектов в композиции;
- **Composite (Picture)** – составной объект:
 - определяет поведение компонентов, у которых есть потомки;
 - хранит компоненты-потомки;
 - реализует относящиеся к управлению потомками операции в интерфейсе класса Component;
- **Client** – клиент:
 - манипулирует объектами композиции через интерфейс Component.

Цель

Назначение

Компонует объекты в древовидные структуры для представления иерархий часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

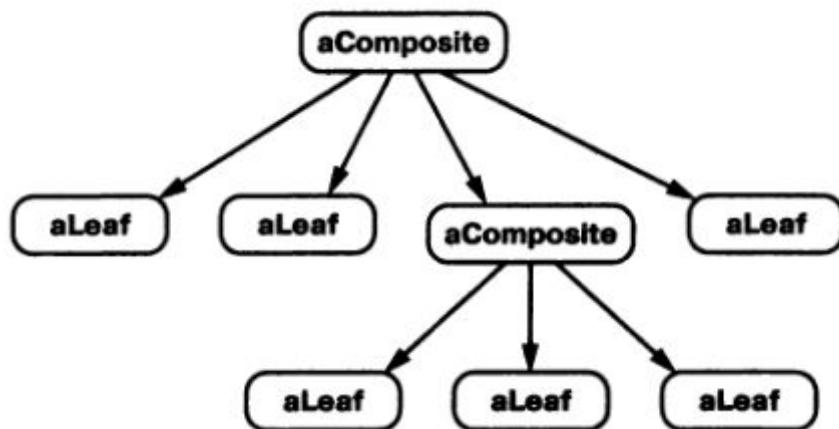
Применимость

Применимость

Используйте паттерн **компоновщик**, когда:

- ❑ нужно представить иерархию объектов вида часть-целое;
- ❑ хотите, чтобы клиенты единообразно трактовали составные и индивидуальные объекты.

Пример структуры:



Результаты

Результаты

Паттерн компоновщик:

- *определяет иерархии классов, состоящие из примитивных и составных объектов.* Из примитивных объектов можно составлять более сложные, которые, в свою очередь, участвуют в более сложных композициях и так далее. Любой клиент, ожидающий примитивного объекта, может работать и с составным;
- *упрощает архитектуру клиента.* Клиенты могут единообразно работать с индивидуальными и объектами и с составными структурами. Обычно клиенту неизвестно, взаимодействует ли он с листовым или составным объектом. Это упрощает код клиента, поскольку нет необходимости писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают;
- *облегчает добавление новых видов компонентов.* Новые подклассы классов Composite или Leaf будут автоматически работать с уже существующими структурами и клиентским кодом. Изменять клиента при добавлении новых компонентов не нужно;
- *способствует созданию общего дизайна.* Однако такая простота добавления новых компонентов имеет и свои отрицательные стороны: становится трудно наложить ограничения на то, какие объекты могут входить в состав композиции. Иногда желательно, чтобы составной объект мог включать только определенные виды компонентов. Паттерн компоновщик не позволяет воспользоваться для реализации таких ограничений статической системой типов. Вместо этого следует проводить проверки во время выполнения.

???

юзается приспособленец. Что это за паттерн

Реализация

- ▶ 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
- ▶ Операция getComposite() – более аккуратный аналог cast-а
- ▶ Где определять список потомков
 - ▶ В Composite, экономия памяти
 - ▶ В Component, единство 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
 - ▶ Инвалидизация кеша
- ▶ Удаление потомков
 - ▶ Если нет сборки мусора, то лучше в Composite
 - ▶ Следует опасаться разделяемых листьев/поддеревьев

Родственные паттерны

Родственные паттерны

Отношение компонент-родитель используется в паттерне цепочка обязанностей.

Паттерн декоратор часто применяется совместно с компоновщиком. Когда декораторы и компоновщики используются вместе, у них обычно бывает общий родительский класс. Поэтому декораторам придется поддержать интерфейс компонентов такими операциями, как Add, Remove и GetChild.

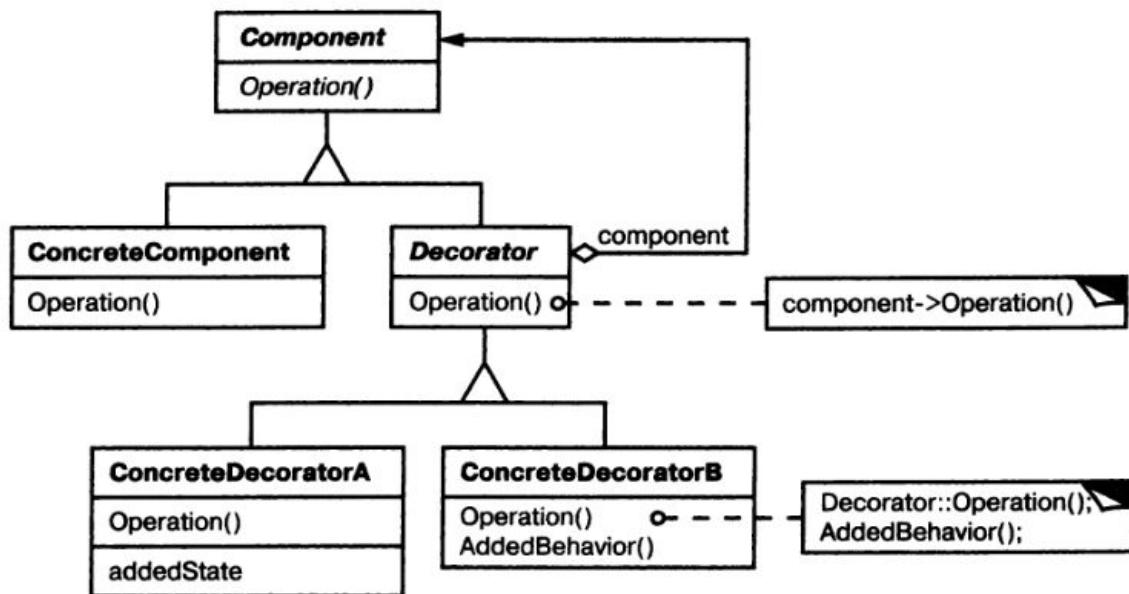
Паттерн приспособленец позволяет разделять компоненты, но ссылаясь на своих родителей они уже не могут.

Итератор можно использовать для обхода составных объектов.

Посетитель локализует операции и поведение, которые в противном случае пришлось бы распределять между классами Composite и Leaf.

27. Паттерн «Декоратор».

Структура



Участники

- **Component** (*VisualComponent*) – компонент:
 - определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности;
- **ConcreteComponent** (*TextView*) – конкретный компонент:
 - определяет объект, на который возлагаются дополнительные обязанности;
- **Decorator** – декоратор:
 - хранит ссылку на объект **Component** и определяет интерфейс, соответствующий интерфейсу **Component**;
- **ConcreteDecorator** (*BorderDecorator*, *ScrollDecorator*) – конкретный декоратор:
 - возлагает дополнительные обязанности на компонент.

Назначение

:

Назначение

Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Мотивация

Применимость

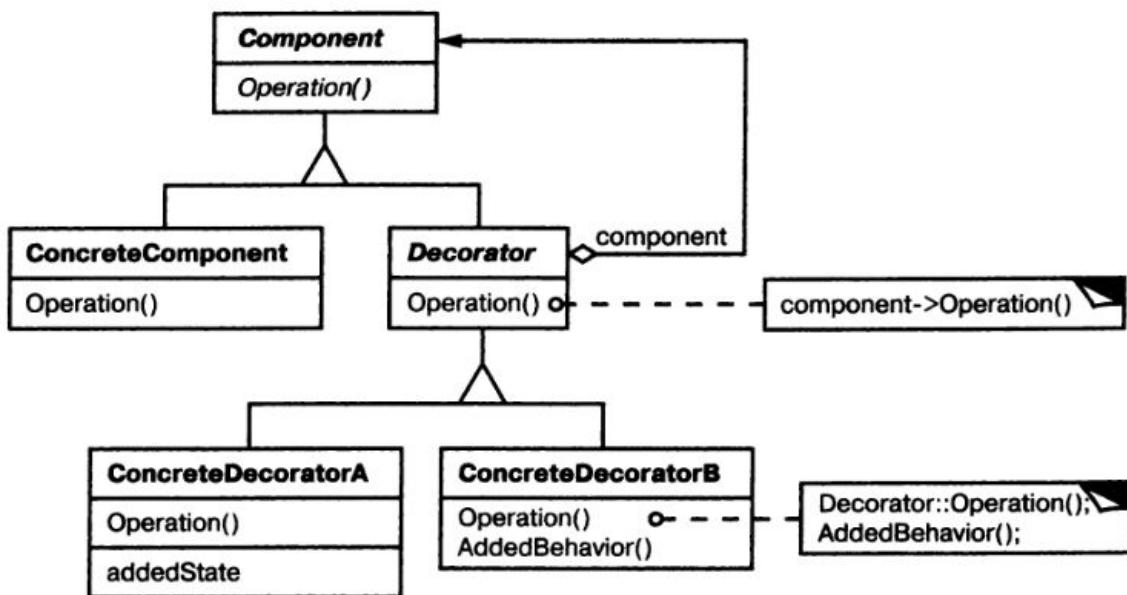
Применимость

Используйте паттерн декоратор:

- для динамического, прозрачного для клиентов добавления обязанностей объектам;
- для реализации обязанностей, которые могут быть сняты с объекта;
- когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно. Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к комбинаторному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.

Структура

Структура



Отношения

Отношения

Decorator переадресует запросы объекту **Component**. Может выполнять и дополнительные операции до и после переадресации.

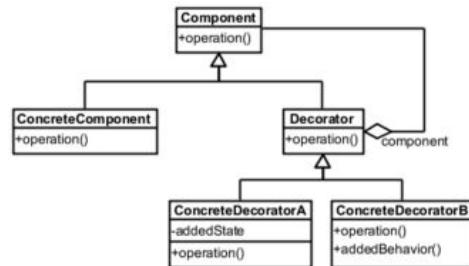
Результаты

- ▶ Динамическое добавление (и удаление) обязанностей объектов
 - ▶ Большая гибкость, чем у наследования
 - ▶ Позволяет избежать перегруженных функциональностью базовых классов
 - ▶ Много мелких объектов
- *декоратор и его компонент не идентичны.* Декоратор действует как прозрачное обрамление. Но декорированный компонент все же не идентичен исходному. При использовании декораторов это следует иметь в виду;

Детали реализации

“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



Родственные паттерны

Родственные паттерны

Адаптер: если декоратор изменяет только обязанности объекта, но не его интерфейс, то адаптер придает объекту совершенно новый интерфейс.

Паттерн Facade



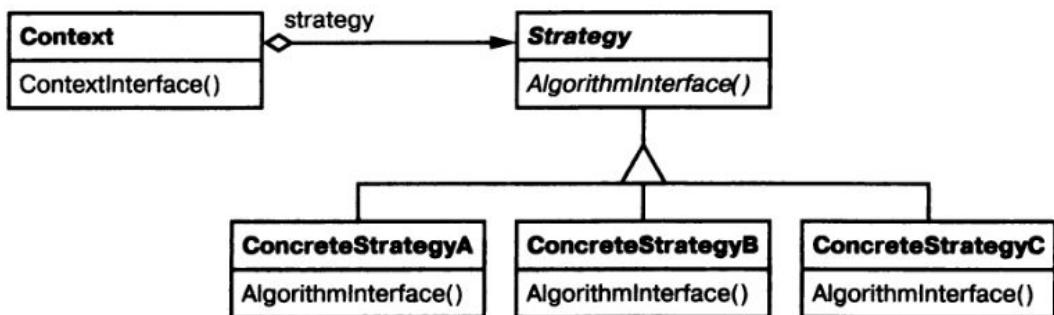
183

Компоновщик: декоратор можно считать вырожденным случаем составного объекта, у которого есть только один компонент. Однако декоратор добавляет новые обязанности, агрегирование объектов не является его целью.

Стратегия: декоратор позволяет изменить внешний облик объекта, стратегия – его внутреннее содержание. Это два взаимодополняющих способа изменения объекта.

28. Паттерн «Стратегия».

Структура



Участники

- **Strategy** (Compositor) – стратегия:
 - объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy;
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor) – конкретная стратегия:
 - реализует алгоритм, использующий интерфейс, объявленный в классе Strategy;
- **Context** (Composition) – контекст:
 - конфигурируется объектом класса ConcreteStrategy;
 - хранит ссылку на объект класса Strategy;
 - может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.

Назначение

Назначение

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

Мотивация

Мотивация

Существует много алгоритмов для разбиения текста на строки. Жестко «зашивать» все подобные алгоритмы в классы, которые в них нуждаются, нежелательно по нескольким причинам:

Паттерн Strategy

 301

- клиент, которому требуется алгоритм разбиения на строки, усложняется при включении в него соответствующего кода. Таким образом, клиенты становятся более громоздкими, а сопровождать их труднее, особенно если нужно поддерживать сразу несколько алгоритмов;
- в зависимости от обстоятельств стоит применять тот или иной алгоритм. Не хотелось бы поддерживать несколько алгоритмов разбиения на строки, если мы не будем ими пользоваться;
- если разбиение на строки – неотъемлемая часть клиента, то задача добавления новых и модификации существующих алгоритмов усложняется.

Всех этих проблем можно избежать, если определить классы, инкапсулирующие различные алгоритмы разбиения на строки. Инкапсулированный таким образом алгоритм называется *стратегией*.

Применимость

Применимость

Используйте паттерн **стратегия**, когда:

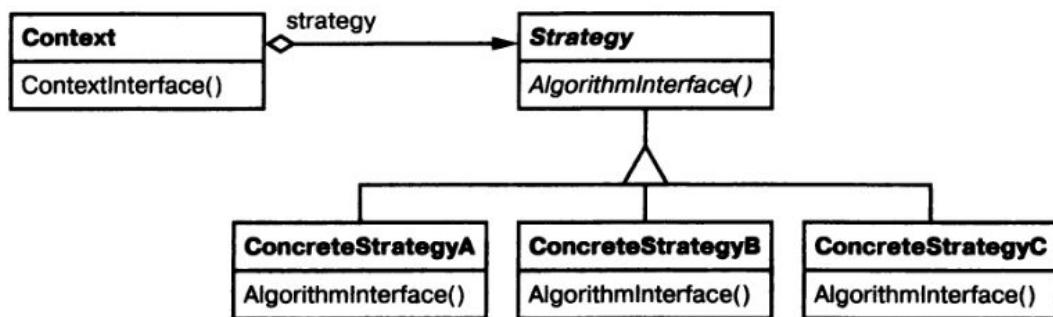
- имеется много родственных классов, отличающихся только поведением. Стратегия позволяет сконфигурировать класс, задав одно из возможных поведений;

 02

Паттерны поведения

- вам нужно иметь несколько разных вариантов алгоритма. Например, можно определить два варианта алгоритма, один из которых требует больше времени, а другой – больше памяти. Стратегии разрешается применять, когда варианты алгоритмов реализованы в виде иерархии классов [HO87];
- в алгоритме содержатся данные, о которых клиент не должен «знать». Используйте паттерн **стратегия**, чтобы не раскрывать сложные, специфичные для алгоритма структуры данных;
- в классе определено много поведений, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Структура



Участники

- **Strategy** (Compositor) – стратегия:
 - объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс **Context** пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе **ConcreteStrategy**;
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor) – конкретная стратегия:
 - реализует алгоритм, использующий интерфейс, объявленный в классе **Strategy**;
- **Context** (Composition) – контекст:
 - конфигурируется объектом класса **ConcreteStrategy**;
 - хранит ссылку на объект класса **Strategy**;
 - может определять интерфейс, который позволяет объекту **Strategy** получить доступ к данным контекста.

Отношения

Отношения

- классы **Strategy** и **Context** взаимодействуют для реализации выбранного алгоритма. Контекст может передать стратегии все необходимые алгоритму данные в момент его вызова. Вместо этого контекст может позволить обращаться к своим операциям в нужные моменты, передав ссылку на самого себя операциям класса **Strategy**;
- контекст переадресует запросы своих клиентов объекту-стратегии. Обычно клиент создает объект **ConcreteStrategy** и передает его контексту, после

чего клиент «общается» исключительно с контекстом. Часто в распоряжении клиента находится несколько классов **ConcreteStrategy**, которые он может выбирать.

Результаты

Результаты

У паттерна стратегия есть следующие достоинства и недостатки:

- ***семейства родственных алгоритмов.*** Иерархия классов `Strategy` определяет семейство алгоритмов или поведений, которые можно повторно использовать в разных контекстах. Наследование позволяет выделить общую для всех алгоритмов функциональность;
- ***альтернатива порождению подклассов.*** Наследование поддерживает многообразие алгоритмов или поведений. Можно напрямую породить от `Context` подклассы с различными поведениями. Но при этом поведение жестко «зашивается» в класс `Context`. Вот почему реализации алгоритма и контекста смешиваются, что затрудняет понимание, сопровождение и расширение контекста. Кроме того, заменить алгоритм динамически уже не удастся. В результате вы получите множество родственных классов, отличающихся только алгоритмом или поведением. Инкапсуляции алгоритма в отдельный класс `Strategy` позволяют изменять его независимо от контекста;
- ***с помощью стратегий можно избавиться от условных операторов.*** Благодаря паттерну стратегия удается отказаться от условных операторов при выборе нужного поведения. Когда различные поведения помещаются в один класс, трудно выбрать нужное без применения условных операторов. Инкапсуляция же каждого поведения в отдельный класс `Strategy` решает эту проблему.

Если код содержит много условных операторов, то часто это признак того, что нужно применить паттерн **стратегия**;

- **выбор реализации.** Стратегии могут предлагать различные реализации *одного и того же поведения*. Клиент вправе выбирать подходящую стратегию в зависимости от своих требований к быстродействию и памяти;
- **клиенты должны «знать» о различных стратегиях.** Потенциальный недостаток этого паттерна в том, что для выбора подходящей стратегии клиент должен понимать, чем отличаются разные стратегии. Поэтому наверняка придется раскрыть клиенту некоторые особенности реализации. Отсюда следует, что паттерн **стратегия** стоит применять лишь тогда, когда различия в поведении имеют значение для клиента;
- **обмен информацией между стратегией и контекстом.** Интерфейс класса **Strategy** разделяется всеми подклассами **ConcreteStrategy** – неважно, сложна или тривиальна их реализация. Поэтому вполне вероятно, что некоторые стратегии не будут пользоваться всей передаваемой им информацией, особенно простые. Это означает, что в отдельных случаях контекст создаст и проинициализирует параметры, которые никому не нужны. Если возникнет проблема, то между классами **Strategy** и **Context** придется установить более тесную связь;
- **увеличение числа объектов.** Применение стратегий увеличивает число объектов в приложении. Иногда эти издержки можно сократить, если реализовать стратегии в виде объектов без состояния, которые могут разделяться несколькими контекстами. Остаточное состояние хранится в самом контексте и передается при каждом обращении к объекту-стратегии. Разделяемые стратегии не должны сохранять состояние между вызовами. В описании паттерна приспособленец этот подход обсуждается более подробно.

Реализация

Реализация

Рассмотрим следующие вопросы реализации:

- **определение интерфейсов классов Strategy и Context.** Интерфейсы классов *Strategy* и *Context* могут обеспечить объекту класса *ConcreteStrategy* эффективный доступ к любым данным контекста, и наоборот.

Например, *Context* передает данные в виде параметров операциям класса *Strategy*. Это разрывает тесную связь между контекстом и стратегией. При этом не исключено, что контекст будет передавать данные, которые стратегии не нужны.

Другой метод – передать контекст в качестве аргумента, в таком случае стратегия будет запрашивать у него данные, или, например, сохранить ссылку на свой контекст, так что передавать вообще ничего не придется. И в том, и в другом случаях стратегия может запрашивать только ту информацию, которая реально необходима. Но тогда в контексте должен быть определен более развитый интерфейс к своим данным, что несколько усиливает связанность классов *Strategy* и *Context*.

Какой подход лучше, зависит от конкретного алгоритма и требований, которые он предъявляет к данным;

- *стратегии как параметры шаблона.* В C++ для конфигурирования класса стратегией можно использовать шаблоны. Этот способ хорош, только если стратегия определяется на этапе компиляции и ее не нужно менять во время выполнения. Тогда конфигурируемый класс (например, Context) определяется в виде шаблона, для которого класс Strategy является параметром:

```
template <class AStrategy>
class Context {
    void Operation() { theStrategy.DoAlgorithm(); }
    // ...
private:
    AStrategy theStrategy;
};
```

Затем этот класс конфигурируется классом Strategy в момент инстанцирования:

```
class MyStrategy {
public:
    void DoAlgorithm();
};

Context<MyStrategy> aContext;
```

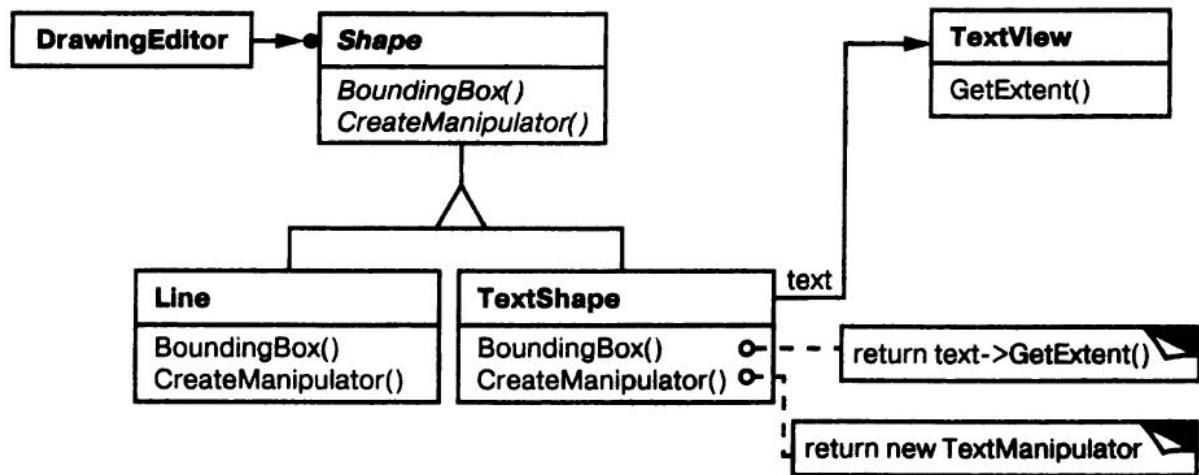
При использовании шаблонов отпадает необходимость в абстрактном классе для определения интерфейса Strategy. Кроме того, передача стратегии в виде параметра шаблона позволяет статически связать стратегию с контекстом, вследствие чего повышается эффективность программы;

- *объекты-стратегии можно не задавать.* Класс Context разрешается упростить, если для него отсутствие какой бы то ни было стратегии является нормой. Прежде чем обращаться к объекту Strategy, объект Context проверяет наличие стратегии. Если да, то работа продолжается как обычно, в противном случае контекст реализует некое поведение по умолчанию. Достоинство такого подхода в том, что клиентам вообще не нужно иметь дело со стратегиями, если их устраивает поведение по умолчанию.

Отношение 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.
- **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.
- **Стратегия** меняет поведение объекта «изнутри», а **Декоратор** изменяет его «снаружи».
- **Шаблонный метод** использует наследование, чтобы расширять части алгоритма. **Стратегия** использует делегирование, чтобы изменять выполняемые алгоритмы на лету. **Шаблонный метод** работает на уровне классов. **Стратегия** позволяет менять логику отдельных объектов.
- **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.

29. Паттерн «Адаптер».



Участники

- **Target (Shape)** – целевой:
 - определяет зависящий от предметной области интерфейс, которым пользуется Client;
- **Client (DrawingEditor)** – клиент:
 - вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;
- **Adaptee (TextView)** – адаптируемый:
 - определяет существующий интерфейс, который нуждается в адаптации;
- **Adapter (TextShape)** – адаптер:
 - адаптирует интерфейс Adaptee к интерфейсу Target.

Назначение

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

Мотивация

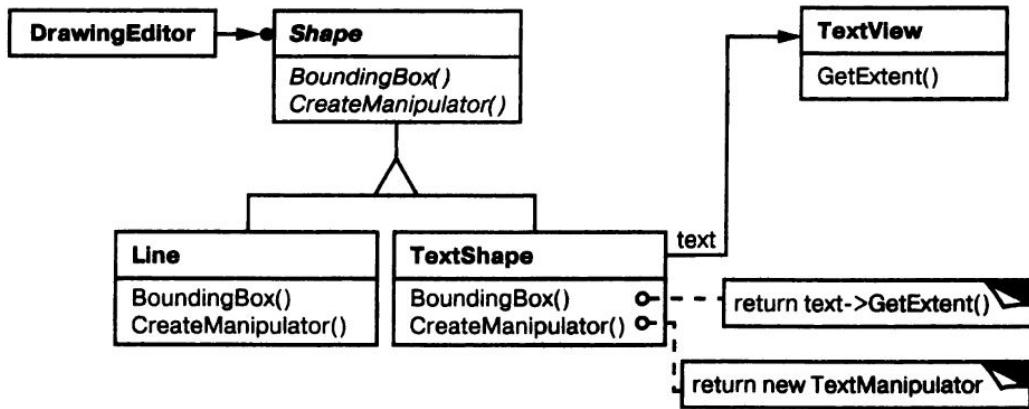
Иногда класс из инструментальной библиотеки, спроектированный для повторного использования, не удается использовать только потому, что его интерфейс не соответствует тому, который нужен конкретному приложению.

Рассмотрим, например, графический редактор, благодаря которому пользователи могут рисовать на экране графические элементы (линии, многоугольники, текст и т.д.) и организовывать их в виде картинок и диаграмм. Основной абстракцией графического редактора является графический объект, который имеет изменяемую форму и изображает сам себя. Интерфейс графических объектов определен абстрактным классом `Shape`. Редактор определяет подкласс класса `Shape` для каждого вида графических объектов: `LineShape` для прямых, `PolygonShape` для многоугольников и т.д.

Классы для элементарных геометрических фигур, например `LineShape` и `PolygonShape`, реализовать сравнительно просто, поскольку заложенные в них возможности рисования и редактирования крайне ограничены. Но подкласс `TextShape`, умеющий отображать и редактировать текст, уже значительно сложнее, поскольку даже для простейших операций редактирования текста нужно нетривиальным образом обновлять экран и управлять буферами. В то же время, возможно, существует уже готовая библиотека для разработки пользовательских интерфейсов, которая предоставляет развитый класс `TextView`, позволяющий отображать и редактировать текст. В идеале мы хотели бы повторно использовать `TextView` для реализации `TextShape`, но библиотека разрабатывалась без учета классов `Shape`, поэтому заставить объекты `TextView` и `Shape` работать совместно не удается.

Так каким же образом существующие и независимо разработанные классы вроде `TextView` могут работать в приложении, которое спроектировано под другой, несовместимый интерфейс? Можно было бы так изменить интерфейс класса `TextView`, чтобы он соответствовал интерфейсу `Shape`, только для этого нужен исходный код. Но даже если он доступен, то вряд ли разумно изменять `TextView`; библиотека не должна приспосабливаться к интерфейсам каждого конкретного приложения.

Вместо этого мы могли бы определить класс `TextShape` так, что он будет *адаптировать* интерфейс `TextView` к интерфейсу `Shape`. Это допустимо сделать двумя способами: наследуя интерфейс от `Shape`, а реализацию от `TextView`; включив экземпляр `TextView` в `TextShape` и реализовав `TextShape` в терминах интерфейса `TextView`. Два данных подхода соответствуют вариантам паттерна *адаптер* в его классовой и объектной ипостасях. Класс `TextShape` мы будем называть *адаптером*.



Применимость

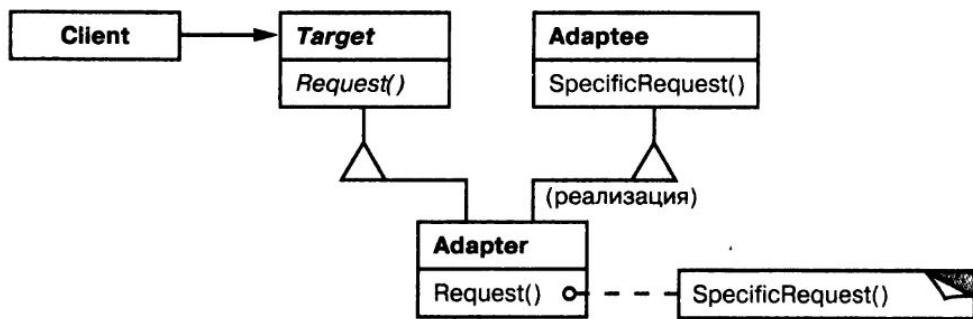
Применяйте паттерн *адаптер*, когда:

- хотите использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
- собираетесь создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;
- (*только для адаптера объектов!*) нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспособливать интерфейс их общего родительского класса.

Структура

Структура

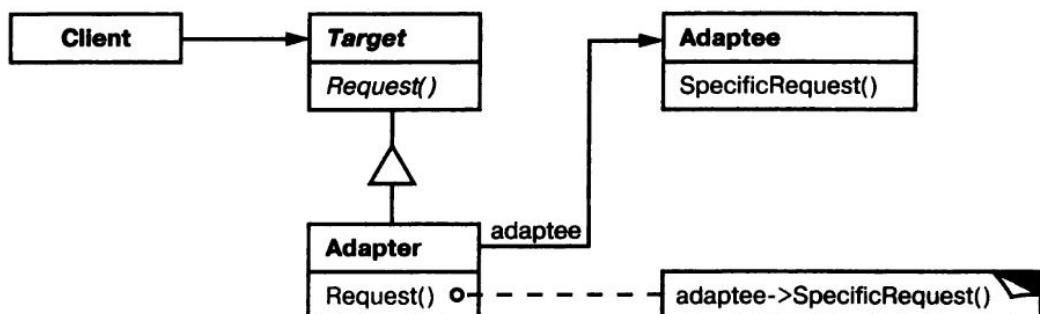
Адаптер класса использует множественное наследование для адаптации одного интерфейса к другому.



Адаптер объекта применяет композицию объектов.

4

Структурные паттерны



Участники

- **Target (Shape)** – целевой:
 - определяет зависящий от предметной области интерфейс, которым пользуется Client;
- **Client (DrawingEditor)** – клиент:
 - вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;
- **Adaptee (TextView)** – адаптируемый:
 - определяет существующий интерфейс, который нуждается в адаптации;
- **Adapter (TextShape)** – адаптер:
 - адаптирует интерфейс Adaptee к интерфейсу Target.

Отношения

Клиенты вызывают операции экземпляра адаптера Adapter. В свою очередь адаптер вызывает операции адаптируемого объекта или класса Adaptee, который и выполняет запрос.

Результаты

Результаты применения адаптеров объектов и классов различны. Адаптер класса:

- адаптирует Adaptee к Target, перепоручая действия конкретному классу Adaptee. Поэтому данный паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы;
- позволяет адаптеру Adapter заместить некоторые операции адаптируемого класса Adaptee, так как Adapter есть не что иное, как подкласс Adaptee;
- вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

Адаптер объектов:

- позволяет одному адаптеру Adapter работать со многими адаптируемыми объектами Adaptee, то есть с самим Adaptee и его подклассами (если такие имеются). Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;
- затрудняет замещение операций класса Adaptee. Для этого потребуется породить от Adaptee подкласс и заставить Adapter ссылаться на этот подкласс, а не на сам Adaptee.

При реализации задуматься о:

Ниже приведены вопросы, которые следует рассмотреть, когда вы решаете применить паттерн адаптер:

- *объем работы по адаптации.* Адаптеры сильно отличаются по тому объему работы, который необходим для адаптации интерфейса Adaptee к интерфейсу Target. Это может быть как простейшее преобразование, например изменение имен операций, так и поддержка совершенно другого набора операций. Объем работы зависит от того, насколько сильно отличаются друг от друга интерфейсы целевого и адаптируемого классов;

??? сменный адаптер -- хватит ли “использования абстрактных операций”?

□ *сменные адаптеры*. Рассмотрим три способа реализации сменных адаптеров для описанного выше виджета TreeDisplay, который может автоматически отображать иерархические структуры.

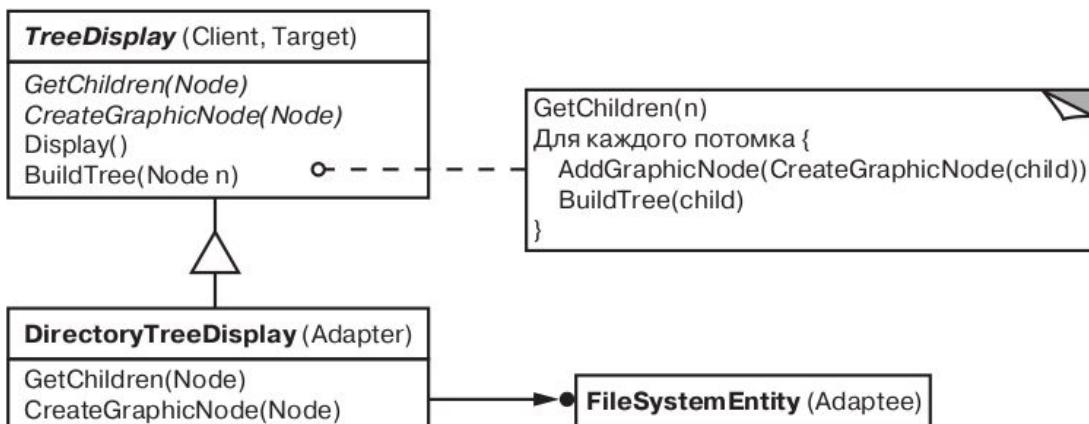
Первый шаг, общий для всех трех реализаций, – найти «узкий» интерфейс для Adaptee, то есть наименьшее подмножество операций, позволяющее выполнить адаптацию. «Узкий» интерфейс, состоящий всего из пары итераций, легче адаптировать, чем интерфейс из нескольких десятков операций. Для TreeDisplay адаптации подлежит любая иерархическая структура. Минимальный интерфейс мог бы включать всего две операции: одна определяет графическое представление узла в иерархической структуре, другая – доступ к потомкам узла.

«Узкий» интерфейс приводит к трем подходам к реализации:

- *использование абстрактных операций*. Определим в классе TreeDisplay абстрактные операции, которые соответствуют «узкому» интерфейсу класса Adaptee. Подклассы должны реализовывать эти абстрактные операции

и адаптировать иерархически структурированный объект. Например, подкласс DirectoryTreeDisplay при их реализации будет осуществлять доступ к структуре каталогов файловой системы.

DirectoryTreeDisplay специализирует узкий интерфейс таким образом, чтобы он мог отображать структуру каталогов, составленную из объектов FileSystemEntity;

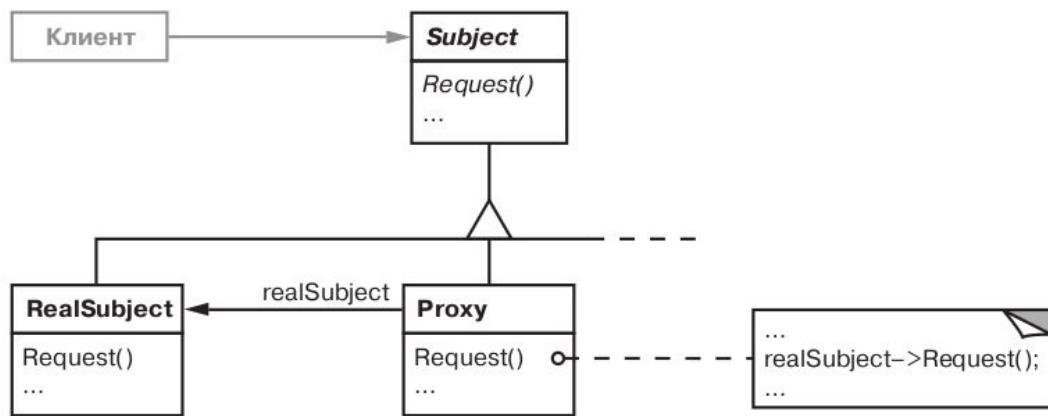


↔ Отношения с другими паттернами

- **Мост** проектируют загодя, чтобы развивать большие части приложения отдельно друг от друга. **Адаптер** применяется постфактум, чтобы заставить несовместимые классы работать вместе.
- **Адаптер** меняет интерфейс существующего объекта. **Декоратор** улучшает другой объект без изменения его интерфейса. Причём Декоратор поддерживает рекурсивную вложенность, чего не скажешь об Адаптере.
- **Адаптер** предоставляет классу альтернативный интерфейс. **Декоратор** предоставляет расширенный интерфейс. **Заместитель** предоставляет тот же интерфейс.
- **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.
- **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.

30. Паттерн «Заместитель».

Структура



Участники

□ **Proxy** (*ImageProxy*) – заместитель:

- хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту. Объект класса **Proxy** может обращаться к объекту класса **Subject**, если интерфейсы классов **RealSubject** и **Subject** одинаковы;
- предоставляет интерфейс, идентичный интерфейсу **Subject**, так что заместитель всегда может быть подставлен вместо реального субъекта;
- контролирует доступ к реальному субъекту и может отвечать за его создание и удаление;
- прочие обязанности зависят от вида заместителя:
 - *удаленный заместитель* отвечает за кодирование запроса и его аргументов и отправление закодированного запроса реальному субъекту в другом адресном пространстве;
 - *виртуальный заместитель* может кэшировать дополнительную информацию о реальном субъекте, чтобы отложить его создание. Например, класс *ImageProxy* из раздела «Мотивация» кэширует размеры реального изображения;
 - *защищающий заместитель* проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права;

□ **Subject** (*Graphic*) – субъект:

- определяет общий для **RealSubject** и **Proxy** интерфейс, так что класс **Proxy** можно использовать везде, где ожидается **RealSubject**;

□ **RealSubject** (*Image*) – реальный субъект:

- определяет реальный объект, представленный заместителем.

Отношения

Proxy при необходимости переадресует запросы объекту **RealSubject**. Детали зависят от вида заместителя.

Назначение

Является суррогатом другого объекта и контролирует доступ к нему.

Мотивация

Разумно управлять доступом к объекту, поскольку тогда можно отложить расходы на создание и инициализацию до момента, когда объект действительно понадобится. Рассмотрим редактор документов, который допускает встраивание в документ графических объектов. Затраты на создание некоторых таких объектов, например больших растровых изображений, могут быть весьма значительны. Но документ должен открываться быстро, поэтому следует избегать создания всех «тяжелых» объектов на стадии открытия (да и вообще это излишне, поскольку не все они будут видны одновременно).

В связи с такими ограничениями кажется разумным создавать «тяжелые» объекты *по требованию*. Это означает «когда изображение становится видимым». Но что поместить в документ вместо изображения? И как, не усложняя реализации редактора, скрыть то, что изображение создается по требованию? Например, оптимизация не должна отражаться на коде, отвечающем за рисование и форматирование.

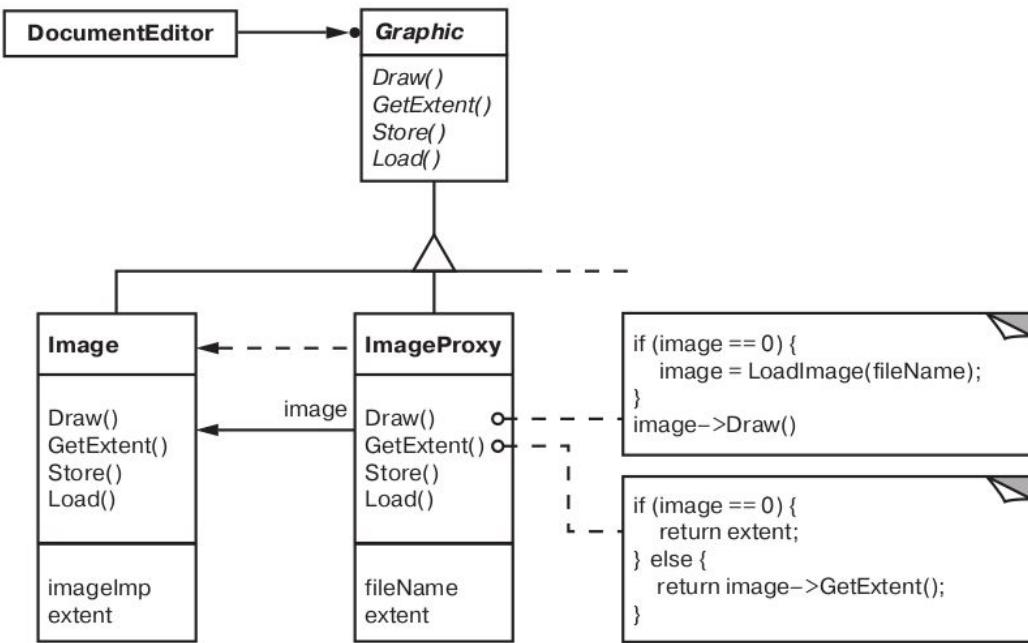
Решение состоит в том, чтобы использовать другой объект – *заместитель* изображения, который временно подставляется вместо реального изображения. Заместитель ведет себя точно так же, как само изображение, и выполняет при необходимости его инстанцирование.



Заместитель создает настоящее изображение, только если редактор документа вызовет операцию `Draw`. Все последующие запросы заместитель переадресует непосредственно изображению. Поэтому после создания изображения он должен сохранить ссылку на него.

Предположим, что изображения хранятся в отдельных файлах. В таком случае мы можем использовать имя файла как ссылку на реальный объект. Заместитель хранит также размер изображения, то есть длину и ширину. «Зная» ее, заместитель может отвечать на запросы форматера о своем размере, не инстанцируя изображение.

На следующей диаграмме классов этот пример показан более подробно.



Редактор документов получает доступ к встроенным изображениям только через интерфейс, определенный в абстрактном классе **Graphic**. **ImageProxy** – это класс для представления изображений, создаваемых по требованию. В **ImageProxy** хранится имя файла, играющее роль ссылки на изображение, которое находится на диске. Имя файла передается конструктору класса **ImageProxy**.

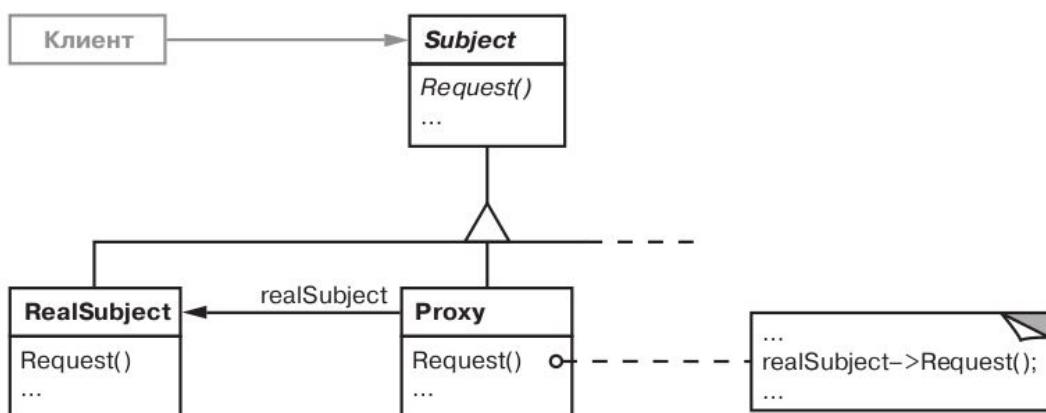
В объекте **ImageProxy** находятся также ограничивающий прямоугольник изображения и ссылка на экземпляр реального объекта **Image**. Ссылка остается недействительной, пока заместитель не инстанцирует реальное изображение. Операцией **Draw** гарантируется, что изображение будет создано до того, как заместитель переадресует ему запрос. Операция **GetExtent** переадресует запрос изображению, только если оно уже инстанцировано; в противном случае **ImageProxy** возвращает размеры, которые хранит сам.

Применимость

Паттерн **заместитель** применим во всех случаях, когда возникает необходимость сослаться на объект более изощренно, чем это возможно, если использовать простой указатель. Вот несколько типичных ситуаций, где **заместитель** оказывается полезным:

- **удаленный заместитель** предоставляет локального представителя вместо объекта, находящегося в другом адресном пространстве. В системе NEXTSTEP [Add94] для этой цели применяется класс NXProxy. Заместителя такого рода Джеймс Коплиен [Cop92] называет «послом»;
- **виртуальный заместитель** создает «тяжелые» объекты по требованию. Примером может служить класс ImageProxy, описанный в разделе «Мотивация»;
- **защищающий заместитель** контролирует доступ к исходному объекту. Такие заместители полезны, когда для разных объектов определены различные права доступа. Например, в операционной системе Choices [CIRM93] объекты KernelProxy ограничивают права доступа к объектам операционной системы;
- **«умная» ссылка** – это замена обычного указателя. Она позволяет выполнить дополнительные действия при доступе к объекту. К типичным применением такой ссылки можно отнести:
 - подсчет числа ссылок на реальный объект, с тем чтобы занимаемую им память можно было освободить автоматически, когда не останется ни одной ссылки (такие ссылки называют еще «умными» указателями [Ede92]);
 - загрузку объекта в память при первом обращении к нему;
 - проверку и установку блокировки на реальный объект при обращении к нему, чтобы никакой другой объект не смог в это время изменить его.

Структура



Вот как может выглядеть диаграмма объектов для структуры с заместителем во время выполнения.



Участники

□ **Proxy** (*ImageProxy*) – заместитель:

- хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту. Объект класса *Proxy* может обращаться к объекту класса *Subject*, если интерфейсы классов *RealSubject* и *Subject* одинаковы;
- предоставляет интерфейс, идентичный интерфейсу *Subject*, так что заместитель всегда может быть подставлен вместо реального субъекта;
- контролирует доступ к реальному субъекту и может отвечать за его создание и удаление;
- прочие обязанности зависят от вида заместителя:
 - *удаленный заместитель* отвечает за кодирование запроса и его аргументов и отправление закодированного запроса реальному субъекту в другом адресном пространстве;
 - *виртуальный заместитель* может кэшировать дополнительную информацию о реальном субъекте, чтобы отложить его создание. Например, класс *ImageProxy* из раздела «Мотивация» кэширует размеры реального изображения;
 - *защищающий заместитель* проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права;

□ **Subject** (*Graphic*) – субъект:

- определяет общий для *RealSubject* и *Proxy* интерфейс, так что класс *Proxy* можно использовать везде, где ожидается *RealSubject*;

□ **RealSubject** (*Image*) – реальный субъект:

- определяет реальный объект, представленный заместителем.

Отношения

Proxy при необходимости переадресует запросы объекту *RealSubject*. Детали зависят от вида заместителя.

Результаты

С помощью паттерна **заместитель** при доступе к объекту вводится дополнительный уровень косвенности. У этого подхода есть много вариантов в зависимости от вида заместителя:

- удаленный заместитель может скрыть тот факт, что объект находится в другом адресном пространстве;
- виртуальный заместитель может выполнять оптимизацию, например создание объекта по требованию;
- защищающий заместитель и «умная» ссылка позволяют решать дополнительные задачи при доступе к объекту.

Copy on write

Есть еще одна оптимизация, которую паттерн **заместитель** иногда скрывает от клиента. Она называется *копированием при записи* (copy-on-write) и имеет много общего с созданием объекта по требованию. Копирование большого и сложного объекта – очень дорогая операция. Если копия не модифицировалась, то нет смысла эту цену платить. Если отложить процесс копирования, применив заместитель, то можно быть уверенным, что эта операция произойдет только тогда, когда он действительно был изменен.

Чтобы во время записи можно было копировать, необходимо подсчитывать ссылки на субъект. Копирование заместителя просто увеличивает счетчик ссылок. И только тогда, когда клиент запрашивает операцию, изменяющую субъект, заместитель действительно выполняет копирование. Одновременно заместитель должен уменьшить счетчик ссылок. Когда счетчик ссылок становится равным нулю, субъект уничтожается.

Копирование при записи может существенно уменьшить плату за копирование «тяжелых» субъектов.

Пример для Copy On Write:

```
std::string x("Hello");
std::string y = x; // x and y use the same buffer
y += ", World!";
// now y uses a different buffer
// x still uses the same old buffer
```

Реализация

- *заместителю не всегда должен быть известен тип реального объекта.* Если класс Proxy может работать с субъектом только через его абстрактный интерфейс, то не нужно создавать Proxy для каждого класса реального субъекта RealSubject; заместитель может обращаться к любому из них единообразно. Но если заместитель должен инстанцировать реальных субъектов (как обстоит дело в случае виртуальных заместителей), то знание конкретного класса обязательно.

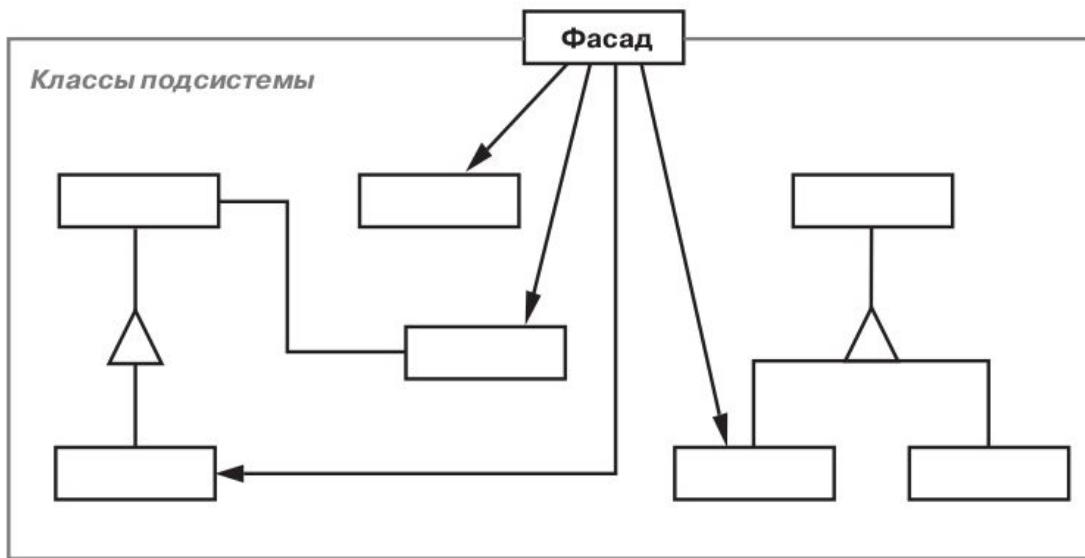
К проблемам реализации можно отнести и решение вопроса о том, как обращаться к еще не инстанцированному субъекту. Некоторые заместители должны обращаться к своим субъектам вне зависимости от того, где они находятся – диске или в памяти. Это означает, что нужно использовать какую-то форму не зависящих от адресного пространства идентификаторов объектов. В разделе «Мотивация» для этой цели использовалось имя файла.

???

Перегрузку оператора доступа к членам в C++

31. Паттерн «Фасад».

Структура



Участники

- **Facade (Compiler)** – фасад:
 - «знает», каким классам подсистемы адресовать запрос;
 - делегирует запросы клиентов подходящим объектам внутри подсистемы;
- **Классы подсистемы** (Scanner, Parser, ProgramNode и т.д.):
 - реализуют функциональность подсистемы;
 - выполняют работу, порученную объектом Facade;
 - ничего не «знают» о существовании фасада, то есть не хранят ссылок на него.

Назначение

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

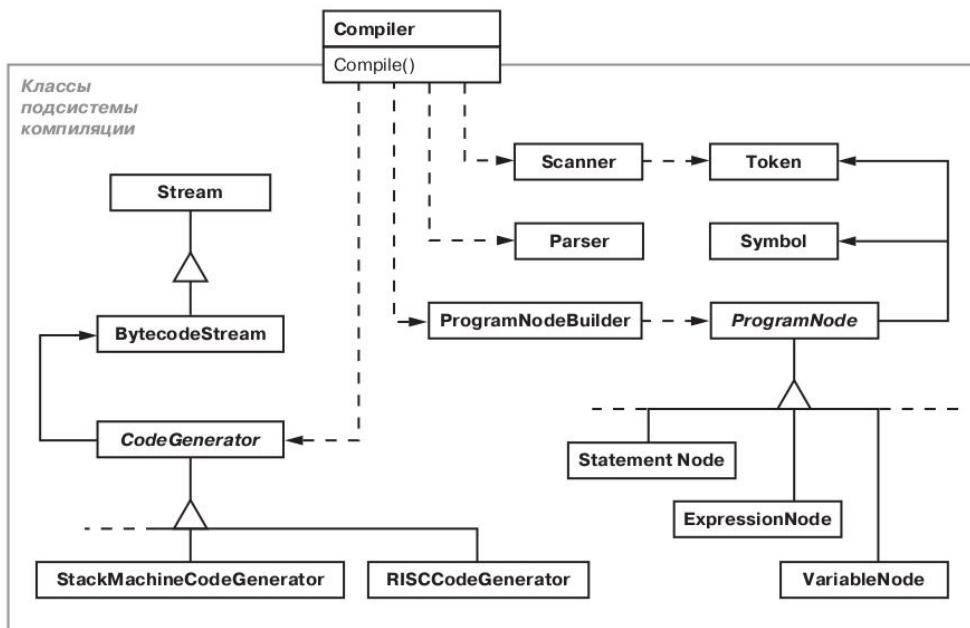
Мотивация

Мотивация

Разбиение на подсистемы облегчает проектирование сложной системы в целом. Общая цель всякого проектирования – свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними. Один из способов решения этой задачи – введение объекта **фасад**, предоставляющий единый упрощенный интерфейс к более сложным системным средствам.



Рассмотрим, например, среду программирования, которая дает приложениям доступ к подсистеме компиляции. В этой подсистеме имеются такие классы, как Scanner (лексический анализатор), Parser (синтаксический анализатор), ProgramNode (узел программы), BytecodeStream (поток байтовых кодов) и ProgramNodeBuilder (строитель узла программы). Все вместе они составляют компилятор. Некоторым специализированным приложениям, возможно, понадобится прямой доступ к этим классам. Но для большинства клиентов компилятора такие детали, как синтаксический разбор и генерация кода, обычно не нужны; им просто требуется откомпилировать некоторую программу. Для таких клиентов применение мощного, но низкоуровневого интерфейса подсистемы компиляции только усложняет задачу.



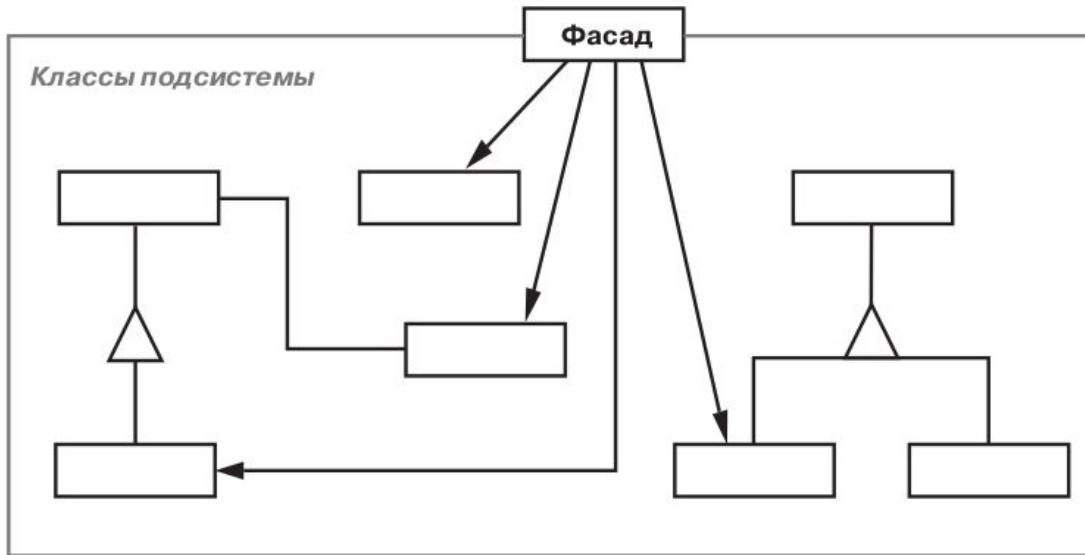
Чтобы предоставить интерфейс более высокого уровня, изолирующий клиента от этих классов, в подсистему компиляции включен также класс `Compiler` (компилятор). Он определяет унифицированный интерфейс ко всем возможностям компилятора. Класс `Compiler` выступает в роли фасада: предлагает простой интерфейс к более сложной подсистеме. Он «склеивает» классы, реализующие функциональность компилятора, но не скрывает их полностью. Благодаря фасаду компилятора работа большинства программистов облегчается. При этом те, кому нужен доступ к средствам низкого уровня, не лишаются его.

Применимость

Используйте паттерн **фасад**, когда:

- хотите предоставить простой интерфейс к сложной подсистеме. Часто подсистемы усложняются по мере развития. Применение большинства паттернов приводит к появлению меньших классов, но в большем количестве. Такую подсистему проще повторно использовать и настраивать под конкретные нужды, но вместе с тем применять подсистему без настройки становится труднее. **Фасад** предлагает некоторый вид системы по умолчанию, устраивающий большинство клиентов. И лишь те объекты, которым нужны более широкие возможности настройки, могут обратиться напрямую к тому, что находится за фасадом;
- между клиентами и классами реализации абстракции существует много зависимостей. **Фасад** позволит отделить подсистему как от клиентов, так и от других подсистем, что, в свою очередь, способствует повышению степени независимости и переносимости;
- вы хотите разложить подсистему на отдельные слои. Используйте фасад для определения точки входа на каждый уровень подсистемы. Если подсистемы зависят друг от друга, то зависимость можно упростить, разрешив подсистемам обмениваться информацией только через фасады.

Структура



Участники

- **Facade (Compiler)** – фасад:
 - «знает», каким классам подсистемы адресовать запрос;
 - делегирует запросы клиентов подходящим объектам внутри подсистемы;
- **Классы подсистемы** (Scanner, Parser, ProgramNode и т.д.):
 - реализуют функциональность подсистемы;
 - выполняют работу, порученную объектом Facade;
 - ничего не «знают» о существовании фасада, то есть не хранят ссылок на него.

Отношения

Клиенты общаются с подсистемой, посыпая запросы фасаду. Он переадресует их подходящим объектам внутри подсистемы. Хотя основную работу выполняют именно объекты подсистемы, фасаду, возможно, придется преобразовать свой интерфейс в интерфейсы подсистемы.

Клиенты, пользующиеся фасадом, не имеют прямого доступа к объектам подсистемы.

Результаты

Результаты

У паттерна фасад есть следующие преимущества:

- изолирует клиентов от компонентов подсистемы, уменьшая тем самым число объектов, с которыми клиентам приходится иметь дело, и упрощая работу с подсистемой;
- позволяет ослабить связанность между подсистемой и ее клиентами. Зачастую компоненты подсистемы сильно связаны. Слабая связанность позволяет видоизменять компоненты, не затрагивая при этом клиентов. Фасады помогают разложить систему на слои и структурировать зависимости между объектами, а также избежать сложных и циклических зависимостей. Это может оказаться важным, если клиент и подсистема реализуются независимо. Уменьшение числа зависимостей на стадии компиляции чрезвычайно важно в больших системах. Хочется, конечно, чтобы время, уходящее на перекомпиляцию после изменения классов подсистемы, было минимальным. Сокращение числа зависимостей за счет фасадов может уменьшить количество нуждающихся в повторной компиляции файлов после небольшой модификации какой-нибудь важной подсистемы. Фасад может также упростить процесс переноса системы на другие платформы, поскольку уменьшается вероятность того, что в результате изменения одной подсистемы понадобится изменять и все остальные;
- фасад не препятствует приложениям напрямую обращаться к классам подсистемы, если это необходимо. Таким образом, у вас есть выбор между простотой и общностью.

Реализация

Реализация

При реализации фасада следует обратить внимание на следующие вопросы:

- *уменьшение степени связанности клиента с подсистемой.* Степень связанности можно значительно уменьшить, если сделать класс Facade абстрактным. Его конкретные подклассы будут соответствовать различным реализациям подсистемы. Тогда клиенты смогут взаимодействовать с подсистемой через интерфейс абстрактного класса Facade. Это изолирует клиентов от информации о том, какая реализация подсистемы используется.

Вместо порождения подклассов можно сконфигурировать объект Facade различными объектами подсистем. Для настройки фасада достаточно заменить один или несколько таких объектов;

- *открытые и закрытые классы подсистем.* Подсистема похожа на класс в том отношении, что у обоих есть интерфейсы и оба что-то инкапсулируют. Класс инкапсулирует состояние и операции, а подсистема – классы. И если полезно различать открытый и закрытый интерфейсы класса, то не менее разумно говорить об открытом и закрытом интерфейсах подсистемы.

Открытый интерфейс подсистемы состоит из классов, к которым имеют доступ все клиенты; закрытый интерфейс доступен только для расширения подсистемы. Класс Facade, конечно же, является частью открытого интерфейса, но это не единственная часть. Другие классы подсистемы также могут быть открытыми. Например, в системе компиляции классы Parser и Scanner – часть открытого интерфейса.

Делать классы подсистемы закрытыми иногда полезно, но это поддерживается немногими объектно-ориентированными языками. И в C++, и в Smalltalk для классов традиционно использовалось глобальное пространство имен.

Другие паттерны

Родственные паттерны

Паттерн **абстрактная фабрика** допустимо использовать вместе с фасадом, чтобы предоставить интерфейс для создания объектов подсистем способом, не зависимым от этих подсистем. **Абстрактная фабрика** может выступать и как альтернатива **фасаду**, чтобы скрыть платформенно-зависимые классы.

Паттерн **посредник** аналогичен **фасаду** в том смысле, что абстрагирует функциональность существующих классов. Однако назначение **посредника** – абстрагировать произвольное взаимодействие между «сотрудничающими» объектами. Часто он централизует функциональность, не присущую ни одному из них. Коллеги посредника обмениваются информацией именно с ним, а не напрямую между собой. Напротив, фасад просто абстрагирует интерфейс объектов подсистемы, чтобы ими было проще пользоваться. Он не определяет новой функциональности, и классам подсистемы ничего неизвестно о его существовании.

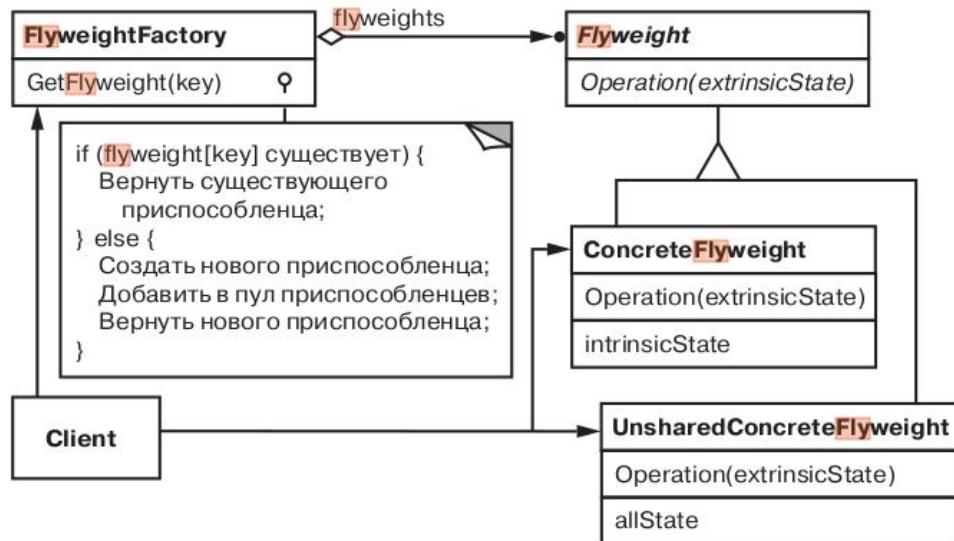
Обычно требуется только один **фасад**. Поэтому объекты фасадов часто являются **одиночками**.

↔ Отношения с другими паттернами

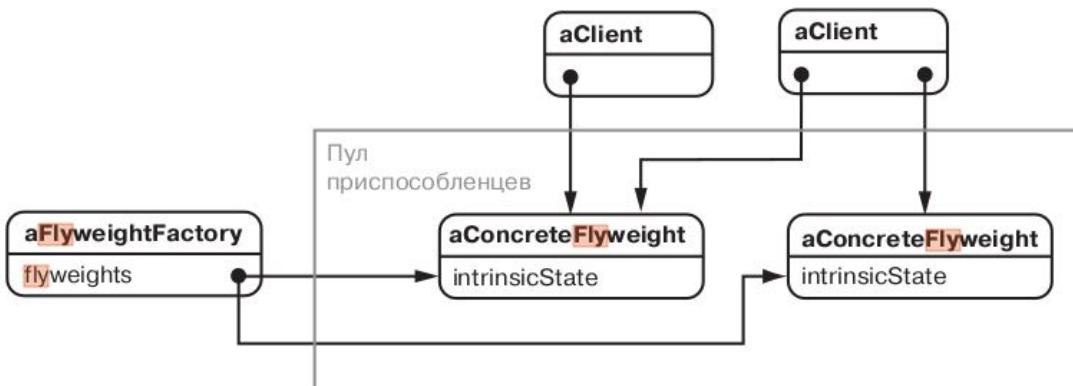
- **Фасад** задаёт новый интерфейс, тогда как **Адаптер** повторно использует старый. **Адаптер** оборачивает только один класс, а **Фасад** оборачивает целую подсистему. Кроме того, **Адаптер** позволяет двум существующим интерфейсам работать сообща, вместо того, чтобы задать полностью новый.
- **Абстрактная 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.

32. Паттерн «Приспособленец».

Структура



На следующей диаграмме показано, как приспособленцы разделяются.



Участники

- **Flyweight** (Glyph) – приспособленец:
 - объявляет интерфейс, с помощью которого приспособленцы могут получать внешнее состояние или как-то воздействовать на него;
- **ConcreteFlyweight** (Character) – конкретный приспособленец:
 - реализует интерфейс класса **Flyweight** и добавляет при необходимости внутреннее состояние. Объект класса **ConcreteFlyweight** должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть не зависящим от контекста;
- **UnsharedConcreteFlyweight** (Row, Column) – неразделяемый конкретный приспособленец:
 - не все подклассы **Flyweight** обязательно должны быть разделяемыми. Интерфейс **Flyweight** допускает разделение, но не навязывает его. Часто у объектов **UnsharedConcreteFlyweight** на некотором уровне структуры

Назначение

Использует разделение для эффективной поддержки множества мелких объектов.

Мотивация

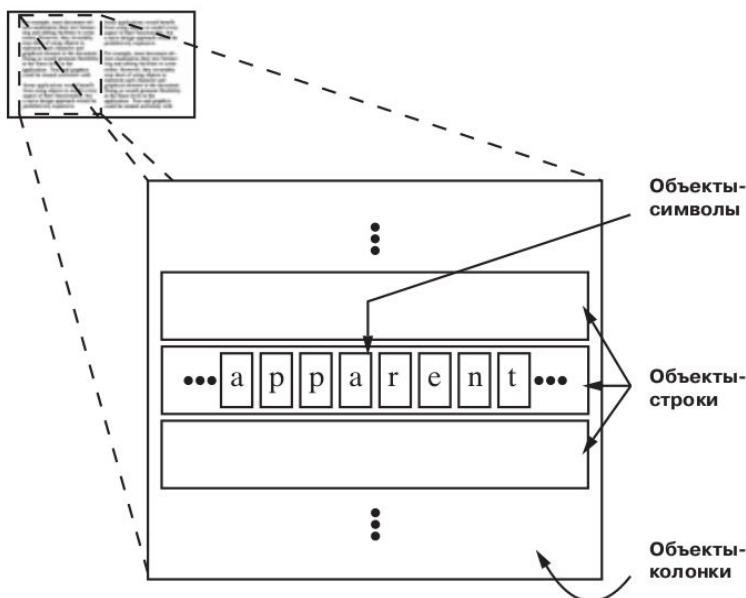
Паттерн приспособленец показывает, как разделять очень мелкие объекты без недопустимо высоких издержек.

Мотивация

В некоторых приложениях использование объектов могло бы быть очень полезным, но прямолинейная реализация оказывается недопустимо расточительной.

Например, в большинстве редакторов документов имеются средства форматирования и редактирования текстов, в той или иной степени модульные. Объектно-ориентированные редакторы обычно применяют объекты для представления таких встроенных элементов, как таблицы и рисунки. Но они не используют объекты для представления каждого символа, несмотря на то что это увеличило бы гибкость на самых низких уровнях приложения. Ведь тогда к рисованию и формированию символов и встроенных элементов можно было бы применить единообразный подход. И для поддержки новых наборов символов не пришлось бы как-либо затрагивать остальные функции редактора. Да и общая структура приложения отражала бы физическую структуру документа. На следующей диаграмме показано, как редактор документов мог бы воспользоваться объектами для представления символов.

У такого дизайна есть один недостаток – стоимость. Даже в документе скромных размеров было бы несколько сотен тысяч объектов-символов, а это привело бы к расходованию огромного объема памяти и неприемлемым затратам во время выполнения. Паттерн приспособленец показывает, как разделять очень мелкие объекты без недопустимо высоких издержек.



Приспособленец – это разделяемый объект, который можно использовать одновременно в нескольких контекстах. В каждом контексте он выглядит как независимый объект, то есть неотличим от экземпляра, который не разделяется. Приспособленцы не могут делать предположений о контексте, в котором работают.

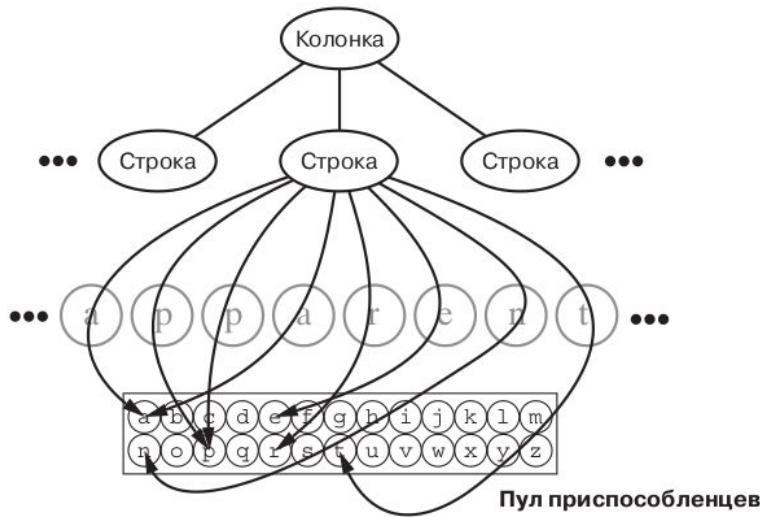
Ключевая идея здесь – различие между *внутренним* и *внешним* состояниями. Внутреннее состояние хранится в самом приспособленце и состоит из информации, не зависящей от его контекста. Именно поэтому он может разделяться. Внешнее состояние зависит от контекста и изменяется вместе с ним, поэтому не подлежит разделению. Объекты-клиенты отвечают за передачу внешнего состояния приспособленцу, когда в этом возникает необходимость.

Приспособленцы моделируют концепции или сущности, число которых слишком велико для представления объектами. Например, редактор документов мог бы создать по одному приспособленцу для каждой буквы алфавита. Каждый приспособленец хранит код символа, но координаты положения символа в документе и стиль его начертания определяются алгоритмами размещения текста и командами форматирования, действующими в том месте, где символ появляется. Код символа – это внутреннее состояние, а все остальное – внешнее.

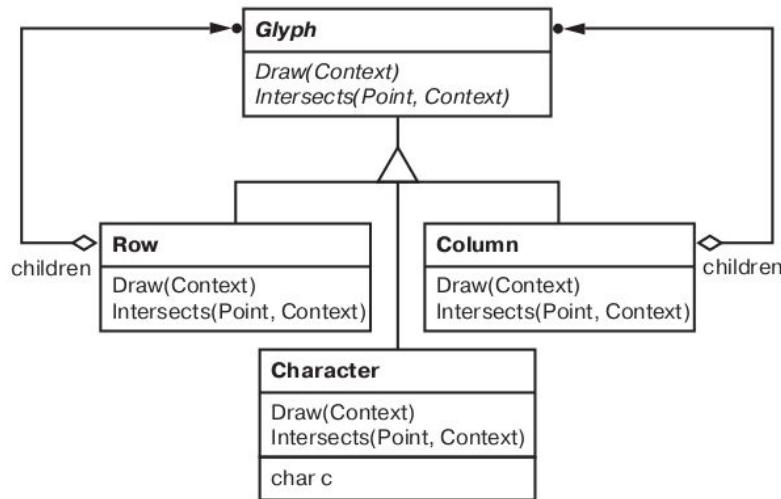
Логически для каждого вхождения данного символа в документ существует объект.



Физически, однако, есть лишь по одному объекту-приспособленцу для каждого символа, который появляется в различных контекстах в структуре документа. Каждое вхождение данного объекта-символа ссылается на один и тот же экземпляр в разделяемом пуле объектов-приспособленцев.



Ниже изображена структура класса для этих объектов. *Glyph* – это абстрактный класс для представления графических объектов (некоторые из них могут быть приспособленцами). Операции, которые могут зависеть от внешнего состояния, передают его в качестве параметра. Например, операциям *Draw* (рисование) и *Intersects* (пересечение) должно быть известно, в каком контексте встречается глиф, иначе они не смогут выполнить то, что от них требуется.



Приспособленец, представляющий букву «а», содержит только соответствующий ей код; ни положение, ни шрифт буквы ему хранить не надо. Клиенты передают приспособленцу всю зависящую от контекста информацию, которая нужна, чтобы он мог изобразить себя. Например, глифу Row известно, где его потомки должны себя показать, чтобы это выглядело как горизонтальная строка. Поэтому вместе с запросом на рисование он может передавать каждому потомку координаты.

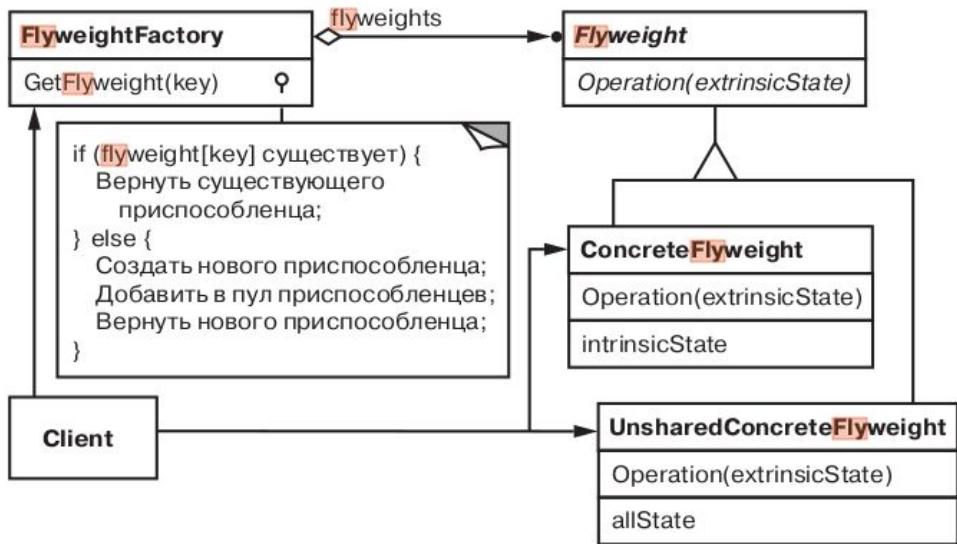
Поскольку число различных объектов-символов гораздо меньше, чем число символов в документе, то и общее количество объектов существенно меньше, чем было бы при простой реализации. Документ, в котором все символы изображаются одним шрифтом и цветом, создаст порядка 100 объектов-символов (это примерно равно числу кодов в таблице ASCII) независимо от своего размера. А поскольку в большинстве документов применяется не более десятка различных комбинаций шрифта и цвета, то на практике эта величина возрастет несущественно. Поэтому абстракция объекта становится применимой и к отдельным символам.

Применимость

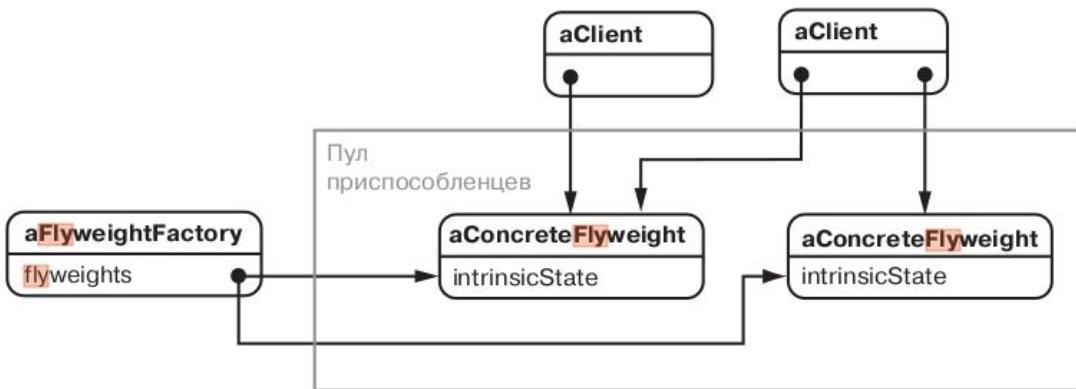
Эффективность паттерна приспособленец во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены *все* нижеперечисленные условия:

- в приложении используется большое число объектов;
 - из-за этого накладные расходы на хранение высоки;
 - большую часть состояния объектов можно вынести вовне;
 - многие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено;
-
- приложение не зависит от идентичности объекта. Поскольку объекты-приспособленцы могут разделяться, то проверка на идентичность возвратит «истину» для концептуально различных объектов.

Структура



На следующей диаграмме показано, как приспособленцы разделяются.



Участники

- **Flyweight** (Glyph) – приспособленец:
 - объявляет интерфейс, с помощью которого приспособленцы могут получать внешнее состояние или как-то воздействовать на него;
- **ConcreteFlyweight** (Character) – конкретный приспособленец:
 - реализует интерфейс класса **Flyweight** и добавляет при необходимости внутреннее состояние. Объект класса **ConcreteFlyweight** должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть не зависящим от контекста;
- **UnsharedConcreteFlyweight** (Row, Column) – неразделяемый конкретный приспособленец:
 - не все подклассы **Flyweight** обязательно должны быть разделяемыми. Интерфейс **Flyweight** допускает разделение, но не навязывает его. Часто у объектов **UnsharedConcreteFlyweight** на некотором уровне структуры

приспособленца есть потомки в виде объектов класса `ConcreteFlyweight`, как, например, у объектов классов `Row` и `Column`;

□ **FlyweightFactory** – фабрика приспособленцев:

- создает объекты-приспособленцы и управляет ими;
- обеспечивает должное разделение приспособленцев. Когда клиент запрашивает приспособленца, объект `FlyweightFactory` предоставляет существующий экземпляр или создает новый, если готового еще нет;

□ **Client** – клиент:

- хранит ссылки на одного или нескольких приспособленцев;
- вычисляет или хранит внешнее состояние приспособленцев.

Отношения

- состояние, необходимое приспособленцу для нормальной работы, можно охарактеризовать как внутреннее или внешнее. Первое хранится в самом объекте `ConcreteFlyweight`. Внешнее состояние хранится или вычисляется клиентами. Клиент передает его приспособленцу при вызове операций;
- клиенты не должны создавать экземпляры класса `ConcreteFlyweight` напрямую, а могут получать их только от объекта `FlyweightFactory`. Это позволит гарантировать корректное разделение.

Результаты

При использовании приспособленцев не исключены затраты на передачу, поиск или вычисление внутреннего состояния, особенно если раньше оно хранилось как внутреннее. Однако такие расходы с лихвой компенсируются экономией памяти за счет разделения объектов-приспособленцев.

Экономия памяти возникает по ряду причин:

- уменьшение общего числа экземпляров;
- сокращение объема памяти, необходимого для хранения внутреннего состояния;
- вычисление, а не хранение внешнего состояния (если это действительно так).

Чем выше степень разделения приспособленцев, тем существеннее экономия. С увеличением объема разделяемого состояния экономия также возрастает. Самого большого эффекта удается добиться, когда суммарный объем внутренней и внешней информации о состоянии велик, а внешнее состояние вычисляется, а не хранится. Тогда разделение уменьшает стоимость хранения внутреннего состояния, а за счет вычислений сокращается память, отводимая под внешнее состояние.

Паттерн приспособленец часто применяется вместе с компоновщиком для представления иерархической структуры в виде графа с разделяемыми листовыми узлами. Из-за разделения указатель на родителя не может храниться в листовом узле-приспособленце, а должен передаваться ему как часть внешнего состояния. Это оказывает заметное влияние на способ взаимодействия объектов иерархии между собой.

Реализация

При реализации приспособленца следует обратить внимание на следующие вопросы:

- *вынесение внешнего состояния.* Применимость паттерна в значительной степени зависит от того, насколько легко идентифицировать внешнее состояние и вынести его за пределы разделяемых объектов. Вынесение внешнего состояния не уменьшает стоимости хранения, если различных внешних состояний так же много, как и объектов до разделения. Лучший вариант – внешнее состояние вычисляется по объектам с другой структурой, требующей значительно меньшей памяти.

Например, в нашем редакторе документов мы можем поместить карту с типографской информацией в отдельную структуру, а не хранить шрифт и начертание вместе с каждым символом. Данная карта будет отслеживать непрерывные серии символов с одинаковыми типографскими атрибутами. Когда объект-символ изображает себя, он получает типографские атрибуты от алгоритма обхода. Поскольку обычно в документах используется немногого разных шрифтов и начертаний, то хранить эту информацию отдельно от объекта-символа гораздо эффективнее, чем непосредственно в нем;

- *управление разделяемыми объектами.* Так как объекты разделяются, клиенты не должны инстанцировать их напрямую. Фабрика FlyweightFactory позволяет клиентам найти подходящего приспособленца. В объектах этого класса часто есть хранилище, организованное в виде ассоциативного массива, с помощью которого можно быстро находить приспособленца, нужного клиенту. Так, в примере редактора документов фабрика приспособленцев может содержать внутри себя таблицу, индексированную кодом символа, и возвращать нужного приспособленца по его коду. А если требуемый приспособленец отсутствует, он тут же создается.

Разделяемость подразумевает также, что имеется некоторая форма подсчета ссылок или сбора мусора для освобождения занимаемой приспособленцем памяти, когда необходимость в нем отпадает. Однако ни то, ни другое необязательно, если число приспособленцев фиксировано и невелико (например, если речь идет о представлении набора символов кода ASCII). В таком случае имеет смысл хранить приспособленцев постоянно.

Родственные паттерны

Паттерн приспособленец часто используется в сочетании с компоновщиком для реализации иерархической структуры в виде ациклического направленного графа с разделяемыми листовыми вершинами.

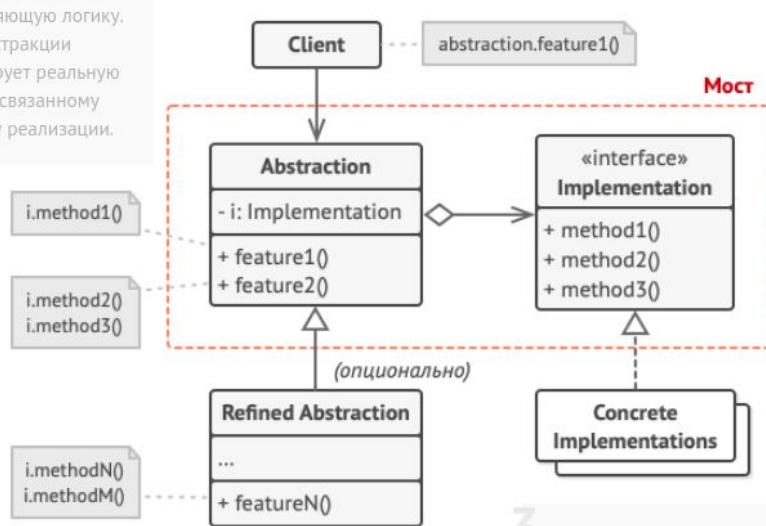
Часто наилучшим способом реализации объектов состояния и стратегии является паттерн приспособленец.

↔ Отношения с другими паттернами

- **Компоновщик** часто совмещают с **Легковесом**, чтобы реализовать общие ветки 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.

33. Паттерн «Мост».

1 Абстракция содержит управляющую логику. Код абстракции делегирует реальную работу связанному объекту реализации.



5 Клиент работает только с объектами абстракции. Не считая начального связывания абстракции с одной из реализаций, клиентский код не имеет прямого доступа к объектам реализации.

2 Реализация задаёт общий интерфейс для всех реализаций. Все методы, которые здесь описаны, будут доступны из класса абстракции и его подклассов.

Интерфейсы абстракции и реализаций могут как совпадать, так и быть совершенно разными. Но обычно в реализации живут базовые операции, на которых строятся сложные операции абстракции.

4 Расширенные абстракции содержат различные вариации управляющей логики. Как и родитель, работает с реализациями только через общий интерфейс реализации.

3 Конкретные реализации содержат платформо-зависимый код.

Назначение

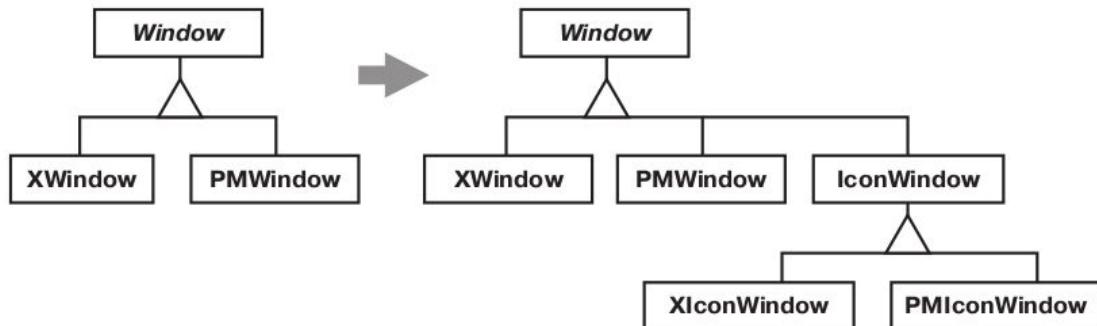
Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

Мотивация

Если для некоторой абстракции возможно несколько реализаций, то обычно применяют наследование. Абстрактный класс определяет интерфейс абстракции, а его конкретные подклассы по-разному реализуют его. Но такой подход не всегда обладает достаточной гибкостью. Наследование жестко привязывает реализацию к абстракции, что затрудняет независимую модификацию, расширение и повторное использование абстракции и ее реализаций.

Рассмотрим реализацию переносимой абстракции окна в библиотеке для разработки пользовательских интерфейсов. Написанные с ее помощью приложения должны работать в разных средах, например под X Window System и Presentation Manager (PM) от компании IBM. С помощью наследования мы могли бы определить абстрактный класс `Window` и его подклассы `XWindow` и `PMWindow`, реализующие интерфейс окна для разных платформ. Но у такого решения есть два недостатка:

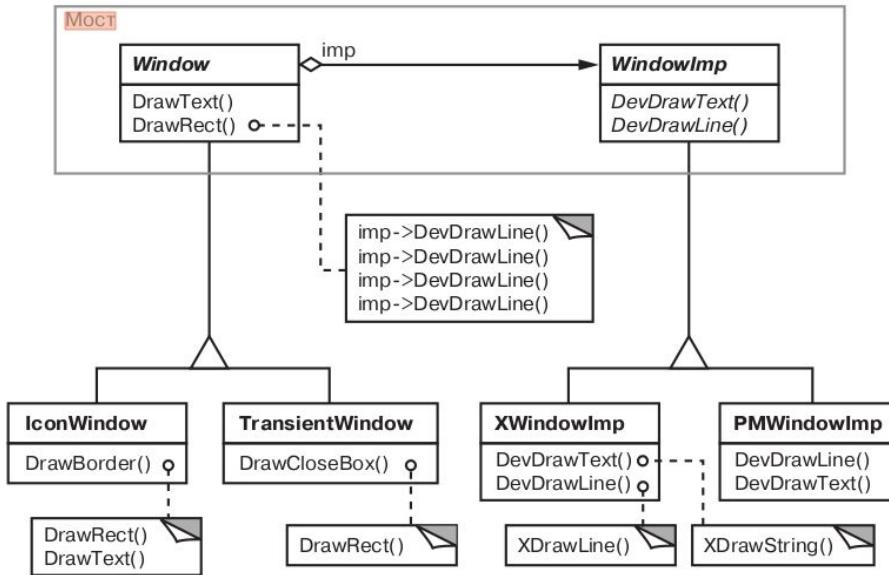
- неудобно распространять абстракцию `Window` на другие виды окон или новые платформы. Представьте себе подкласс `IconWindow`, который специализирует абстракцию окна для пиктограмм. Чтобы поддержать пиктограммы на обеих платформах, нам придется реализовать *два* новых подкласса `XIconWindow` и `PMIconWindow`. Более того, по два подкласса необходимо определять для *каждого* вида окон. А для поддержки третьей платформы придется определять для всех видов окон новый подкласс `Window`;



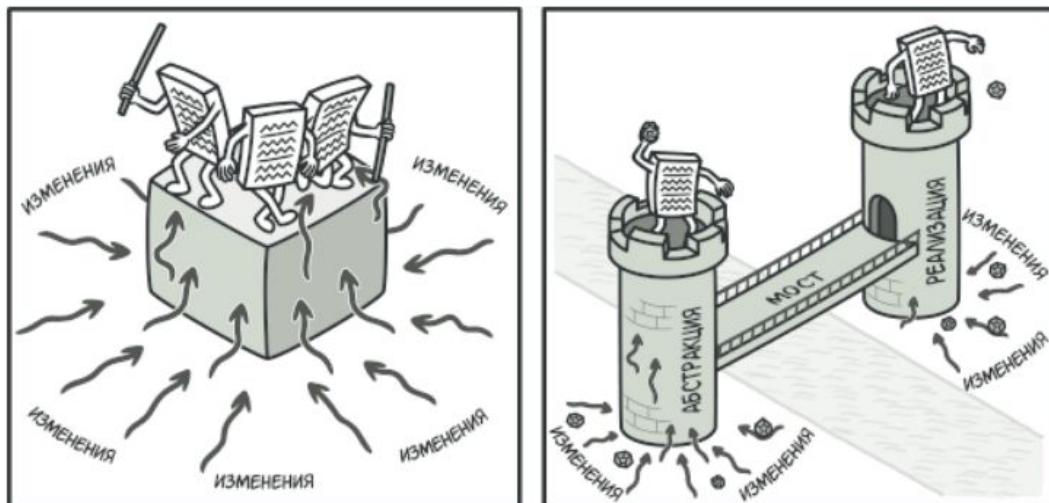
- клиентский код становится платформенно-зависимым. При создании окна клиент инстанцирует конкретный класс, имеющий вполне определенную реализацию. Например, создавая объект `XWindow`, мы привязываем абстракцию окна к ее реализации для системы X Window и, следовательно, делаем код клиента ориентированным именно на эту оконную систему. Таким образом усложняется перенос клиента на другие платформы.

Клиенты должны иметь возможность создавать окно, не привязываясь к конкретной реализации. Только сама реализация окна должна зависеть от платформы, на которой работает приложение. Поэтому в клиентском коде не может быть никаких упоминаний о платформах.

С помощью паттерна **мост** эти проблемы решаются. Абстракция окна и ее реализация помещаются в раздельные иерархии классов. Таким образом, существует одна иерархия для интерфейсов окон (`Window`, `IconWindow`, `TransientWindow`) и другая (с корнем `WindowImp`) – для платформенно-зависимых реализаций. Так, подкласс `XWindowImp` предоставляет реализацию в системе X Window System.



Все операции подклассов `Window` реализованы в терминах абстрактных операций из интерфейса `WindowImp`. Это отделяет абстракцию окна от различных ее платформенно-зависимых реализаций. Отношение между классами `Window` и `WindowImp` мы будем называть **мостом**, поскольку между абстракцией и реализацией строится мост, и они могут изменяться независимо.



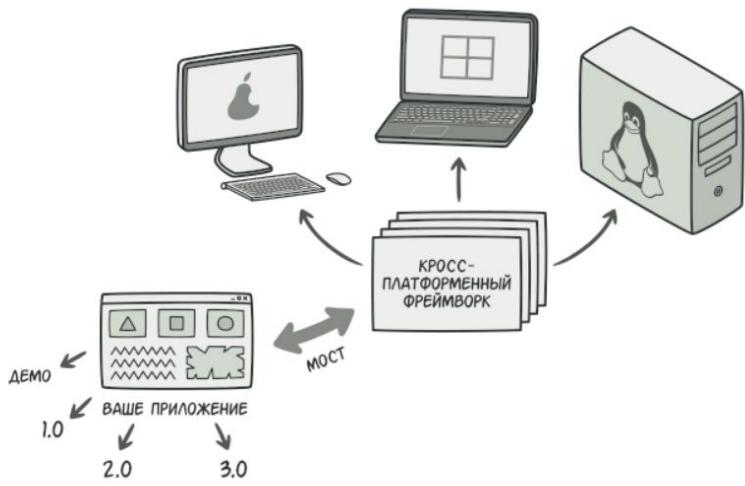
Когда изменения «осаждают» проект, вам легче отбиваться, если разделить монолитный код на части.

Вы можете попытаться структурировать этот хаос, создав для каждой вариации интерфейса-платформы свои подклассы. Но такой подход приведёт к росту классов комбинаций, и с каждой новой платформой их будет всё больше.

Мы можем решить эту проблему, применив Мост. Паттерн предлагает распутать этот код, разделив его на две части:

- Абстракцию: слой графического интерфейса приложения.

- Реализацию: слой взаимодействия с операционной системой.



Абстракция будет делегировать работу одному из объектов реализаций. Причём, реализации можно будет взаимозаменять, но только при условии, что все они будут следовать общему интерфейсу.

Таким образом, вы сможете изменять графический интерфейс приложения, не трогая низкоуровневый код работы с операционной системой. И наоборот, вы сможете добавлять поддержку новых операционных систем, создавая подклассы реализации, без необходимости менять классы графического интерфейса.

↗ ↘ Применимость

- ↗ Когдa вы хотите разделить монолитный класс, который содержит несколько различных реализаций какой-то функциональности (например, если класс может работать с разными системами баз данных).
- ↗ Чем больше класс, тем тяжелее разобраться в его коде, и тем больше это затягивает разработку. Кроме того, изменения, вносимые в одну из реализаций, приводят к редактированию всего класса, что может привести к внесению случайных ошибок в код.

Мост позволяет разделить монолитный класс на несколько отдельных иерархий. После этого вы можете менять их код независимо друг от друга. Это упрощает работу над кодом и уменьшает вероятность внесения ошибок.

- ↗ Когдa класс нужно расширять в двух независимых плоскостях.

- ↗ Мост предлагает выделить одну из таких плоскостей в отдельную иерархию классов, храня ссылку на один из её объектов в первоначальном классе.

- ↗ Когдa вы хотите, чтобы реализацию можно было бы изменять во время выполнения программы.

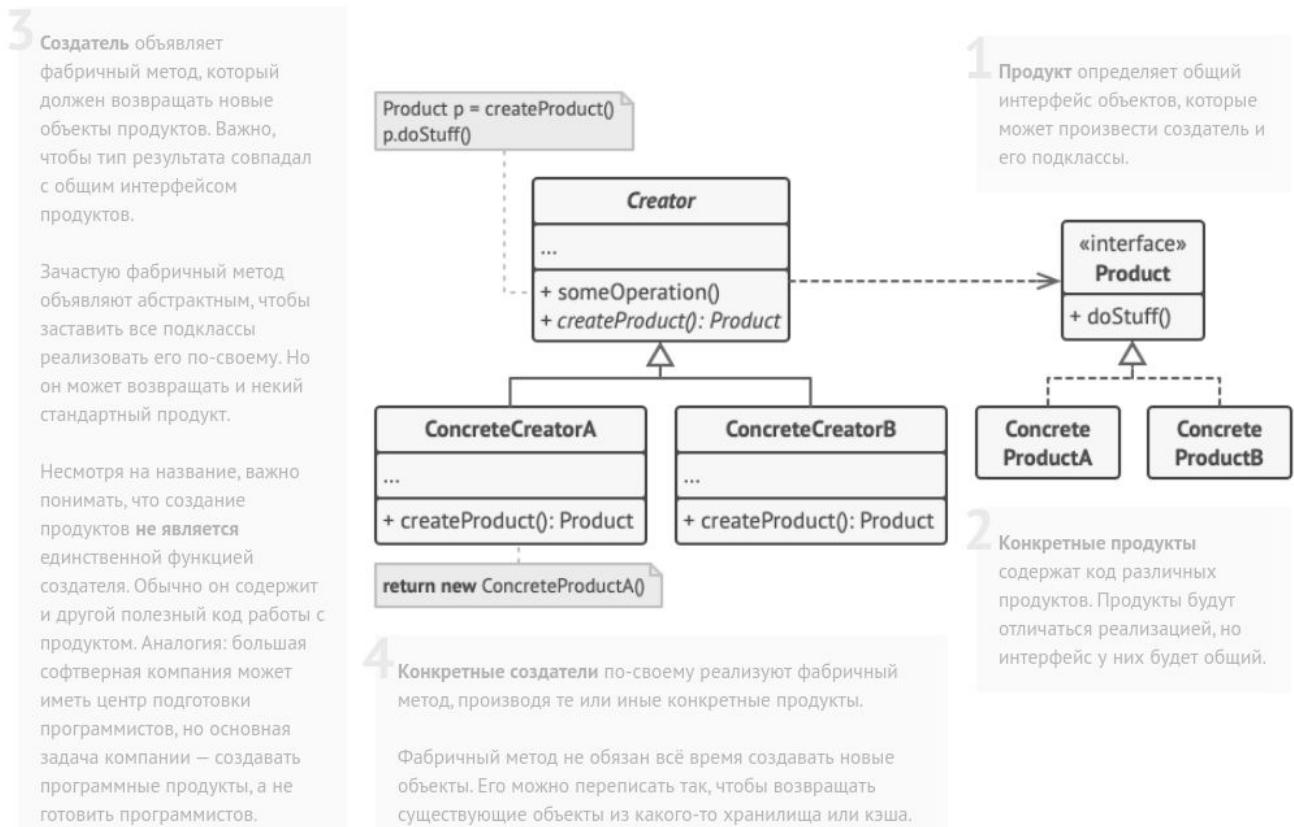
- ↗ Мост позволяет заменять реализацию даже во время выполнения программы, так как конкретная реализация не «вшита» в класс абстракции.

Кстати, из-за этого пункта Мост часто путают со [Стратегией](#). Обратите внимание, что у Моста этот пункт стоит на последнем месте по значимости, поскольку его главная задача – структурная.

⊕⊖ Преимущества и недостатки

- | | |
|---|--|
| <ul style="list-style-type: none">✓ Позволяет строить платформо-независимые программы.✓ Скрывает лишние или опасные детали реализации от клиентского кода.✓ Реализует <i>принцип открытости/закрытости</i>. | <ul style="list-style-type: none">✗ Усложняет код программы из-за введения дополнительных классов. |
|---|--|

34. Паттерн «Фабричный метод».



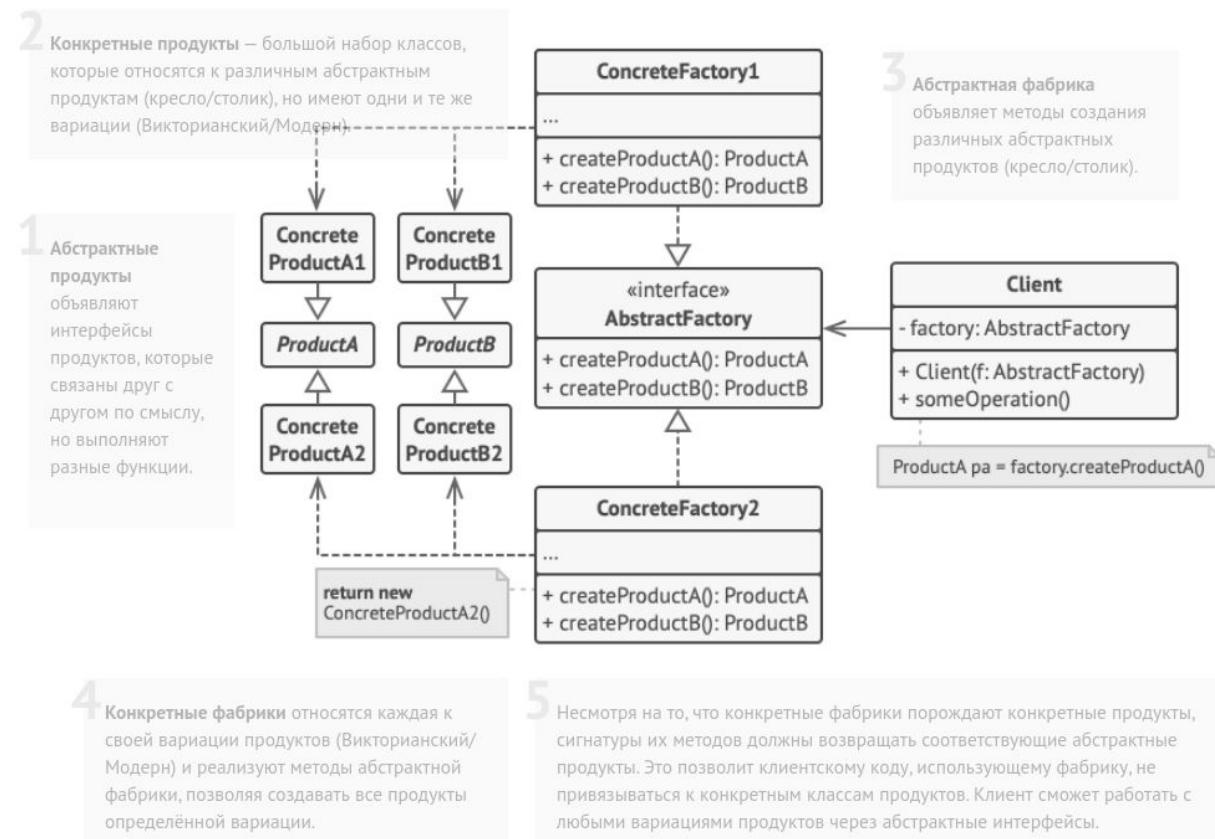
Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Проблема:



Добавить новый класс не так-то просто, если весь код уже завязан на конкретные классы.

35. Паттерн «Абстрактная фабрика».

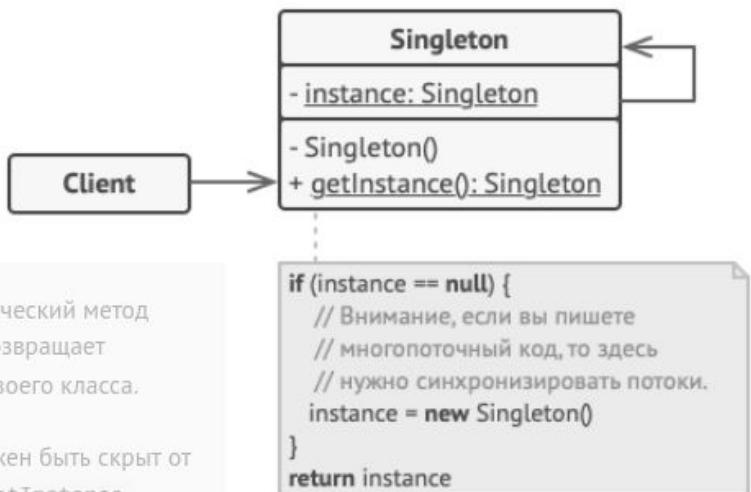


Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Проблема:



36. Паттерн «Одиночка».



1

Одиночка определяет статический метод `getInstance`, который возвращает единственный экземпляр своего класса.

Конструктор одиночки должен быть скрыт от клиентов. Вызов метода `getInstance` должен стать единственным способом получить объект этого класса.

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

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

Минусы:

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

Double-checked locking

Более-менее хорошая многопоточная реализация

```
public class Singleton {  
    private static volatile Singleton instance;  
  
    public static Singleton getInstance() {  
        Singleton localInstance = instance;  
        if (localInstance == null) {  
            synchronized (Singleton.class) {  
                localInstance = instance;  
                if (localInstance == null) {  
                    instance = localInstance = new Singleton();  
                }  
            }  
        }  
        return localInstance;  
    }  
}
```

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

Ленивая инициализация

- ▶ Некоторое упрощение **одиночки**
- ▶ 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
 - ▶ Ленивые структуры данных (списки в Haskell, seq в F#)
 - ▶ Ленивые вычисления (Haskell, Lazy<T> в .NET)
 - ▶ ...
- ▶ Имеет те же проблемы с многопоточностью, что и **одиночка**

Пул объектов

Паттерн “Пул объектов”, мотивация

Потоки в .NET

- ▶ Класс Thread, конструктор создаёт поток и запускает в нём переданную операцию
- ▶ Поток уничтожается, когда операция завершилась
- ▶ Создание и остановка потоков — долгие операции
- ▶ Каждый поток требует системных ресурсов
- ▶ Нет смысла иметь больше потоков, чем ядер процессора

Паттерн “Пул объектов”, мотивация

Потоки в .NET

- ▶ Класс Thread, конструктор создаёт поток и запускает в нём переданную операцию
- ▶ Поток уничтожается, когда операция завершилась
- ▶ Создание и остановка потоков — долгие операции
- ▶ Каждый поток требует системных ресурсов
- ▶ Нет смысла иметь больше потоков, чем ядер процессора

Паттерн “Пул объектов”

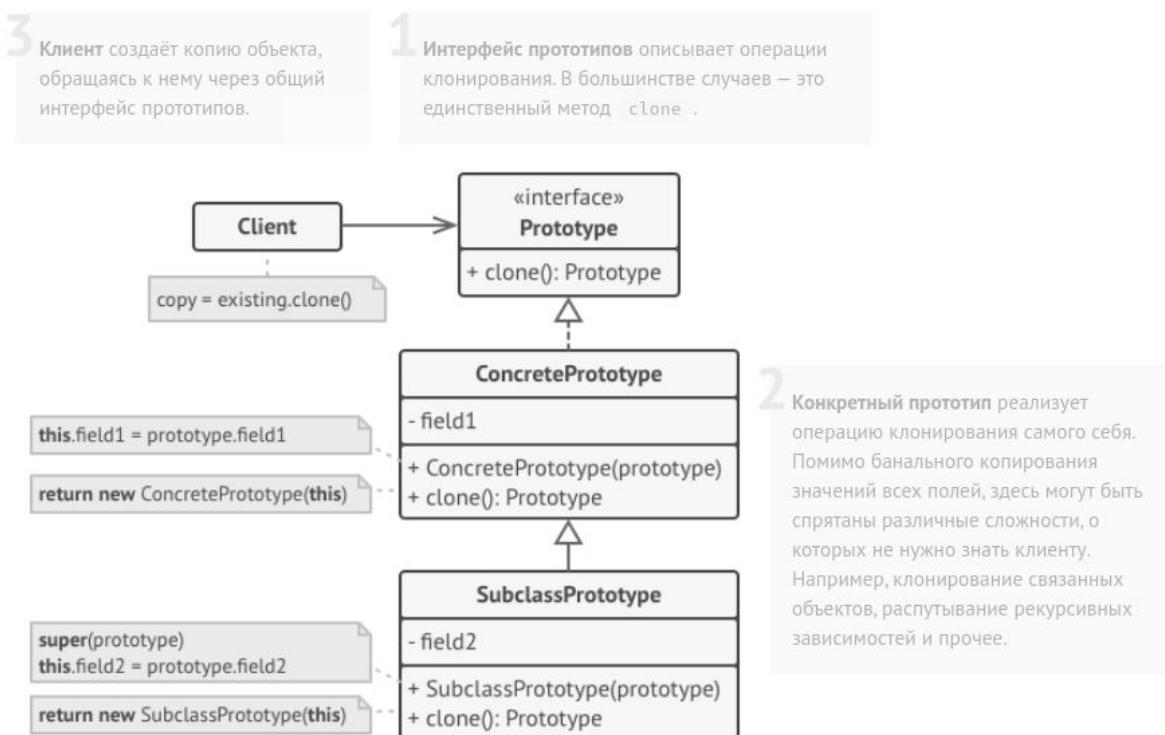
Решение: пул потоков в .NET

- ▶ Класс ThreadPool, синглтон
- ▶ Создаёт заранее N потоков, которые никогда не заканчиваются и ждут задач
- ▶ QueueUserWorkItem принимает задачу на исполнение
 - ▶ Например,
`ThreadPool.QueueUserWorkItem(() => Console.WriteLine("Goodbye, world!"))`
- ▶ Если есть свободный поток, он начинает исполнять задачу
- ▶ Если свободных потоков нет, а задач много, создаётся новый поток
- ▶ Лишние потоки удаляются, если задач нет и число потоков больше N

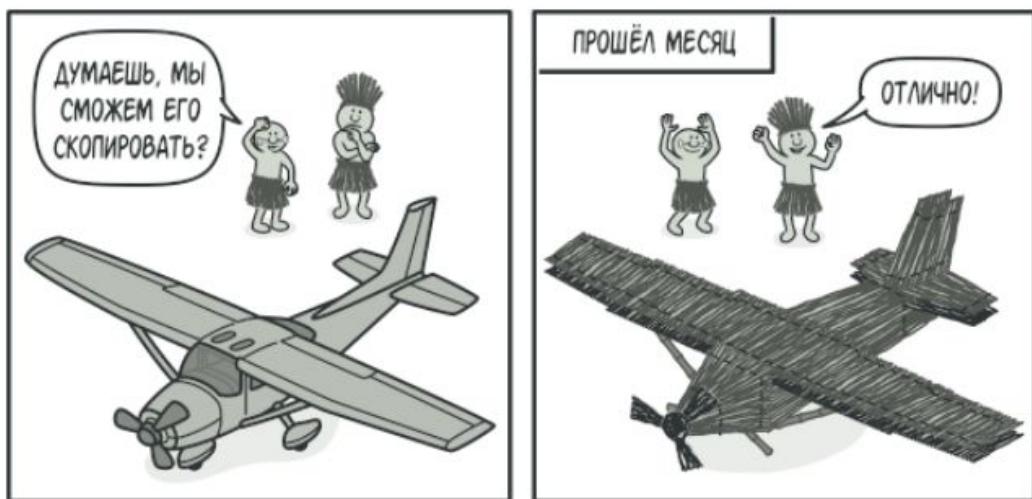
Паттерн “Пул объектов”

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

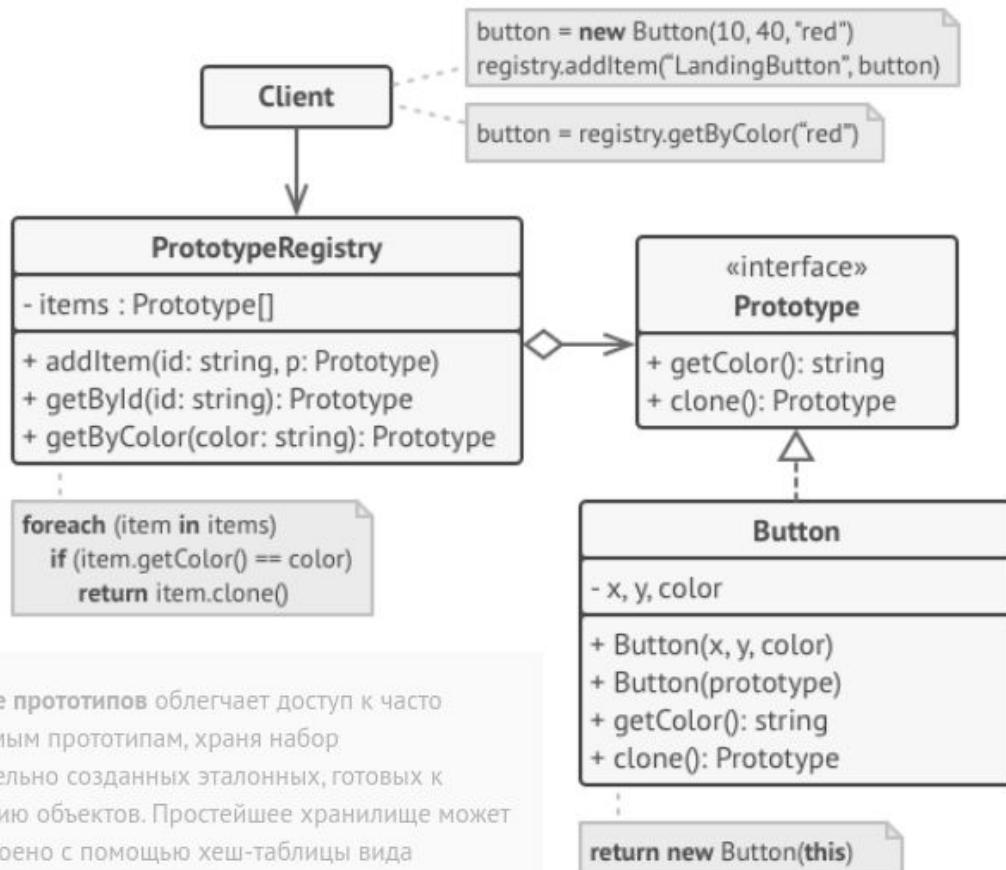
38. Паттерн «Прототип».



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



Копирование «извне» не всегда возможно в реальности.



1 Хранилище прототипов облегчает доступ к часто используемым прототипам, храня набор предварительно созданных эталонных, готовых к копированию объектов. Простейшее хранилище может быть построено с помощью хеш-таблицы вида имя-прототипа → прототип . Но для удобства поиска прототипы можно маркировать и другими критериями, а не только условным именем.

Как глубокого копирования:

- ▶ Сериализовать/десериализовать объект (но помнить про идентичность)

Но! Нужно идентифицировать объект, если это вообще возможно (мб идентификатор в конструкторе, да ещё и рандомно выдаётся)

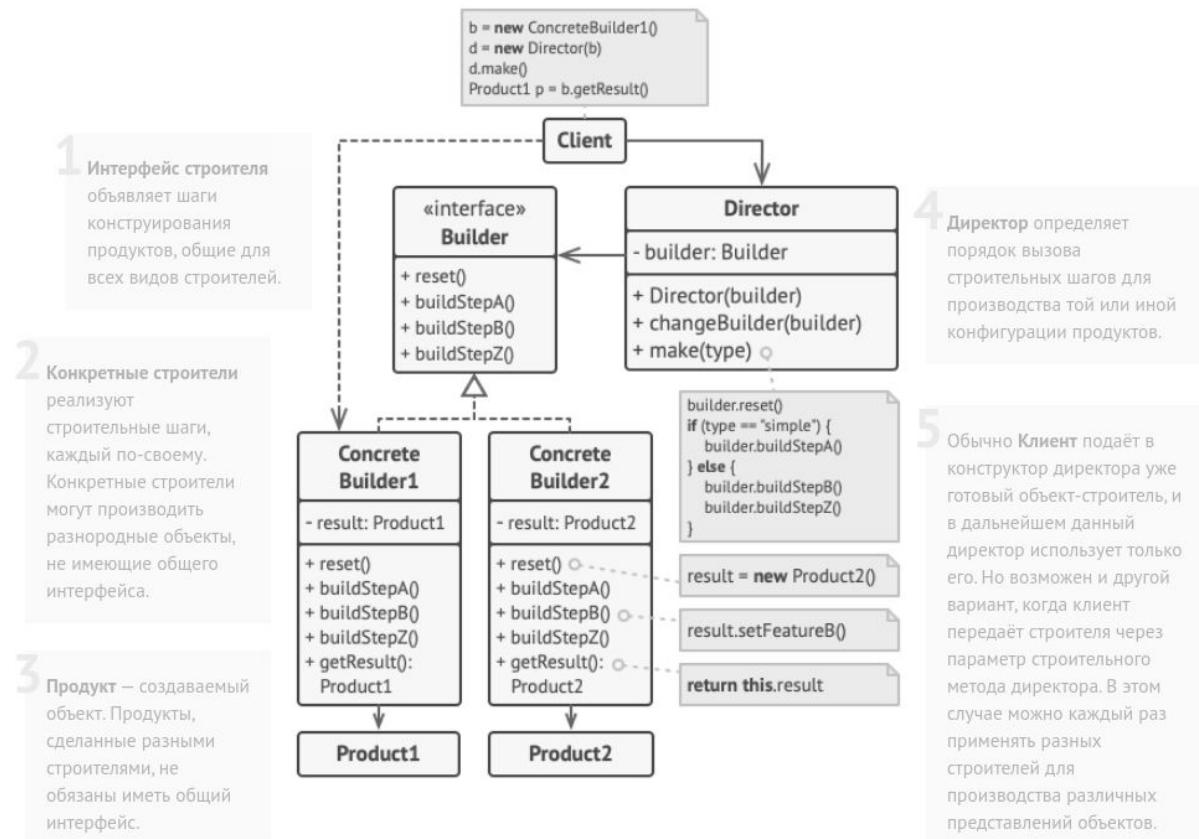
Но правильнее

Пробежаться по графу объектов

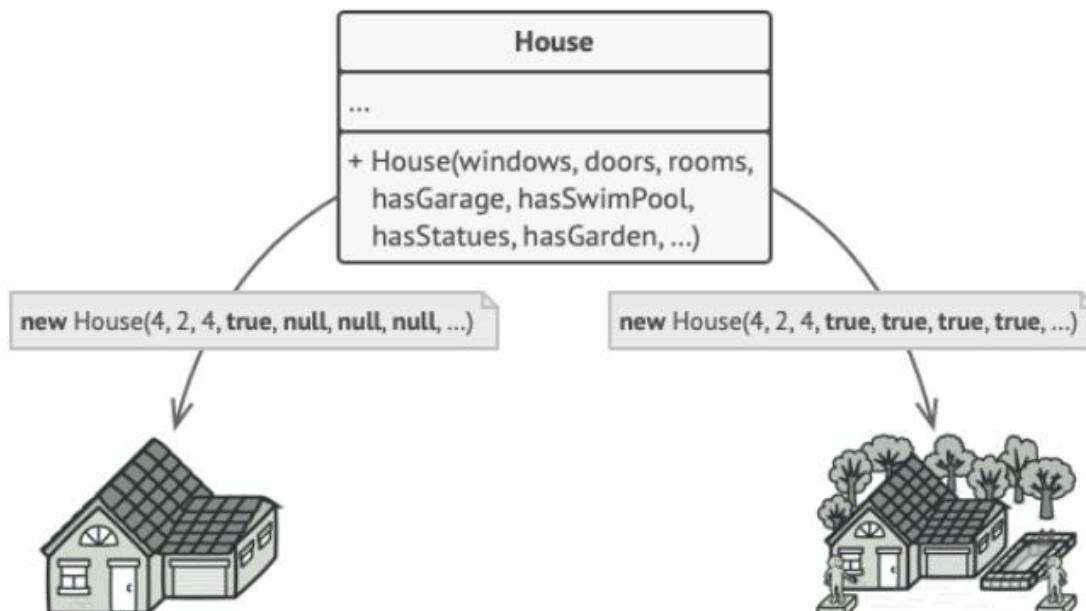
Сделать топологическую сортировку

Циклы нашлись => запомнить, дать объекту по идентификатору, следить за идентификатору на который ссылаемся, подставлять копию объекта если уже ссылались

39. Паттерн «Строитель».



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.



40. Паттерн «Посредник».

1

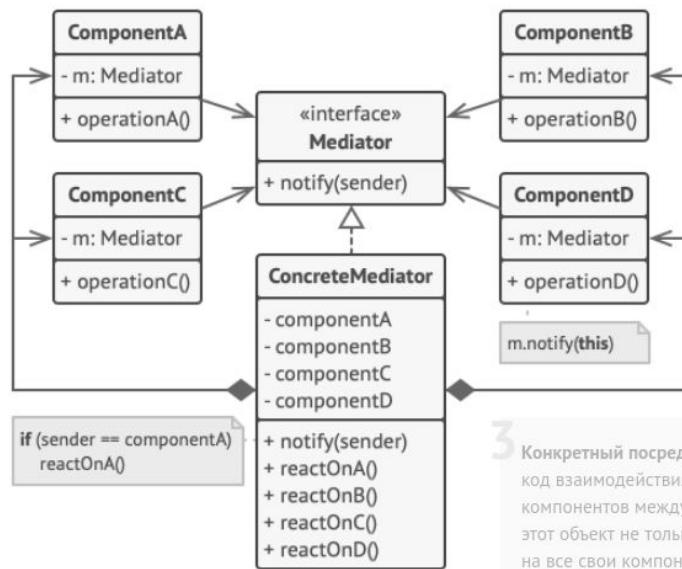
Компоненты — это разнородные объекты, содержащие бизнес-логику программы. Каждый компонент хранит ссылку на объект посредника, но работает с ним только через абстрактный интерфейс посредников. Благодаря этому, компоненты можно повторно использовать в другой программе, связав их с посредником другого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

2

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

4

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

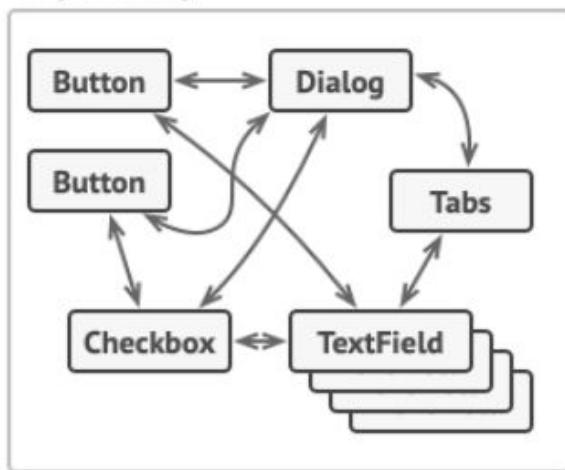


3

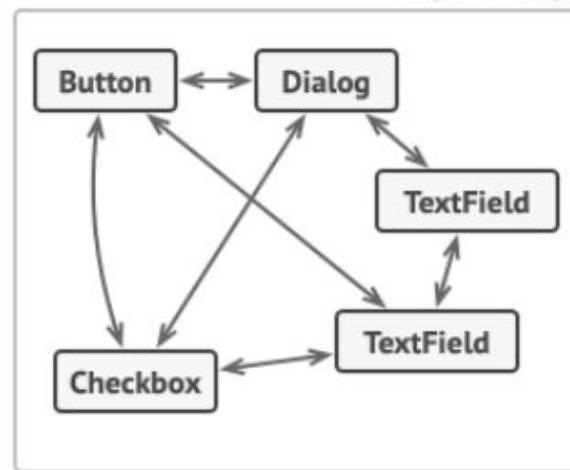
Конкретный посредник содержит код взаимодействия нескольких компонентов между собой. Зачастую этот объект не только хранит ссылки на все свои компоненты, но и сам их создаёт, управляя дальнейшим жизненным циклом.

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

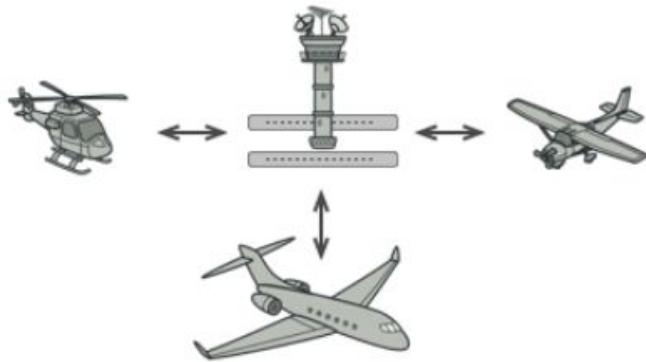
Profile Dialog



Login Dialog



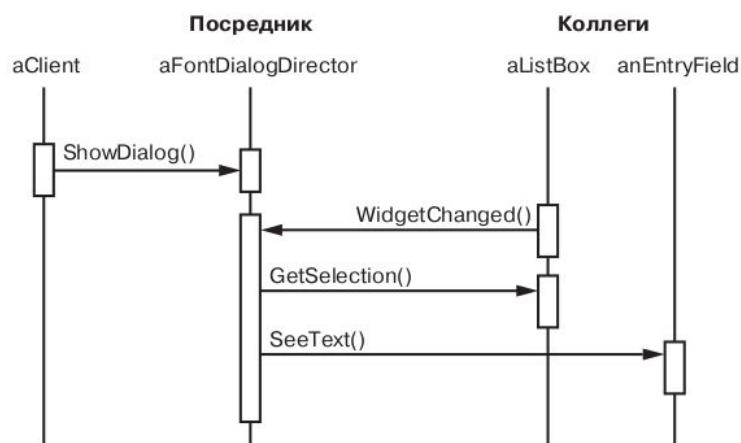
Беспорядочные связи между элементами пользовательского интерфейса.



Пилоты самолётов общаются не напрямую, а через диспетчера.

Сайт + книга

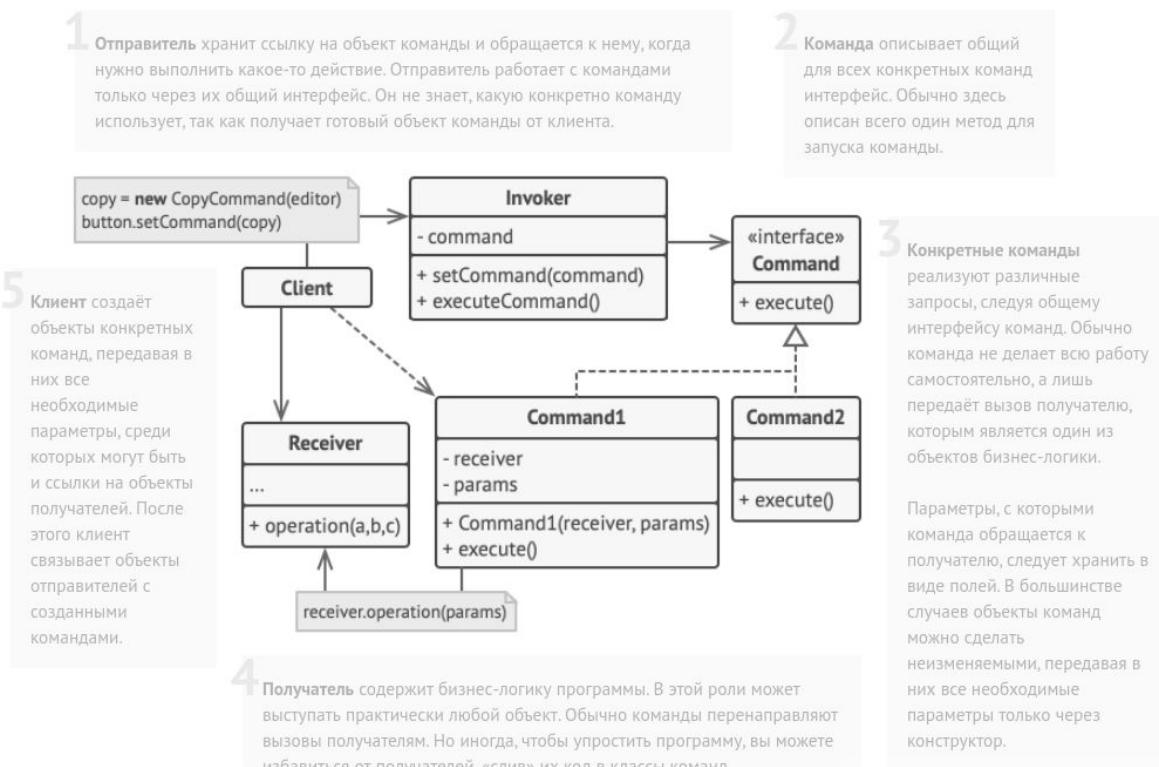
На следующей диаграмме взаимодействий показано, как объекты кооперируются друг с другом, реагируя на изменение выбранного элемента списка.



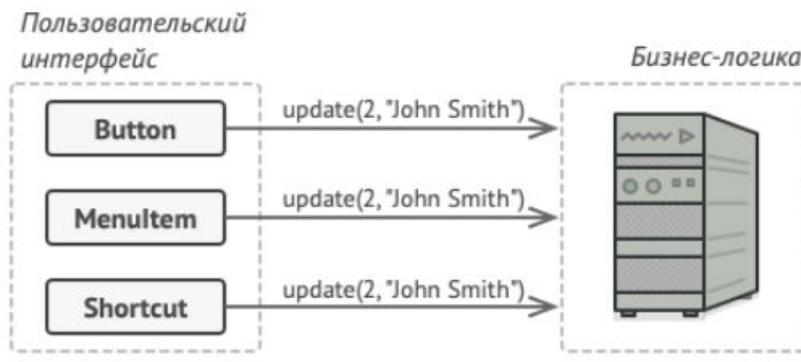
Последовательность событий, в результате которых информация о выбранном элементе списка передается в поле ввода, следующая:

1. Список информирует распорядителя о произошедших в нем изменениях.
2. Распорядитель получает от списка выбранный элемент.
3. Распорядитель передает выбранный элемент полю ввода.
4. Теперь, когда поле ввода содержит какую-то информацию, распорядитель активизирует кнопки, позволяющие выполнить определенное действие (например, изменить шрифт на полужирный или курсив).

41. Паттерн «Команда».



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

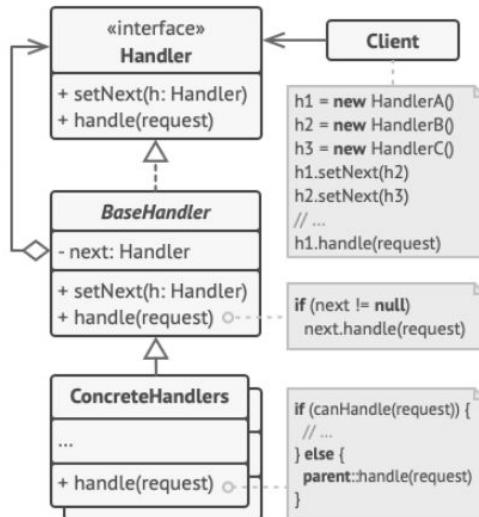


42. Паттерн «Цепочка ответственности».

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

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

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

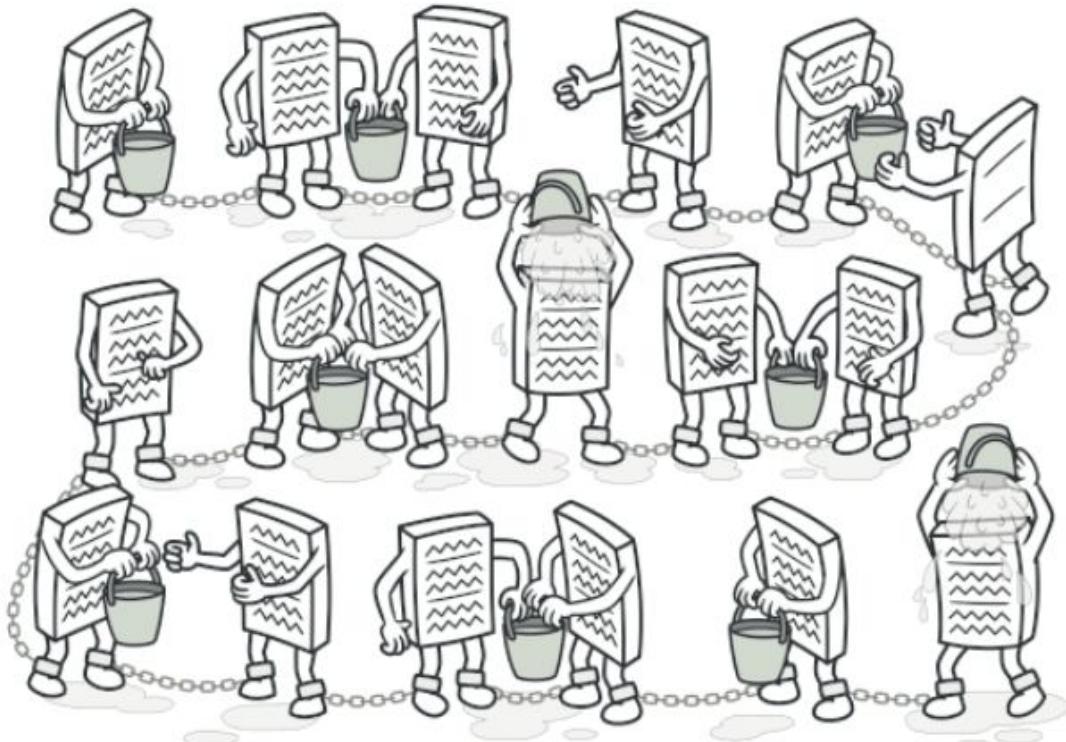


4 Клиент может либо сформировать цепочку обработчиков единожды, либо перестраивать её динамически, в зависимости от логики программы. Клиент может отправлять запросы любому из объектов цепочки, не обязательно первому из них.

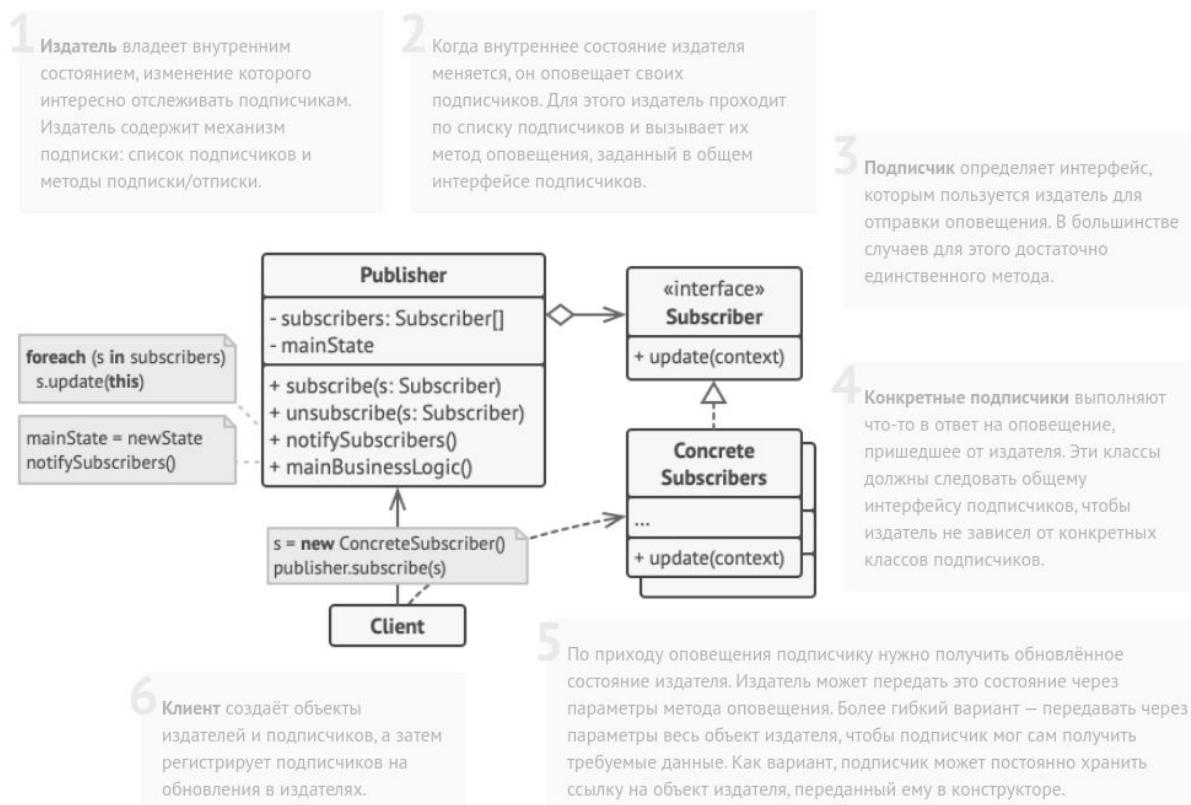
3 Конкретные обработчики содержат код обработки запросов. При получении запроса каждый обработчик решает, может ли он обработать запрос, а также стоит ли передать его следующему объекту.

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

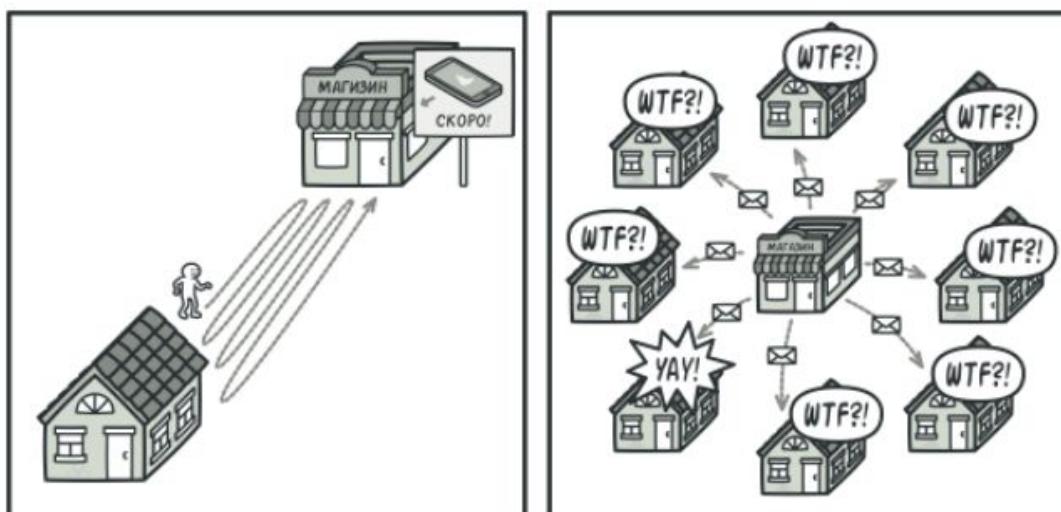
Цепочка обязанностей — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи



43. Паттерн «Наблюдатель».



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



Постоянное посещение магазина или спам?

44. Паттерн «Состояние».

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.

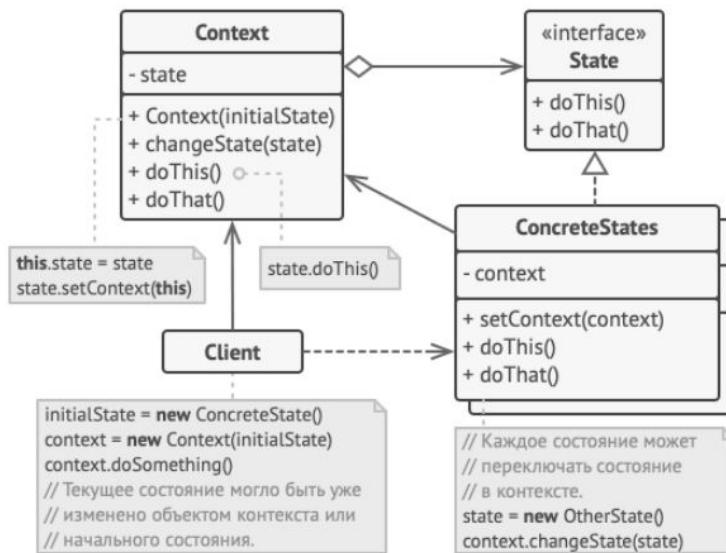
3

Конкретные состояния реализуют поведения, связанные с определённым состоянием контекста. Иногда приходится создавать целые иерархии классов состояний, чтобы обобщить дублирующий код.

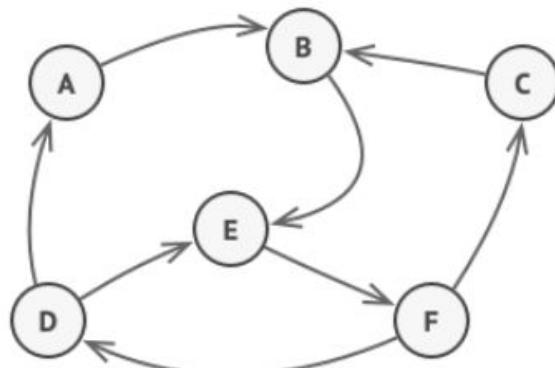
Состояние может иметь обратную ссылку на объект контекста. Через неё не только удобно получать из контекста нужную информацию, но и осуществлять смену его состояния.

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.



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

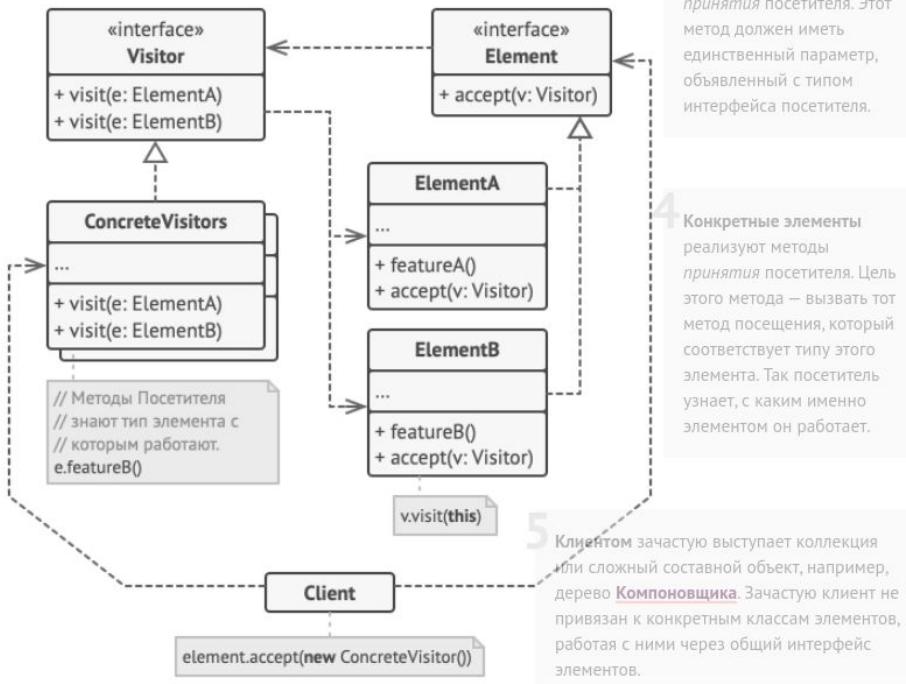


Конечный автомат.

45. Паттерн «Посетитель».

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.

Двойная диспетчеризация

```

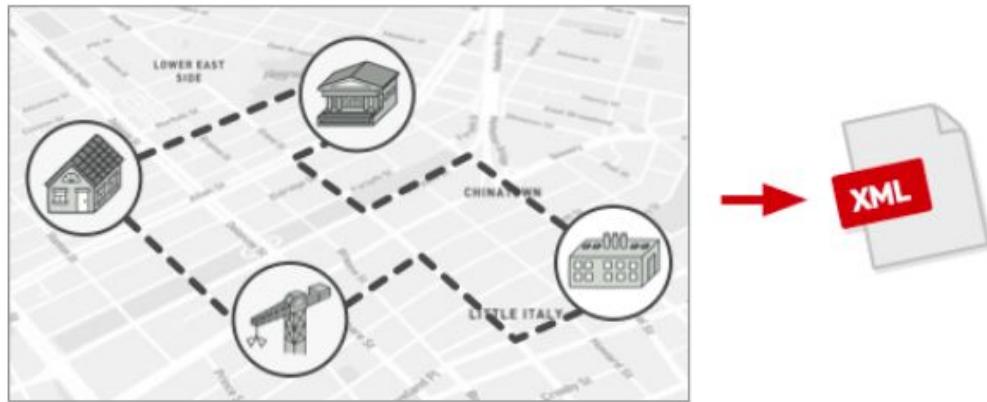
class Dot implements Shape {
    ...
    method accept(v: Visitor) {
        v.visitDot(this)
    }
}

class Circle implements Shape {
    ...
    method accept(v: Visitor) {
        v.visitCircle(this)
    }
}

class Rectangle implements Shape {
    ...
    method accept(v: Visitor) {
        v.visitRectangle(this)
    }
}

class CompoundShape implements Shape {
    ...
    method accept(v: Visitor) {
        v.visitCompoundShape(this)
    }
}
  
```

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



Экспорт геоузлов в XML.

```
// Конкретный посетитель реализует одну операцию для всей
// иерархии элементов. Новая операция = новый посетитель.
// Посетитель выгодно применять, когда новые элементы
// добавляются очень редко, а новые операции — часто.
```

Новая операция -- это новая работа над элементами структуры. =>
Нужен новый посетитель, который реализует новую операцию для каждого типа элемента структуры.

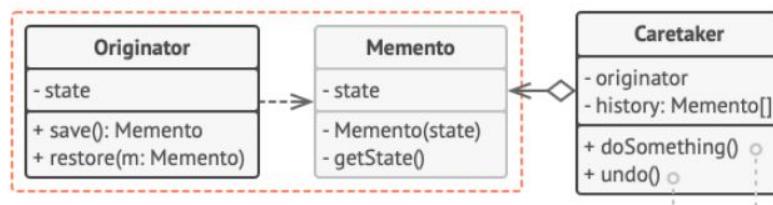
46. Паттерн «Хранитель».

Классическая реализация на вложенных классах

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.



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.

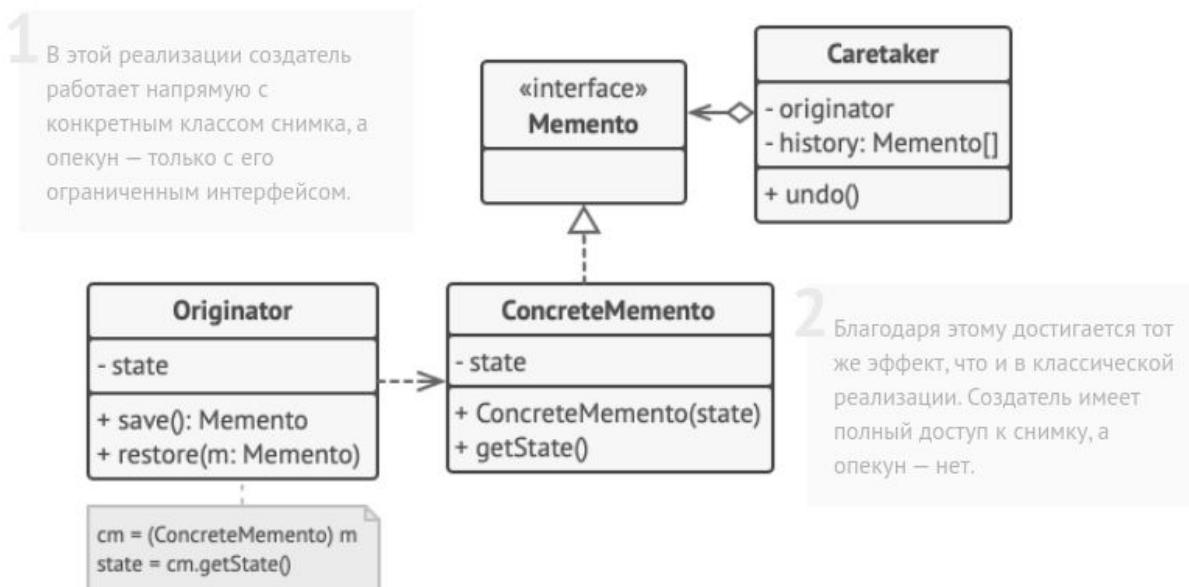
Опекун может хранить историю прошлых состояний создателя в виде стека из снимков. Когда понадобится отменить выполненную операцию, он возьмёт «верхний» снимок из стека и передаст его создателю для восстановления.

```
m = history.pop()
originator.restore(m)

m = originator.save()
history.push(m)
// originator.change()
```

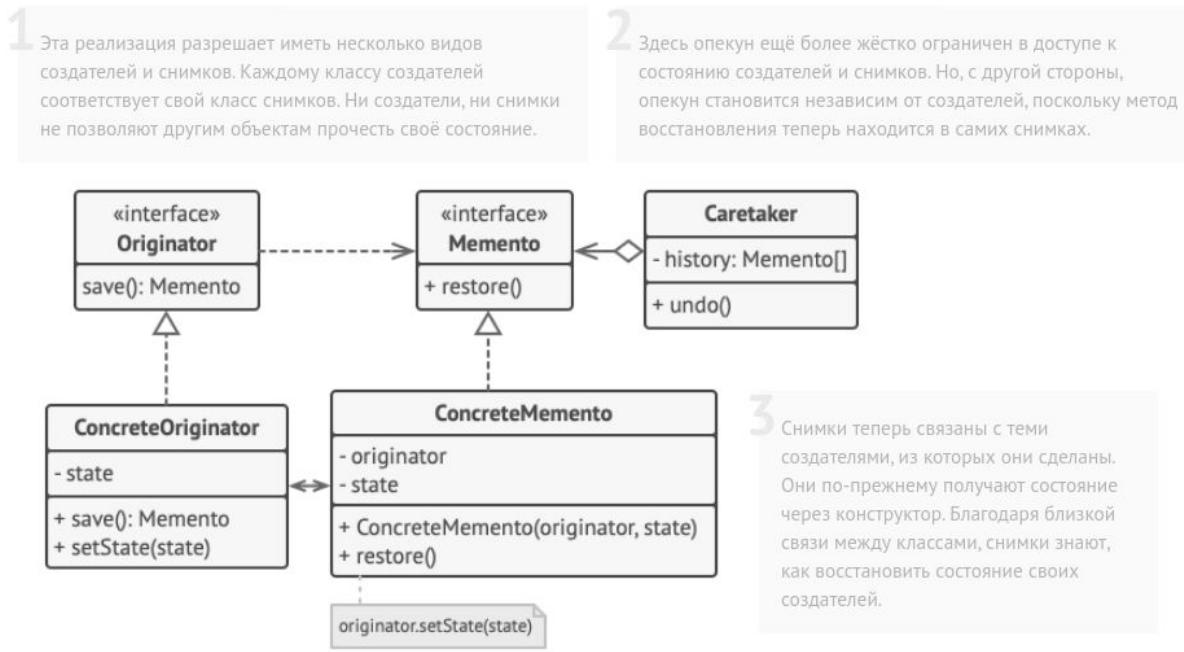
Реализация с пустым промежуточным интерфейсом

Подходит для языков, не имеющих механизма вложенных классов (например, PHP).



Снимки с повышенной защитой

Когда нужно полностью исключить возможность доступа к состоянию создателей и снимков.



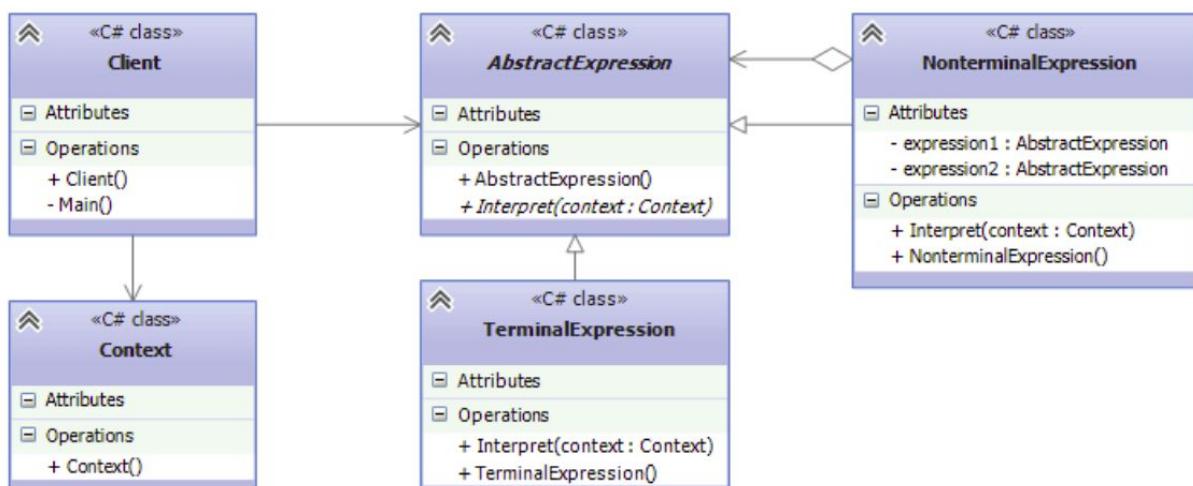
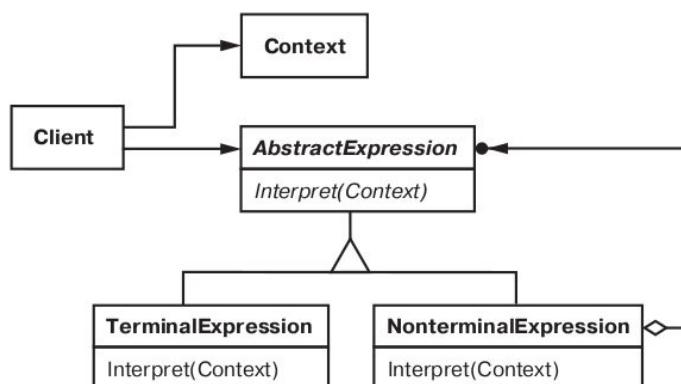
Снимок — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.



Как команде создать снимок состояния редактора, если все его поля приватные?

Хранитель -- сохраняет состояние объекта. Прототип -- копирует весь объект.

47. Паттерн «Интерпретатор».



<https://metanit.com/sharp/patterns/3.8.php>

+

книга 237 стр

48. Паттерн «Спецификация».

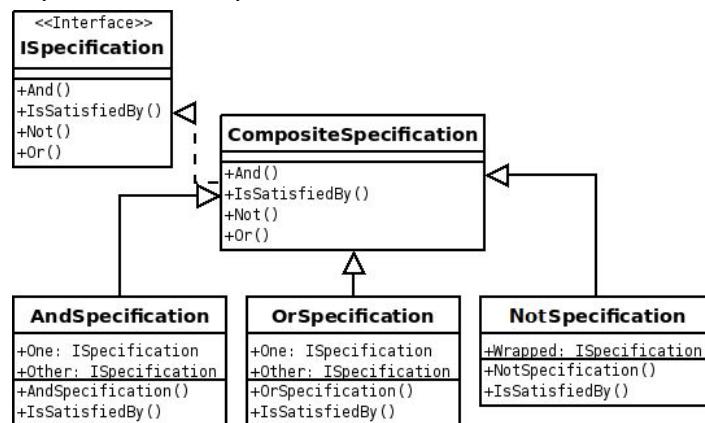
Спецификация инкапсулирует ограничение в отдельном объекте

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

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

Композитная спецификация

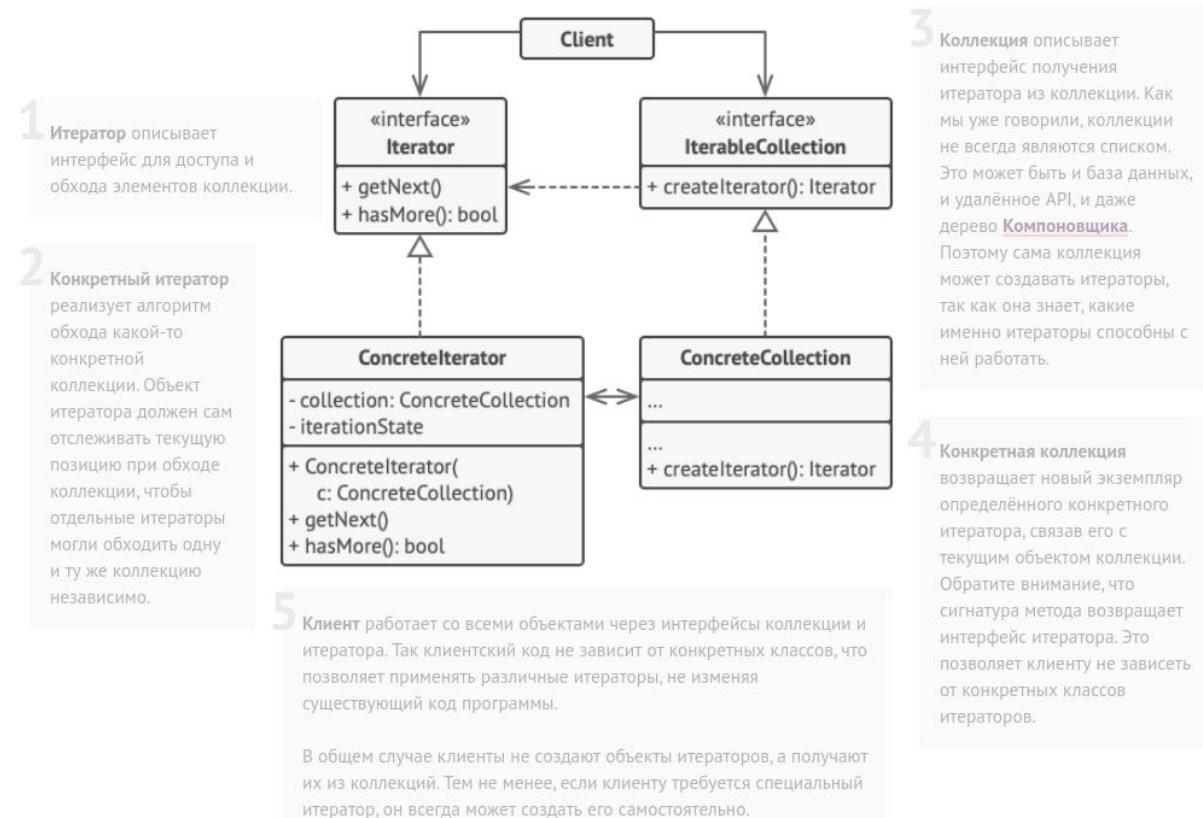
Содержит цепочки простых спецификаций.



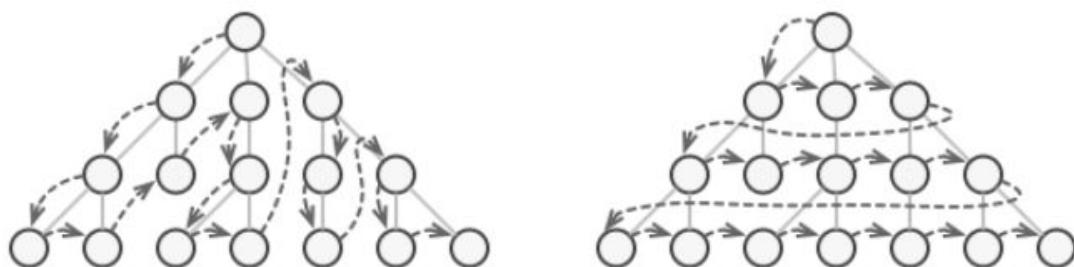
22 стр:

file:///media/patoshca/%D0%A5%D1%80%D0%B0%D0%BD%D0%B8%D0%BB%D0%B8%D1%89%D0%B5/University/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0%20%D0%9F%D0%9E/11-ddd-text.pdf

49. Паттерн «Итератор».



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



Одну и ту же коллекцию можно обходить разными способами.

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

Антипаттерны — что это и зачем?

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

Семь причин провала проектов

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

Круговая зависимость

Circular dependency

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

Последовательная связность

Sequential coupling

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

Вызов предка

Call super

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

Проблема йо-йо

Yo-yo problem

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

<file:///media/patoshca/%D0%A5%D1%80%D0%B0%D0%BD%D0%B8%D0%BB%D0%B8%D1%89%D0%B5/University/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0%20%D0%9F%D0%9E/09-antipatterns-text.pdf>

Наследование плохо, т.к.:

Вообще, 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 (в частности поэтому во второй лекции говорилось, что наследование — это плохо). 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», «Сокрытие ошибки», «Магические числа», «Магические строки».

Активное ожидание

Busy waiting

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

Сокрытие ошибки

Error hiding

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

Магические числа и строки

Magic numbers, Magic strings

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

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

Божественный объект

God Object (The Blob)

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

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

God Object, что делать

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

Swiss Army Knife

- ▶ Swiss Army Knife — класс с чрезмерно сложным интерфейсом, который пытается уметь делать всё, что в принципе может понадобиться
 - ▶ Часто появляется в библиотеках как результат попыток вендора сделать свою технологию возможно более применимой
- ▶ Инкапсуляция сложности приносится в жертву гибкости, что лишает абстракцию смысла
- ▶ Отличается от God Object-а тем, что не пытается монополизировать управление
 - ▶ Швейцарских ножей может быть много
 - ▶ God Object часто имеет очень простой public-интерфейс

Поток лавы

Lava Flow

“Oh that! Well Ray and Emil (they’re no longer with the company) wrote that routine back when Jim (who left last month) was trying a workaround for Irene’s input processing code (she’s in another department now, too). I don’t think it’s used anywhere now, but I’m not really sure. Irene didn’t really document it very clearly, so we figured we would just leave well enough alone for now. After all, the bloomin’ thing works doesn’t it?!”

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

Lava Flow, что делать

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

53. Антипаттерны «Функциональная декомпозиция», «Полтергейст», «Золотой МОЛОТОК».

Функциональная декомпозиция

Functional Decomposition

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

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

Функциональная декомпозиция

конспект + <https://sourcemaking.com/antipatterns/functional-decomposition>

Полтергейст

только <https://sourcemaking.com/antipatterns/poltergeists>

- Это времененная прослойка для вызова методов другого класса, единственная задача -- вызвать методы другого.
- Не имеет состояния.
- Живет непродолжительно.
- Класс с подобием управления

ЗОЛОТОЙ МОЛОТОК

конспект

Решение:

- SOAP, REST
- Middleware (связующее программное обеспечение)
 - вызов удаленных процедур (RPC, англ. *remote procedure call*)
- «кросс-технологические» инструменты коммуникации
 - Protobuf/gRPC

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

Остров автоматизации

Stovepipe system

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

Привязка к поставщику

Подразумеваемая архитектура

Программирование комитетом

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,
ограничивают полёт фантазии архитектора
3. приводят к желаемым положительным качествам получаемой системы.

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

Преимущества

Основные характеристики стилей

Известные стили

Трехзвенная архитектура

Многие приложения могут целиком быть реализованы по трёхзвенной схеме, причём ни в одном из этих звеньев не будет содержательной архитектурной сложности, тогда это что-то вроде стиля. Если трёхвенка — это только способ решения одной из задач, то это архитектурный шаблон

Архитектурные паттерны

Решение какой-либо задачи во всей системе, например, взаимодействие с пользователем.

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

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

Слоистый

Oc

Структурный

Клиент сервер

Вырожденный стиль слоистого стиля.

59. Пакетная обработка, каналы и фильтры, Blackboard.

Пакетная обработка

Канал и фильтры

Если не принимать в расчёт

типовизированные каналы, то вообще любые два фильтра можно использовать вместе.

Если принимать, то любые два фильтра, у которых подходящие типы «портов», можно использовать

Это значит, что фильтры можно *композировать*.

Blackboard

60. Событийно-ориентированные стили, Publish-Subscribe.

Стили с неявным вызовом

Издатель-подписчик

Событийно-ориентированное

C2

CORBA

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

Единый язык

Модель и реализация

Изоляция предметной области

Антипаттерн умный GUI

62. DDD, основные структурные элементы модели предметной области.

Жизненный цикл объекта

63. DDD, паттерн «Агрегат».

64. DDD, паттерны «Фабрика», «Репозиторий».

Хранилище

65. Говорящие интерфейсы, функции без побочных эффектов, assertions, замкнутые операции.

“10. Гибкая архитектура” в 11 ddd

66. Ограниченный контекст, непрерывная интеграция, карта контекстов.

Проблемы

Принципы поддержания целостности

67. Подходы к интеграции контекстов.

Типовые ситуации интеграции контекстов

Общее ядро

Заказчик-поставщик

Конформист

Предохранительный уровень

Ещё

Итог шаблонов интеграции

68. Смыслоное ядро, приёмы дистилляции, абстрактное ядро.

Дистилляция

Приемы дистилляции

Абстрактное ядро

69. Крупномасштабная структура, метафора системы, разбиение по уровням.

Метафора системы

Уровневая структура

70. Типичные уровни в производственных системах

Система автоматизации

71. Типичные уровни в финансовых системах

72. Стили «Уровень знаний», «Подключаемые компоненты».

73. Архитектура распределённых систем: понятие распределённой системы, типичные архитектурные стили.

Общий ресурс -- данные, выч ресурсы для тонких клиентов

Заблуждения

- Сеть надежна
- Задержка равна нулю
- Пропускная способность бесконечна
- Сеть безопасна -- даже боты могут взломать
- Топология сети неизменна
 - Система должна подстраиваться под изменение
- Администрирование сети централизованно
 - Школьникам нельзя поставить проги, т.к. заблокена запись
- Передача данных "бесплатна"
- Сеть однородна
 - Мобильные сети
 - спутниковые
 - локальная в универс и дома

Архитектура распределенных систем

Виды сущностей

Объекты -- ощущение, будто бы объект у нас в памяти
Компоненты -- это набор объектов

Виды взаимодействия

- RPC -- как сер-десер параметры, что такое вызов, разные семантики вызова и возврата значений
- RMI -- **плюсом** знает об ООП: передача ссылок, хранение ссылок. Исключение

Роли и обязанности

- Клиент-сервер
- Peer-to-Peer

Варианты размещения

1. Балансировщик разбивает клиентов на сервисы
 - a. проблема синхронизации БД
 - i. Либо блочим всё и уведомляем об изменениях
 - ii. Либо синхронизируем когда-нибудь потом
2. Кэширование
 - a. Чтобы не добираться до сервера
3. Выдается код
 - a. Браузер
 - b. юзается выч ресурсы клиента
 - c. безопасность
4. Выдается целое приложение
 - a. юзается выч ресурсы клиента
 - b. безопасность

Типичные архитектурные стили

74. Межпроцессное сетевое взаимодействие, модель OSI, стек протоколов TCP/IP, сокеты, протоколы «запрос-ответ», протокол HTTP.

Несколько IP для одного компа -- ДА

Несколько одинаковых номеров портов на одном компе -- ДА (TCP, UDP)

ping -- это уровень IP, а порт -- уровень TCP/UDP

Протокол запрос-ответ

- Сервер не хранит состояние
- wait -- async/await C#

Запрос ответ поверх UDP

- Редкие запросы -- не нужно управлять потоком
- хранение “истории” на стороне **сервера** -- чтобы не выполнять два раза операцию от двух расщепленных пакетов

Запрос ответ поверх TCP

- посылка **потока байт** а не пакетов
- сетевой ввод-вывод -- непонятно когда конец
 - нужно отдельным сообщением говорить о конце передачи
- лавинообразный крах сети

HTTP

Протокол передачи html-страниц изначально
затем -- протокол для передачи данных

Общение может выполняться по одному каналу долгое время

Маршаллинг - это сер-десер

HTTP 1 -- текстовый

HTTP 2 -- бинарный

НО для программирования сервисов лучше использовать RPC, RMI

75. Удалённые вызовы процедур (RPC). Protobuf, gRPC.

На клиенте генерируются заглушки по описанию процедур сервера (описание общее для различных ЯП). Заглушки сериализуют параметры для сервера и посылают запрос.

Сервер десер параметры, вызывает нужную процедуру и посыпает ответ обратно

Пишутся крупные сервисы, обеспечивающие функциональность для других, которая вызывается через RPC

Прозрачность вызова

Явно маркируют удаленные вызовы, чтобы понимать, что этот код нужно обезопасить

Структура RPC middleware

Client stub procedure -- **сгенерированный**

Communication module -- *рукописный*, является частью RPC библиотеки

Dispatcher -- *рукописный*, является частью библиотеки

Server stub procedure -- **сгенерирован**

Server procedure -- *рукописно*

protobuf

НЕ про удаленный вызов

ПРО сер-десер

НЕ обязательно по сети передавать. Можно сейвить данные на компе

Старается максимально компактно (**бинарное представление**) -- это набор байт

Языконезависимый формат данных. По описанию protobuf compiler генерит

- структуры на конкретном ЯП
- код сериализатора-десериализатора

Отличается от JSON, XML следующим

- XML: (юзались обычно в локалке => проблем с большими XML файлами с трафиком впустую не было)
- JSON: кусок JS по описанию объекта так, как он бы использовался в JS
 - просто исполняешь JSON и получаешь объект в памяти
 - компактнее чем XML, но можно лучше (т.к. доветочие, скобочки)
 - Это protobuf

PROTOBUF

набор байт

example кодирования: числа кодируются байтами (1 бит -- есть ли далее продолжение числа)

Данный файл .proto компилируется через protoC в .java

- ! Нумерация полей -- вручную. При удалении другие номера не меняются.
 - Для обратной совместимости со старыми версиями: знаем номер -- знаем что за поля

Сборка -- это плагин protoc в nuget

gRPC

Поверх protobuf.

На разных языках мб и **клиент, и сервер**

У клиента заглушки генерятся по gRPC описанию сервиса,
Описание сервиса -- в **том же файле**, что и описание данных protobuf -- файле **.proto**

На сервере генерится скелет. Его нужно реализовать руками.

Сборка -- плагин grpc к protoc

Структура скелета:

- Возвращать хотя бы empty message. **Всегда что-то возвращается.**
- stream -- открывается соединение и передаётся поток байтов

Пример на джаве. Сервер:

onNext -- grpc решит передавать пачками или по одной.
onCompleted -- то, что накопилось, отправляется

Передача в обе стороны СЛОЖНОВАТО....

Клиент:
Конфигурация

Вызовы методов:

76. Удалённые вызовы методов (RMI).

Есть exactly once.

- Хранит общую структуру данных для хранение идентифицируемых объектов разных компонентов
 - марширует объект и передает по сети
 - или предоставляет удаленный доступ
- RMI считает ссылки -- удаляет объекты без ссылок и удаляет из всей системы
 - требуется поддержка от VM

Структура RMI middleware

proxy -- это объект, вызовы методов которого -- это удаленные вызовы по сети ерез RMI

remote reference module -- на каждой машине модуль хранения ссылок
communication -- сер-десер запросы

Скелет -- генерится по описанию объекта и дергает методы настоящего объекта
Dispatcher

77. Веб-сервисы, SOAP. WCF.

SOAP -- архитектурно как семейство протоколов на основе XML.
Очень много overhead

- SOAP
 - решает вопросы то же, что и grpc
 - знает про ООП, про исключения
 - на XML => xml может проверять корректность
 - grpc может распарситься в полный хлам
- WSDL
 - Описывает что умеет делать сервис
 - для генерации заглушек
 - Подробнее grpc-описания сервиса
 - знает про ООП
 - не открывается обычно, т.к. сервис могут сломать, если о нём много знают
- UDDI
 - поиск и публикация серверов

SOAP сообщение

WSDL описание

portType -- объект из ООП, имеющий операции

Достоинства SOAP-based сервисов

не нужно писать руками файлы

Проверка по схеме -- передача строки вместо числа

Проверка SOAP-сервером -- проверяет, например, что переданы все поля, которые нужны, все имеют правильный тип

Недостатки SOAP-based сервисов

Сложность описаний: классы приходится размечать атрибутами

Сложность миграции: как -- деплоить несколько версий

WCF -- Пример поддержки SOAP

Address -- идентификация сервиса

Binding -- как связываться с сервисом, как соединяться

Contract -- о делает сервис

Пример описания контракта

Пример self-hosted service

Пример клиент

<http://localhost...> -- загружает отсюда WSDL

Конфиг на стороне клиента

Генерируется автоматически

78. Очереди сообщений, RabbitMQ.

Очереди сообщений

- Буфер между получателем и отправителем
- Распределитель нагрузки

RabbitMQ

Сервер на ERLANG + набор клиентский библиотек.

using:

- Подключаем клиентскую библиотеку к своему *приложению*
- Поднять свой сервер
- Коннектиться *приложению* к поднятым серверу
- Сообщение в виде **байтового массива**

- Сериализация сообщений -- наше дело (например, protobuf)
-
- На клиенте можно сконфигурировать маршрутизацию сообщений
- На клиенте описать именованную очередь и постим туда сообщения
-
- На сервере описать чтение сообщения из такой-то очереди, вызывая коллбэк
 - либо сам забираю сообщения раз n-ое кол-во времени

Пример отправителя

Создаем exchange с именем hello

- queue -- имя очереди
 - если очередь есть на сервере, то новая не создается
- durable -- хранение на диске
- exchange -- сервис маршрутизации сообщений по очередям.
 - по умолчанию 1 exchange -- 1 очередь
- routingKey -- имя очереди куда отправляем

Пример получатель

BasicConsume -- после включения ожидания надо выключать ожидание сообщений самим (в примере прога будет висеть вечно).

79. Архитектурный стиль REST.

НЕ протокол

ЭТО ПРИНЦИП организации сервисов.

Противовес тяжеловесному SOAP.

Идеология: Хотим в REST делать все проще и быстрее, не нагружая сервер

REST

- Про состояние
 - Хранит данные клиента
 - Но не хранит инфу о сессии с клиентом (**ex:** IP клиента)
 - Каждый запрос с разных IP от одного клиента обрабатывается так, что мы общаемся именно с ЭТИМ клиентом
- можно кешировать между клиентом и сервером и не беспокоить сервер

EX Google Doc -- REST сервер.

На WCF -- один сервер все решает

На REST -- много серверов решают задачи.

Интерфейс сервиса

GET -- получение элементов из коллекции (всегда без тела)

PUT -- добавление элемента в коллекцию (в body -- большой файл можно)

POST -- модификация эл в коллекции (в теле -- большой файл можно)

Форматы сери-ции:

- json
- protobuf
- xml -- редко (ибо тяжеловесно)

Access token -- строка после аутентификации для проверки пользователя.

Быстро протухает за десятки минут--часы

- REST не публикует метаинфу о сервисе => нельзя сгенерить заглушку, как в SOAP
- Вся инфа -- человекочитаемая документация. Поставляется вместе с сервисом.
- Нужно писать клиентские библиотеки самим разработчикам сервисов, чтобы не писать самому страшные url REST-запросы
 - Поэтому клиенту руками запросы писать не нужно

Пример Google Drive REST API

Достоинства по сравнению с SOAP

- Надежность
 - сложнее налажать в протоколе
 - легко отлаживать
- Производительность
 - не нужно парсить XML, сравнение схем XML
- Масштабируемость
 - Запрос имеет в себе всю инфу для его обработки
 - Один запрос может обрабатываться разными сервисами
 - запросили один, ответил другой
- Прозрачность системы взаимодействия
 - можно прямо из браузера писать запросы через url
- Простота интерфейсов
 - инфу о сессиях не хранят => перекидывание запросов на другие сервисы
- Портативность компонент

- можно менять сервисы, клиенты могут не заметить
- Но сильные изменения -- деплоить несколько версий

80. Микросервисы, peer-to-peer.

На REST лучше всего строить микросервисную архитектуру. REST легок и микросервисы легки.

Они используют REST, чтобы описать правила построения распределенных систем

SOAP не подходит, т.к. он тяжел, а общаться надо много.

Микросервисы

Это набор небольших сервисов, которые можно переписать с нуля за неделю.

Деплоятся через Docker, Docker-compose

Используют облачные средства: Amazon, Azure

Монолитные приложения. Минусы по сравнению с микросервисами

+ Плюсы:

Нет оверхеда на сеть

Не разводят зоопарк технологий

Преимущество разбиения на сервисы

Основные особенности

- ▶ Микросервисы и SOA

Не противопоставляются друг другу. Просто на SOA стали делать огромные приложения

- ▶ Smart endpoints and dumb pipes

Общение -- это просто обмен сообщениями. Логика в самих сервисах

- ▶ Проектирование под отказ

Каждый сервис может оставаться в живых один

- ▶ Асинхронные вызовы

Сервис может вызвать сервис может вызвать сервис...

- ▶ Децентрализованное управление данными

Чаще сервис держит свои данные у себя

- ▶ Автоматизация инфраструктуры

- ▶ Эволюционный дизайн

Основные проблемы

- ▶ Сложности выделения границ сервисов

Что кому поручить?

- ▶ Перенос логики на связи между сервисами

- ▶ Большой обмен данными

- ▶ Нетривиальные зависимости

Граф зависимостей, который нигде не видно

- ▶ Нетривиальная инфраструктура

docker, kubernetes, vm, deploy.

- ▶ Нетривиальная переиспользуемость кода

Может быть ваще не возможно и будет куча копипасты

Peer-to-peer

- ▶ Децентрализованный и самоорганизующийся сервис
Каждый компонент полностью решает задачу. Но чем больше вычислителей, тем система работает лучше
- ▶ Динамическая балансировка нагрузки
 - ▶ Вычислительные ресурсы
 - ▶ Хранилища данных
- ▶ Динамическое изменение состава участников

Пример, обмен музыкой

Узнают друг от друга через центральный узел.
Центр однажды забрал черный фургон и система схлопнулась

Пример, Skype

SuperNode -- выход в инет, либо большие выч. ресурсы

Нужна авторизация в LoginServer (можно через super-node). затем общение с пирами напрямую

Со временем supernode переехали на сервера microsoft, ибо:

- юзают ресурсы клиентов зачем-то без разрешения
- контроль трафика

Настоящий peer-to-peer BitTorrent

Основана на DHT distributed hash table. (Информация распределяется среди некоторого набора узлов-хранителей, и восстанавливать их путём распределённого поиска по ключу.)
Не нужно центральное хранилище с информацией откуда нужно забрать контент, т.к.:

По хэшу файла, опрашивая соседние пирры, узнаем есть ли у них файл или нет.

Бестрекерная реализация.

BitTorrent-трекер — сервер, осуществляющий координацию клиентов BitTorrent.

DHT позволяет раздавать без трекера

Такая раздача называется **бестрекерной** (*trackerless*). Торрент для неё создаётся без адреса трекера и клиенты находят друг друга через DHT. Правда, при этом начавший раздачу должен иметь реальный IP-адрес, доступный извне.

DHT (англ. *distributed hash table* — «распределённая хеш-таблица») — это класс децентрализованных распределённых систем поисковой службы, работающей подобно хеш-таблице. Как структура данных, хеш-таблица может представлять собой

ассоциативный массив, содержащий пары (ключ-значение). Также, с термином DHT связан ряд принципов и алгоритмов, позволяющих записывать данные, распределяя информацию среди некоторого набора узлов-хранителей, и восстанавливать их путём распределённого поиска по ключу. Особенностью распределённой таблицы является возможность распределить информацию среди некоторого набора узлов-хранителей таким образом, что каждый участвующий узел смог бы найти значение, ассоциированное с данным ключом. Ответственность за поддержание связи между именем и значением распределяется между узлами, в силу чего изменение набора участников является причиной минимального количества разрывов. Это позволяет легко масштабировать DHT, а также постоянно отслеживать добавление и удаление узлов и ошибки в их работе.

81. Развёртывание и балансировка нагрузки, Docker.

- - Гостевая машина долго грузится да и много ресурсов занимает
- + Виртуалки хорошо, если надо исполнять пользовательский ввод, т.к. в VM большая изоляция
-
- - Контейнер имеет меньший уровень изоляции, чем виртуалка, т.к. можно выбраться из docker => sudo запрещают в dockerfile.
- + быстро запускается
- + легко переносим
- + если внутри контейнера только наши сервисы -- это отличный способ упаковки

Контейнер -- это **свойство** ОС.

Есть бесплатный репо -- DockerHub.
Поддерживают все оркестраторы

Docker image

Docker container

DockerHub

Базовые команды

Сборка образа -- Dockerfile

Балансировка нагрузки -- docker compose

Управление несколькими контейнерами.

Запускает n копий контейнеров.

Проверяет, что все запустили.

Можно юзать несколько машин для

Следить за выделяемым контейнерам ресурсами

Перезапускать при необходимости

Можно добавлять в именованные сети

n контейнеров должны быть запущены на разных машинах. На каждой машине стоит load balancer, который распределяет клиентов по контейнерам разных машин

82. Архитектура системы контроля версий Git.

83. Архитектура командной оболочки Bash.

84. Архитектура компьютерной игры Battle for Wesnoth.