

Архитектура ПО

Лекция 1: Об архитектуре

Юрий Литвинов
yurii.litvinov@gmail.com

01.09.2020г

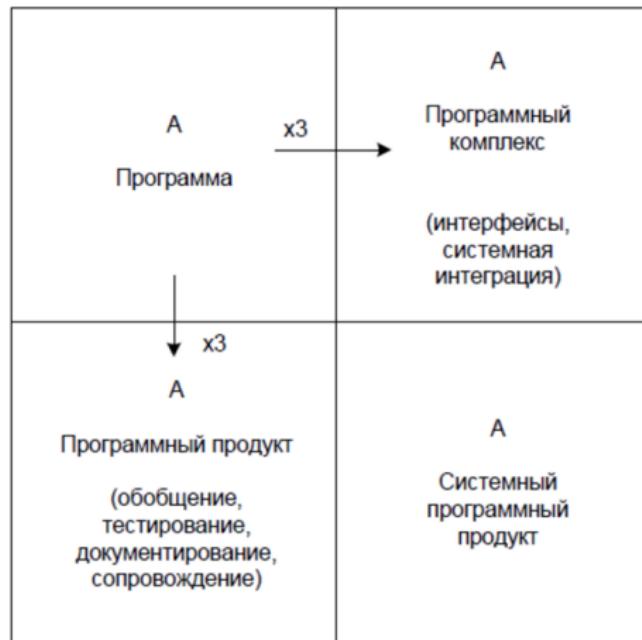
Организационное

- ▶ Лекционный курс, с одной практической работой
- ▶ В конце экзамен
 - ▶ Примерно 80 коротких вопросов всего, в билете два
 - ▶ Литературой пользоваться можно
 - ▶ Будут допы
 - ▶ Если не готовиться, можно получить трояк или вынос
- ▶ Три домашних задания «на потренироваться»
- ▶ ECTS-like балльная система
 - ▶ 30% за домашки, 10% за практическую работу, 80% за экзамен
- ▶ Материалы будут выкладываться в Blackboard

Что будет в курсе

- ▶ Объектно-ориентированное проектирование
- ▶ Моделирование, язык UML и, немножко, другие визуальные языки
- ▶ Шаблоны проектирования и антипаттерны
- ▶ Архитектурные стили
- ▶ Предметно-ориентированное проектирование
- ▶ Проектирование распределённых приложений
- ▶ Примеры архитектур

Программа и программный продукт



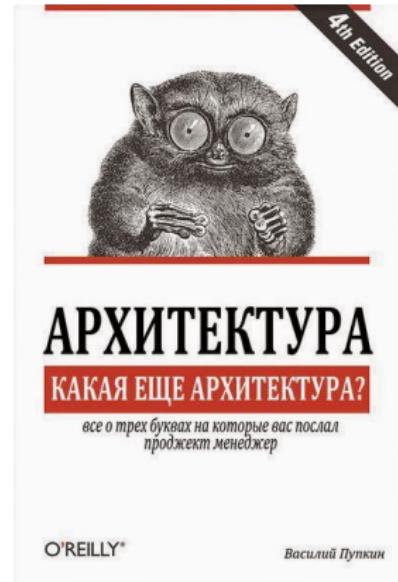
Размер типичного ПО

Простая игра для iOS	10000 LOC
Unix v1.0 (1971)	10000 LOC
Quake 3 engine	310000 LOC
Windows 3.1 (1992)	2.5M LOC
Linux kernel 2.6.0 (2003)	5.2M LOC
MySQL	12.5M LOC
Microsoft Office (2001)	25M LOC
Microsoft Office (2013)	45M LOC
Microsoft Visual Studio 2012	50M LOC
Windows Vista (2007)	50M LOC
Mac OS X 10.4	86M LOC

<http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>

Архитектура

- ▶ Совокупность важнейших решений об организации программной системы
 - ▶ Эволюционирующий свод знаний
 - ▶ Разные точки зрения
 - ▶ Разный уровень детализации
- ▶ Для чего
 - ▶ База для реализации, «фундамент» системы
 - ▶ Инструмент для оценки трудоёмкости и отслеживания прогресса
 - ▶ Средство обеспечения переиспользования
 - ▶ Средство анализа системы ещё до того, как она реализована



Профессия «Архитектор»

- ▶ Архитектор — специально выделенный человек (или группа людей), отвечающий за:
 - ▶ разработку и описание архитектуры системы
 - ▶ доведение её до всех заинтересованных лиц
 - ▶ контроль реализации архитектуры
 - ▶ поддержание её актуального состояния по ходу разработки и сопровождения

Профессиональный стандарт «Архитектор»

Создание и сопровождение архитектуры программных средств, заключающейся

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

Трудовые функции архитектора

По стандарту

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

Архитектор vs разработчик



- ▶ Широта знаний
- ▶ Коммуникационные навыки
- ▶ Часто архитектор играет роль разработчика и наоборот
 - ▶ Архитектор в «башне из слоновой кости» — это плохо

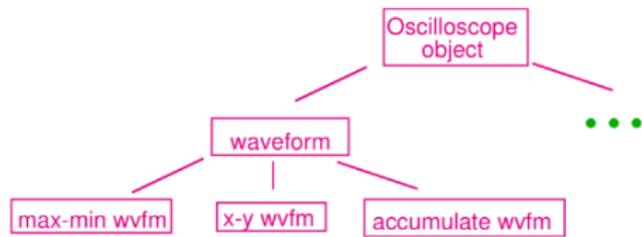
Пример: ПО для осциллографа

- ▶ Считывать параметры сигнала
- ▶ Оцифровывать и сохранять их
- ▶ Выполнять разные фильтрации и преобразования
- ▶ Отображать результаты на экране
 - ▶ С тач-скрином и встроенным хелпом
- ▶ Возможность настройки под конкретные задачи

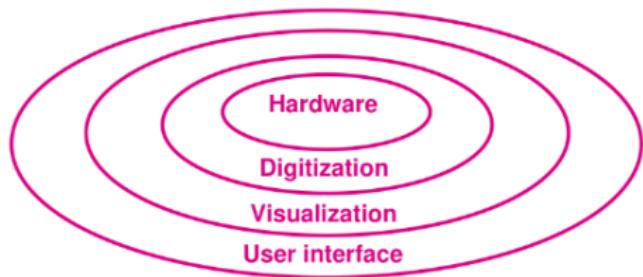


По статье Garlan D., Shaw M. An introduction to software architecture

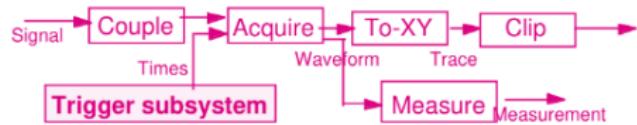
Вариант 1: объектная модель



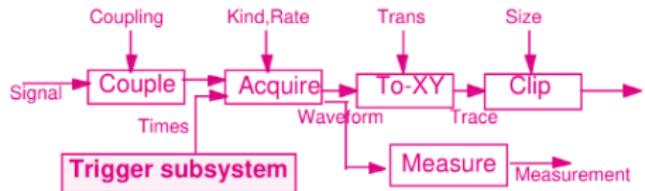
Вариант 2: слоистая архитектура



Вариант 3: каналы и фильтры



Вариант 4: модифицированные каналы и фильтры



Выводы

- ▶ Можем делать утверждения о свойствах системы, базируясь на её структурных свойствах
 - ▶ Не написав ни строчки кода и даже не выбрав язык реализации
- ▶ Рассуждения очень субъективны
 - ▶ Многое зависит от интуиции и вкуса архитектора, однако ошибки очень дороги
- ▶ Можно выделить *архитектурные стили* — «архитектуры архитектур»
- ▶ Можно выделить *архитектурные точки зрения* и *архитектурные виды*
- ▶ Разный уровень детализации

Архитектурные виды

Стандарт IEEE 1016-2009

- ▶ Контекст — фиксирует окружение системы
 - ▶ Диаграммы активностей UML, IDEF0 (SADT)
- ▶ Композиция — описывает крупные части системы и их предназначение
 - ▶ Диаграммы компонентов UML, IDEF0
- ▶ Логическая структура — структура системы в терминах классов, интерфейсов и отношений между ними
 - ▶ Диаграммы классов UML, диаграммы объектов UML

Архитектурные виды (2)

- ▶ Зависимости — определяет связи по данным между элементами
 - ▶ Диаграммы компонентов UML, диаграммы пакетов UML
- ▶ Информационная структура — определяет персистентные данные в системе
 - ▶ Диаграммы классов UML, IDEF1x, ER, ORM
- ▶ Использование шаблонов — документирование использования локальных паттернов проектирования
 - ▶ Диаграммы классов UML, диаграммы пакетов UML, диаграммы коллокации UML

Архитектурные виды (3)

- ▶ Интерфейсы — специфицирует информацию о внешних и внутренних интерфейсах
 - ▶ IDL, диаграммы компонентов UML, макеты пользовательского интерфейса, неформальные описания сценариев использования
- ▶ Структура системы — рекурсивное описание внутренней структуры компонентов системы
 - ▶ Диаграммы композитных структур UML, диаграммы классов UML, диаграммы пакетов UML
- ▶ Взаимодействия — описывает взаимодействие между сущностями
 - ▶ Диаграммы композитных структур UML, диаграммы взаимодействия UML, диаграммы последовательностей UML

Архитектурные виды (4)

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

Ещё про архитектурные виды

- ▶ Пример — http://robotics.ee.uwa.edu.au/courses/design/examples/example_design.pdf
- ▶ Ни один вид не обязателен
- ▶ Активно используются визуальные языки
 - ▶ В основном как дополнение к тексту
- ▶ Моделирование принципиально важно для архитектуры
 - ▶ Нельзя абстрагироваться от сложности, но можно декомпозировать сложность

Роль архитектуры в жизненном цикле

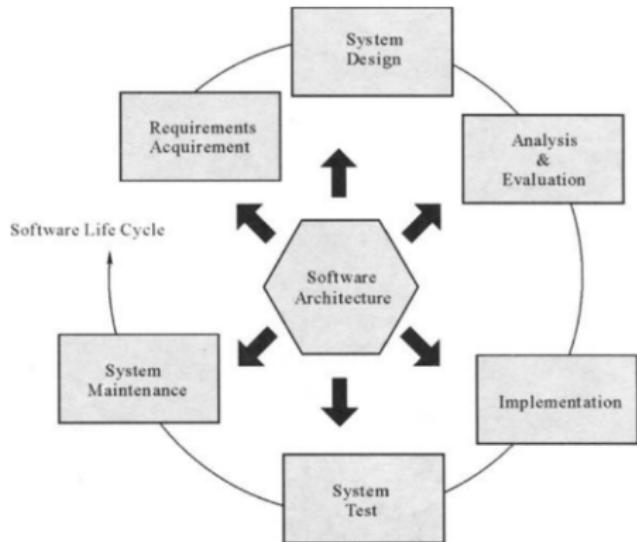
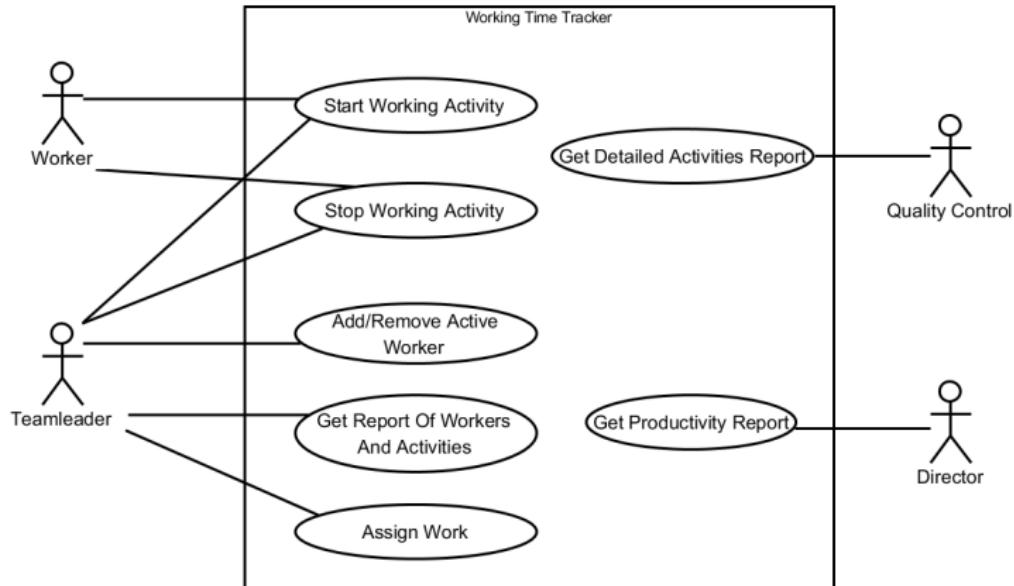


Рисунок из Z. Quin, "Software Architecture"

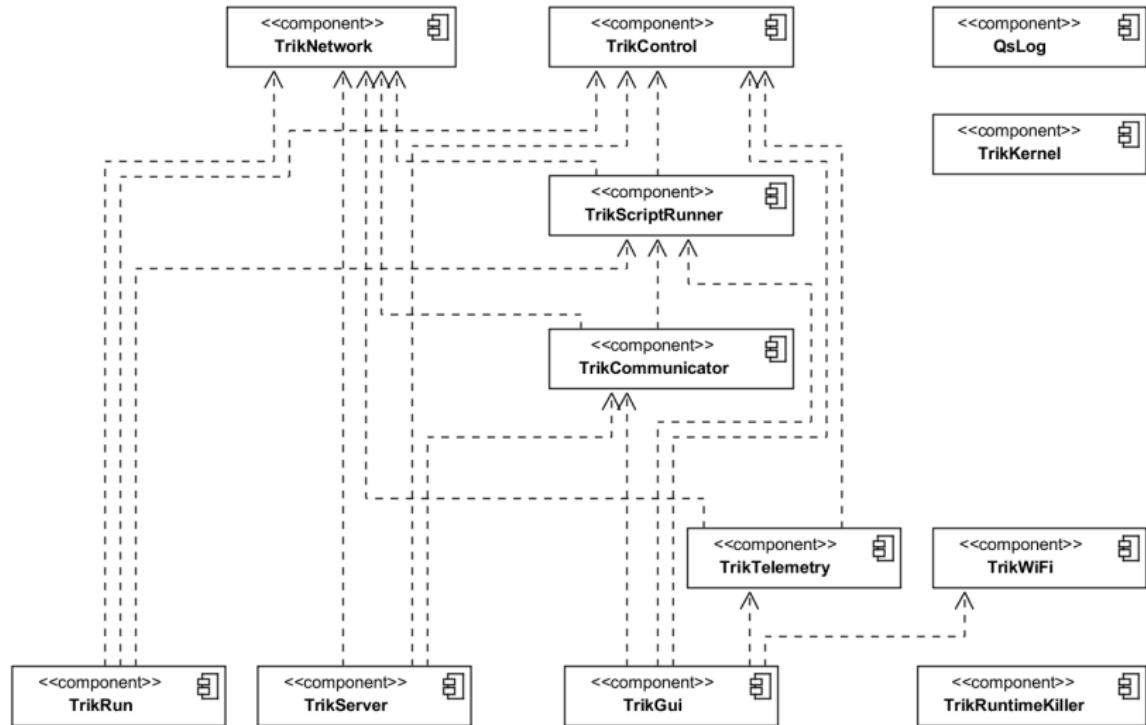
Архитектура и требования



Требования

- ▶ Функциональные — то, что система должна делать
- ▶ Нефункциональные — то, как система должна это делать
 - ▶ Эффективность
 - ▶ Масштабируемость
 - ▶ Удобство использования
 - ▶ Надёжность
 - ▶ Безопасность
 - ▶ Сопровождаемость и расширяемость
 - ▶ ...
- ▶ Ограничения
 - ▶ Технические
 - ▶ Бизнес-ограничения

Архитектура и проектирование



Архитектура и проектирование — задачи

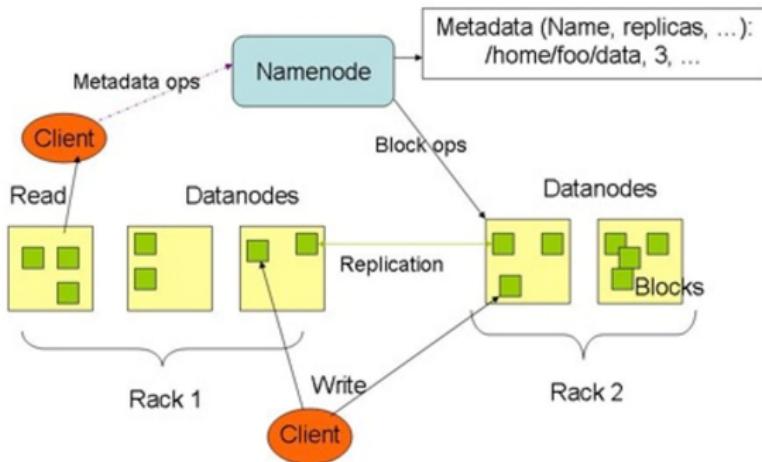
- ▶ Декомпозиция задачи
- ▶ Определение границ компонентов
- ▶ Определение интерфейсов между компонентами
- ▶ Общие для всей системы вопросы
 - ▶ Стратегия обработки ошибок
 - ▶ Стратегия логирования
 - ▶ Стратегия обновлений
 - ▶ Стратегия разделения доступа
 - ▶ Вопросы локализации
 - ▶ ...
- ▶ Анализ и верификация архитектуры

Архитектура и разработка

- ▶ *prescriptive architecture* — архитектура, как её определил архитектор
- ▶ *descriptive architecture* — архитектура, как она есть в системе
 - ▶ Архитектура у ПО есть всегда, как вес у камня
- ▶ *architectural drift* — «сползание» фактической архитектуры
 - ▶ появление в ней важных решений, которых нет в описательной архитектуре
- ▶ *architectural erosion* — «размывание» архитектуры
 - ▶ отклонения от описательной архитектуры, нарушения ограничений
- ▶ Антипаттерн «*Big ball of mud*» — результат эрозии

Пример: Hadoop

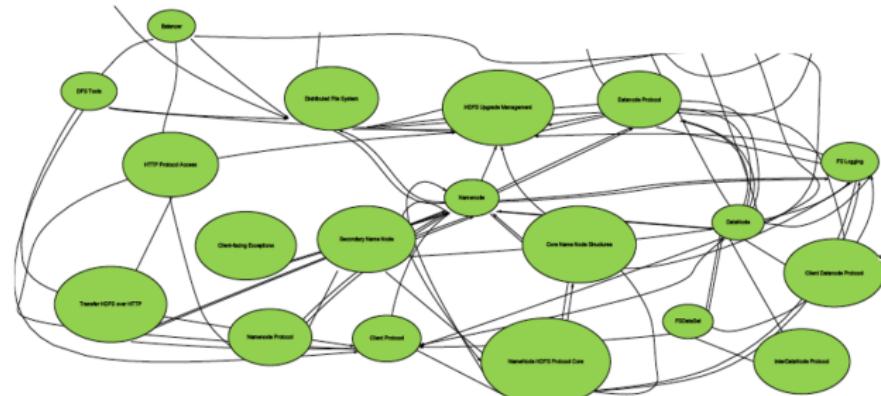
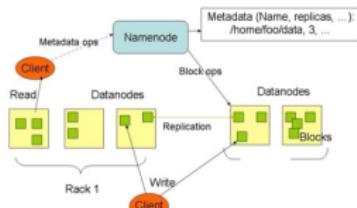
As-designed



Special thanks to prof. Nenad Medvidovic (USC) for kind permission for using his slides

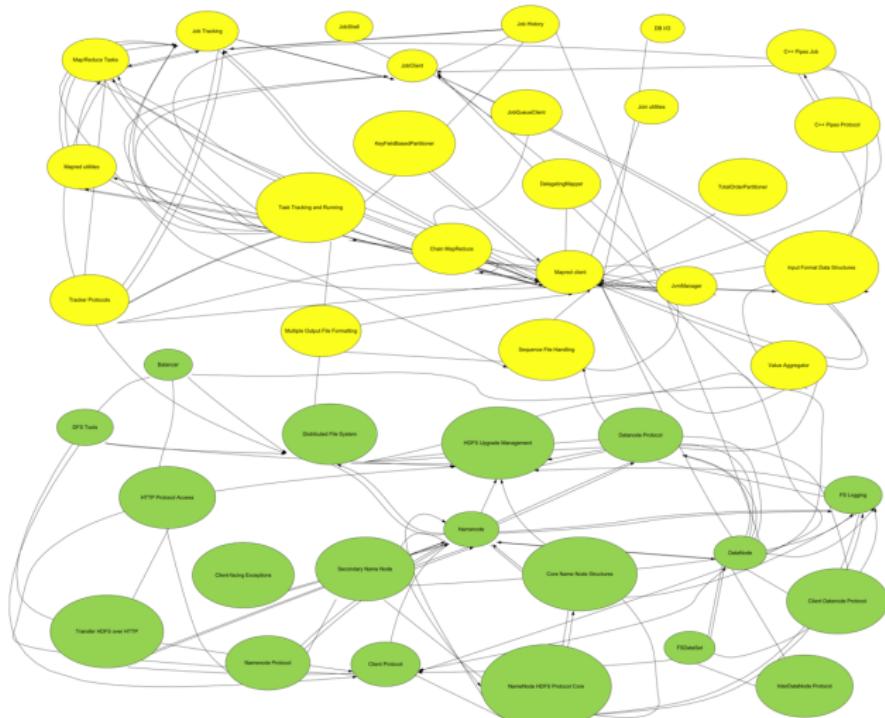
Hadoop as-built

HDFS



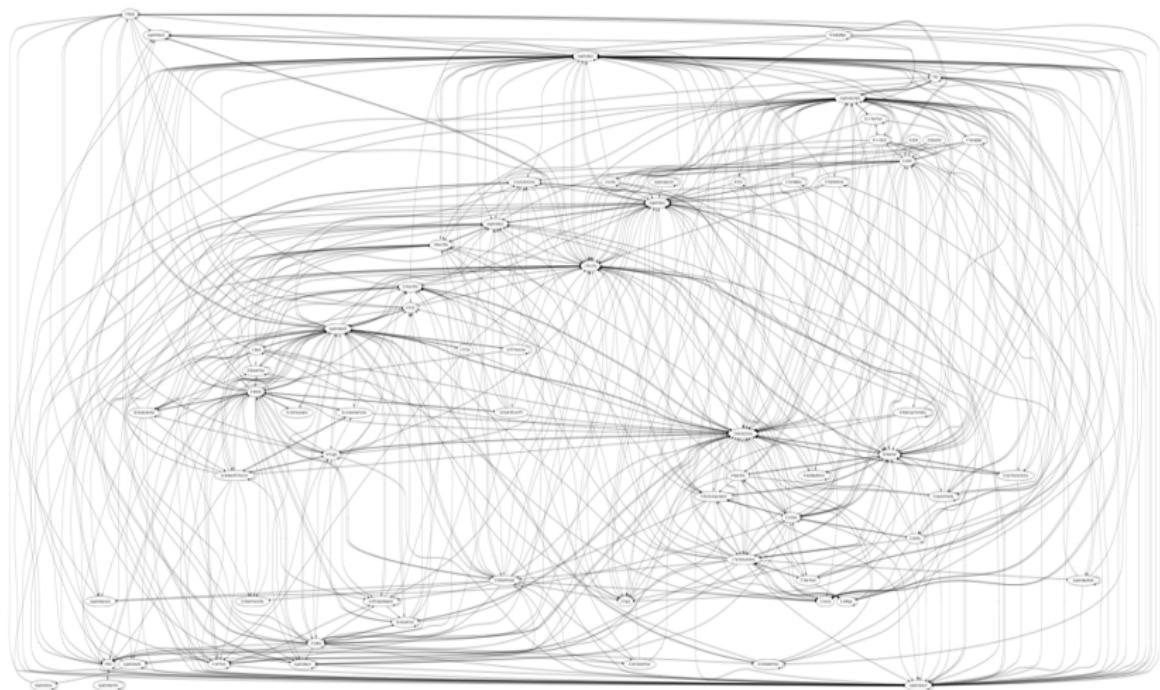
Hadoop as-built

HDFS + MapReduce



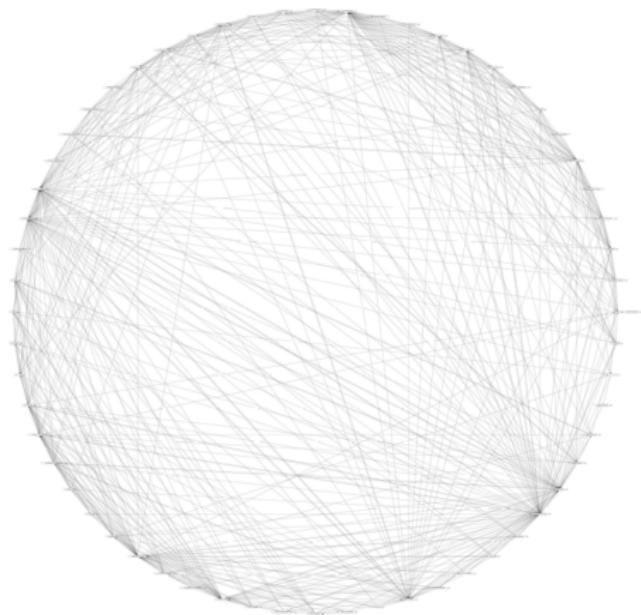
Hadoop as-built

Полная архитектура



Hadoop as-built

Граф зависимостей



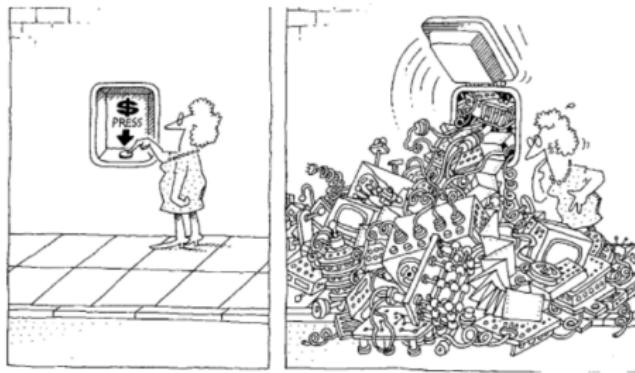
Лекция 2: Декомпозиция, объектно-ориентированное проектирование

Юрий Литвинов
yurii.litvinov@gmail.com

08.09.2020г

Сложность

- ▶ **Существенная сложность** (essential complexity) — сложность, присущая решаемой проблеме; ею можно управлять, но от неё нельзя избавиться
- ▶ **Случайная сложность** (accidental complexity) — сложность, привнесённая способом решения проблемы



© G. Booch, "Object-oriented analysis and design"

Свойства сложных систем

- ▶ Иерархичность — свойство системы состоять из иерархии подсистем или компонентов
 - ▶ Декомпозиция
- ▶ Наличие относительно небольшого количества видов компонентов, экземпляры которых сложно связаны друг с другом
 - ▶ Выделение общих свойств компонентов, абстрагирование
- ▶ Сложная система, как правило, является результатом эволюции простой системы
- ▶ Сложность вполне может превосходить человеческие интеллектуальные возможности

Подходы к декомпозиции

- ▶ Восходящее проектирование
 - ▶ Сначала создаём “кирпичики”, потом собираем из них всё более сложные системы
- ▶ Нисходящее проектирование
 - ▶ Постепенная реализация модулей
 - ▶ Строгое задание интерфейсов
 - ▶ Активное использование “заглушек”
 - ▶ Модули
 - ▶ Четкая декомпозиция
 - ▶ Минимизация
 - ▶ Один модуль — одна функциональность
 - ▶ Отсутствие побочных эффектов
 - ▶ Независимость от других модулей
 - ▶ Принцип сокрытия данных

Модульность

- ▶ Разделение системы на компоненты
- ▶ Потенциально позволяет создавать сколь угодно сложные системы
- ▶ Строгое определение контрактов позволяет разрабатывать независимо
- ▶ Необходим баланс между количеством и размером модулей



Сопряжение и связность

- ▶ **Сопряжение (Coupling)** — мера того, насколько взаимозависимы разные модули в программе
- ▶ **Связность (Cohesion)** — степень, в которой задачи, выполняемые одним модулем, связаны друг с другом
- ▶ Цель: слабое сопряжение и сильная связность

Объекты

- ▶ Objects may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods — **Wikipedia**
- ▶ An object stores its state in fields and exposes its behavior through methods — **Oracle**
- ▶ Each object looks quite a bit like a little computer — it has a state, and it has operations that you can ask it to perform — **Thinking in Java**
- ▶ An object is some memory that holds a value of some type — **The C++ Programming Language**
- ▶ An object is the equivalent of the quanta from which the universe is constructed — **Object Thinking**

Объекты

- ▶ Имеют
 - ▶ Состояние
 - ▶ Инвариант
 - ▶ Поведение
 - ▶ Идентичность
- ▶ Взаимодействуют через посылку и приём сообщений
 - ▶ Объект вправе сам решить, как обработать вызов метода (**полиморфизм**)
 - ▶ Могут существовать в разных потоках
- ▶ Как правило, являются экземплярами **классов**

Абстракция

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

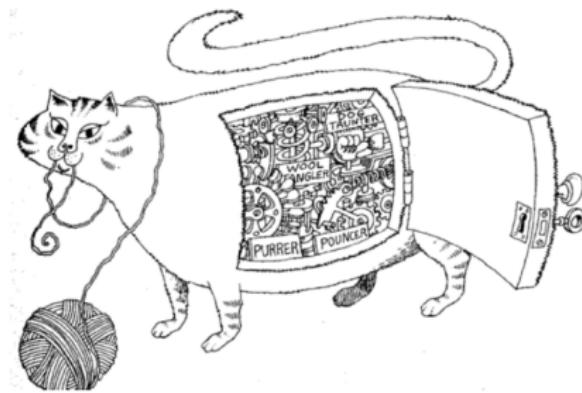


© G. Booch, “Object-oriented analysis and design”

Инкапсуляция

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

Инкапсуляция защищает **инварианты** абстракции



© G. Booch, "Object-oriented analysis and design"

Наследование и композиция

▶ Наследование

- ▶ Отношение “Является” (is-a)
- ▶ Способ абстрагирования и классификации
- ▶ Средство обеспечения полиморфизма

▶ Композиция

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

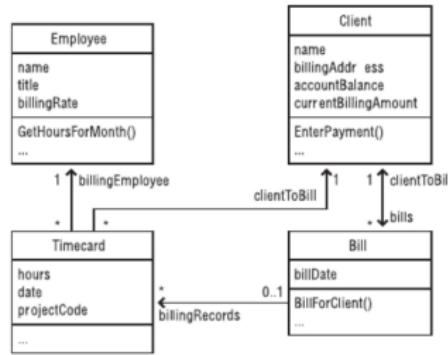
▶ Более-менее взаимозаменяемы

- ▶ Объект-потомок на самом деле включает в себя объект-предок
- ▶ Композиция обычно предпочтительнее

Определение объектов реального мира

Объектная модель предметной области

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



Изоляция сложности

- ▶ Сложные алгоритмы могут быть инкапсулированы
- ▶ Сложные структуры данных — тоже
- ▶ И даже сложные подсистемы
- ▶ Надо внимательно следить за интерфейсами



Изоляция возможных изменений

- ▶ Потенциальные изменения могут быть инкапсулированы
- ▶ Источники изменений
 - ▶ Бизнес-правила
 - ▶ Зависимости от оборудования и операционной системы
 - ▶ Ввод-вывод
 - ▶ Нестандартные возможности языка
 - ▶ Сложные аспекты проектирования и конструирования
 - ▶ Третьесторонние компоненты
 - ▶ ...

Изоляция служебной функциональности

- ▶ Служебная функциональность может быть инкапсулирована
 - ▶ Репозитории
 - ▶ Фабрики
 - ▶ Диспетчеры, медиаторы
 - ▶ Статические классы (*Сервисы*)
 - ▶ ...

Принципы SOLID

- ▶ Single responsibility principle
- ▶ Open/closed principle
- ▶ Liskov substitution principle
- ▶ Interface segregation principle
- ▶ Dependency inversion principle

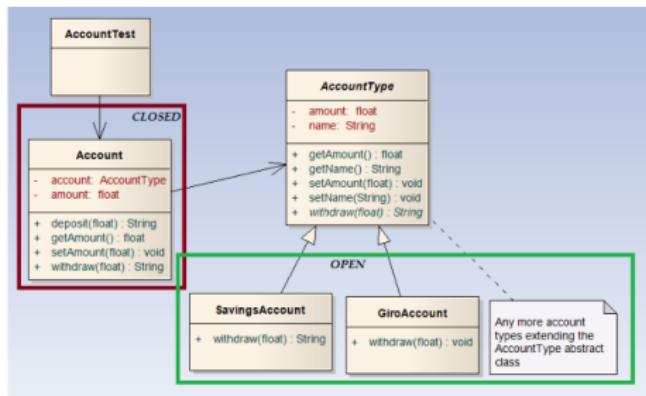
Single responsibility principle

- ▶ Каждый объект должен иметь одну обязанность
- ▶ Эта обязанность должна быть полностью инкапсулирована в объект



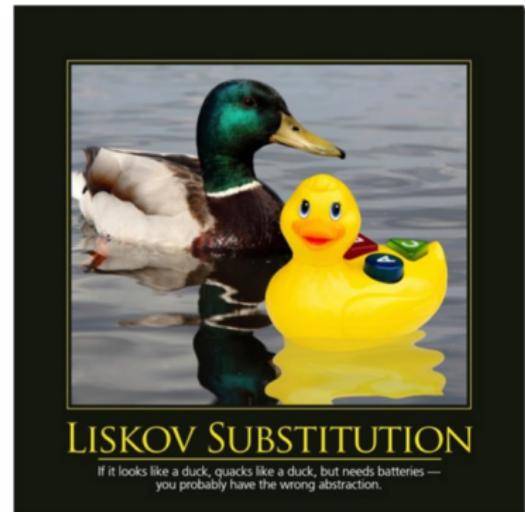
Open/closed principle

- ▶ Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения
 - ▶ Переиспользование через наследование
 - ▶ Неизменные интерфейсы



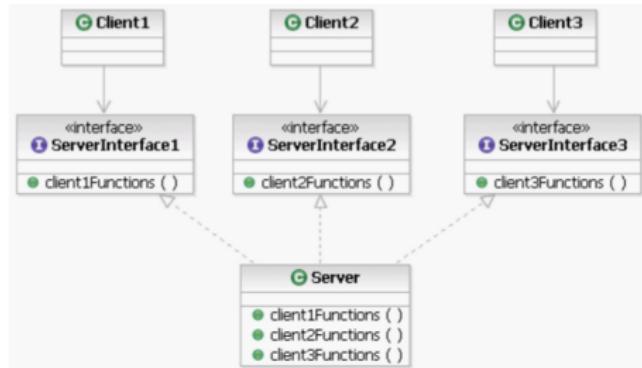
Liskov substitution principle

- ▶ Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом



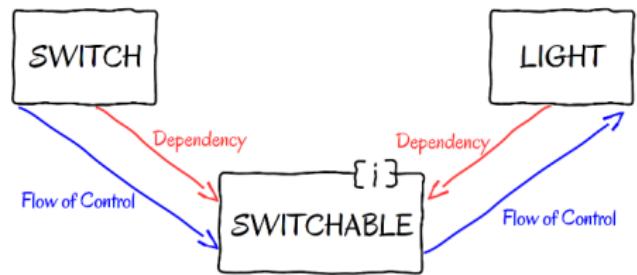
Interface segregation principle

- ▶ Клиенты не должны зависеть от методов, которые они не используют
 - ▶ Слишком “толстые” интерфейсы необходимо разделять на более мелкие и специфические



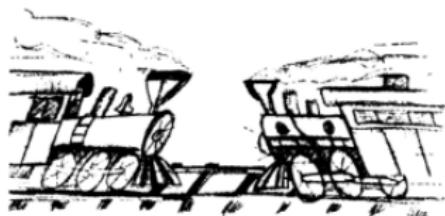
Dependency inversion principle

- ▶ Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций
- ▶ Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций



Закон Деметры

- ▶ “Не разговаривай с незнакомцами!”
- ▶ Объект А не должен иметь возможность получить непосредственный доступ к объекту С, если у объекта А есть доступ к объекту В, и у объекта В есть доступ к объекту С
 - ▶ book.pages.last.text
 - ▶ book.pages().last().text()
 - ▶ book.lastPageText()
- ▶ Иногда называют “Крушение поезда”



© Р. Мартин, “Чистый код”

Абстрактные типы данных

- ▶ `currentFont.size = 16` — плохо
- ▶ `currentFont.size = PointsToPixels(12)` — чуть лучше
- ▶ `currentFont.sizeInPixels = PointsToPixels(12)` — ещё чуть лучше
- ▶ `currentFont.setSizeInPoints(sizeInPoints)`
`currentFont.setSizeInPixels(sizeInPixels)` — совсем хорошо

Пример плохой абстракции

```
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() { ... }  
}
```

Пример хорошей абстракции

```
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() { ... }  
}
```

Ещё один пример абстракции

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

vs

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

Уровень абстракции (плохо)

```
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() { ... }  
}
```

Уровень абстракции (хорошо)

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

Общие рекомендации

- ▶ Про каждый класс знайте, реализацией какой абстракции он является
- ▶ Учитывайте противоположные методы (add/remove, on/off, ...)
- ▶ Соблюдайте принцип единственности ответственности
 - ▶ Может потребоваться разделить класс на несколько разных классов просто потому, что методы по смыслу слабо связаны
- ▶ По возможности делайте некорректные состояния невыразимыми в системе типов
 - ▶ Комментарии в духе “не пользуйтесь объектом, не вызвав init()” можно заменить конструктором
- ▶ При рефакторинге надо следить, чтобы интерфейсы не деградировали

Инкапсуляция

- ▶ Принцип минимизации доступности методов
- ▶ Паблик-полей не бывает:

```
class Point {  
    public float x;  
    public float y;  
    public float z;  
}
```

vs

```
class Point {  
    private float x;  
    private float y;  
    private float z;  
    public float getX() { ... }  
    public float getY() { ... }  
    public float getZ() { ... }  
    public void setX(float x) { ... }  
    public void setY(float y) { ... }  
    public void setZ(float z) { ... }  
}
```

Ещё рекомендации

- ▶ Класс не должен ничего знать о своих клиентах
- ▶ Лёгкость чтения кода важнее, чем удобство его написания
- ▶ Опасайтесь семантических нарушений инкапсуляции
 - ▶ “Не будем вызывать ConnectToDB(), потому что GetRow() сам его вызовет, если соединение не установлено” — это программирование сквозь интерфейс
- ▶ Protected- и package- полей тоже не бывает
 - ▶ На самом деле, у класса два интерфейса — для внешних объектов и для потомков (может быть отдельно третий, для классов внутри пакета, но это может быть плохо)

Наследование

- ▶ Включение лучше
 - ▶ Переконфигурируемо во время выполнения
 - ▶ Более гибко
 - ▶ Иногда более естественно
- ▶ Наследование — отношение “является”, закрытого наследования не бывает
 - ▶ Наследование — это наследование интерфейса (полиморфизм подтипов, *subtyping*)
- ▶ Хороший тон — явно запрещать наследование (*final-* или *sealed*-классы)
- ▶ Не вводите новых методов с такими же именами, как у родителя
- ▶ Code smells:
 - ▶ Базовый класс, у которого только один потомок
 - ▶ Пустые переопределения
 - ▶ Очень много уровней в иерархии наследования

Пример

```
class Operation {
    private char sign = '+';
    private int left;
    private int right;
    public int eval()
    {
        switch (sign) {           vs
            case '+': return left + right;
            case '-': return left - right;
        }
        throw new RuntimeException();
    }
}
```

```
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();
    }
}
```

Конструкторы

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

Мутабельность

Мутабельность — способность изменяться

- ▶ Запутывает поток данных
- ▶ Гонки

Чтобы сделать класс немутабельным, надо:

- ▶ Не предоставлять методы, модифицирующие состояние
 - ▶ Заменить их на методы, возвращающие копию
- ▶ Не разрешать наследоваться от класса
- ▶ Сделать все поля константными
- ▶ Не давать никому ссылок на поля мутабельных типов

Всё должно быть немутабельно по умолчанию!

Про оптимизацию

Во имя эффективности (без обязательности ее достижения) делается больше вычислительных ошибок, чем по каким-либо иным причинам, включая непроходимую тупость.

– William A. Wulf

Мы обязаны забывать о мелких усовершенствованиях, скажем, на 97% рабочего времени: опрометчивая оптимизация — корень всех зол.

– Donald E. Knuth

Что касается оптимизации, то мы следуем двум правилам:

Правило 1. Не делайте этого.

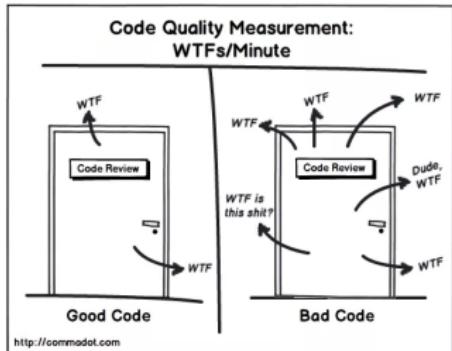
Правило 2 (только для экспертов). Пока не делайте этого – т.е. пока у вас нет абсолютно четкого, но неоптимизированного решения.

– M. A. Jackson

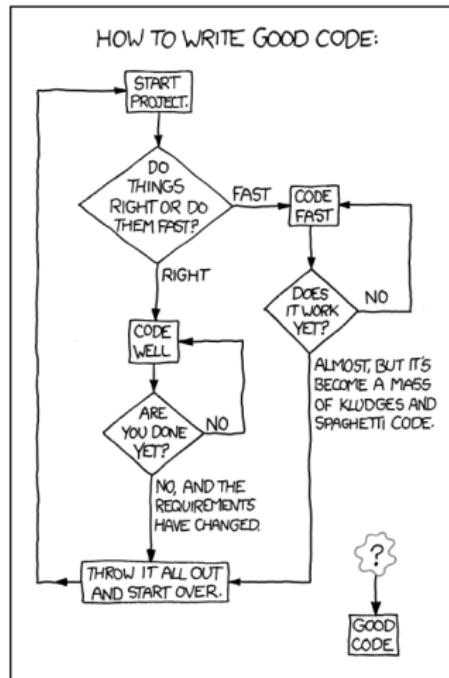
Общие рекомендации

- ▶ Fail Fast
 - ▶ Не доверяйте параметрам, переданным извне
 - ▶ assert-ы – чем больше, тем лучше
- ▶ Документируйте все открытые элементы API
 - ▶ И заодно всё остальное, для тех, кто будет это сопровождать
 - ▶ Предусловия и постусловия, исключения, потокобезопасность
- ▶ Статические проверки и статический анализ лучше, чем проверки в рантайме
 - ▶ Используйте систему типов по максимуму
- ▶ Юнит-тесты
- ▶ Continious Integration
- ▶ Не надо бояться всё переписать

Заключение



© <http://commadot.com>, Thom Holwerda



© <https://xkcd.com>

Лекция 3: Моделирование, UML

Юрий Литвинов
yurii.litvinov@gmail.com

17.09.2020г

Моделирование

- ▶ **Модель** — упрощённое подобие объекта или явления
- ▶ Нужны для изучения некоторых их свойств, абстрагируясь от сложности “настоящего” объекта или явления
- ▶ Модели используются повсеместно
 - ▶ Математические модели
 - ▶ Модели как реальные объекты
 - ▶ Модели в разработке ПО

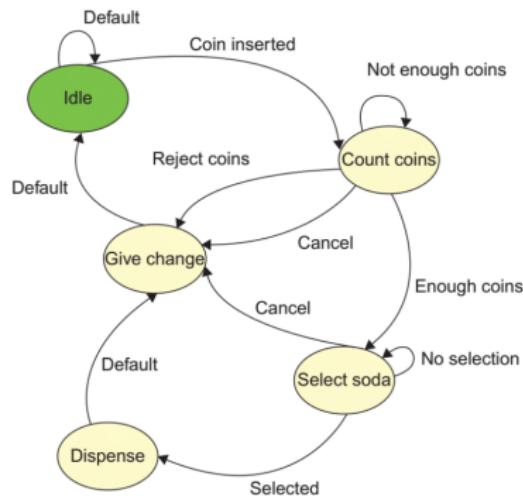
Общие свойства моделей

- ▶ Содержат меньше информации, чем реальность
- ▶ Существуют для определённой цели
- ▶ Модели субъективны, что позволяет отделить существенные свойства от несущественных
- ▶ Модели ограничены

All models are wrong, some are useful

Моделирование ПО

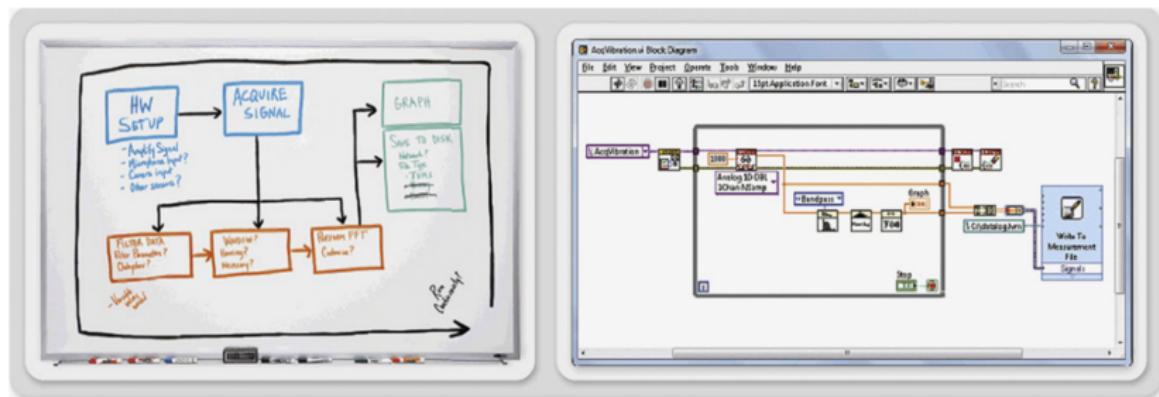
- ▶ Предназначены прежде всего для управления сложностью
- ▶ Могут моделировать как саму систему, так и окружение
- ▶ Позволяют понять, проанализировать и протестировать систему до её реализации



© N. Medvidovic

Модели бывают разные

- ▶ Используемые нотации и способы моделирования зависят от целей моделирования
 - ▶ От неформальных набросков до исполнимых моделей



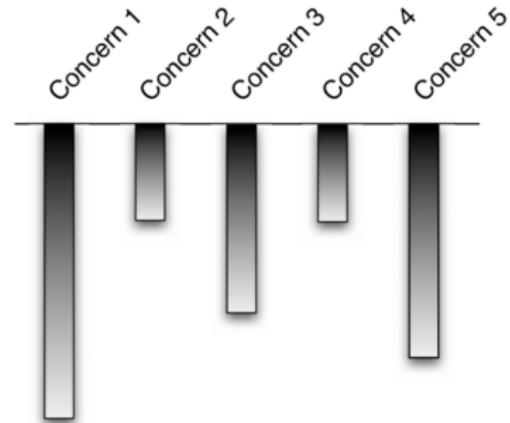
© N. Medvidovic

Архитектурные модели

- ▶ Архитектура — это набор основных решений, принятых для данной системы
- ▶ Архитектурная модель — это некоторый артефакт, который отражает некоторые или все эти решения
- ▶ Архитектурное моделирование — это процесс уточнения и документирования этих решений
- ▶ Моделирование непосредственно связано с используемой нотацией
 - ▶ Нотация архитектурного моделирования — это язык или другое средство описания архитектурных решений

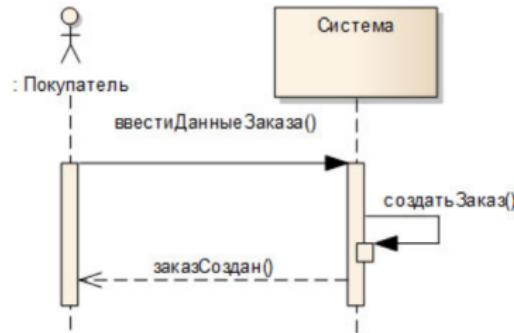
Как выбрать, что моделировать?

- ▶ При моделировании надо определиться с:
 - ▶ Какие архитектурные решения нуждаются в моделировании
 - ▶ На каком уровне детализации
 - ▶ Насколько формально
- ▶ Необходимо учитывать соотношение трудозатрат и выгоды
 - ▶ Стоимость создания *и поддержания* модели не должна быть больше преимуществ от её использования



Возможные преимущества моделей

- ▶ Инструмент, направляющий и облегчающий проектирование
- ▶ Средство коммуникации между разработчиками
- ▶ Наглядный инструмент для общения с заказчиком
- ▶ Средство документирования и фиксации принятых решений
- ▶ Исходник для генерации кода?



Виды моделей

Естественные языки

- ▶ Обычный текст — вполне себе инструмент моделирования
- ▶ Очень выразителен, не требует специальных знаний, максимально гибок
- ▶ Неоднозначен, неформален, не строг, слишком многословен, бесполезен для автоматической обработки

*"The Lunar Lander application consists of three components: a **data store** component, a **calculation** component, and a **user interface** component.*

*The job of the **data store** component is to store and allow other components access to the height, velocity, and fuel of the lander, as well as the current simulator time.*

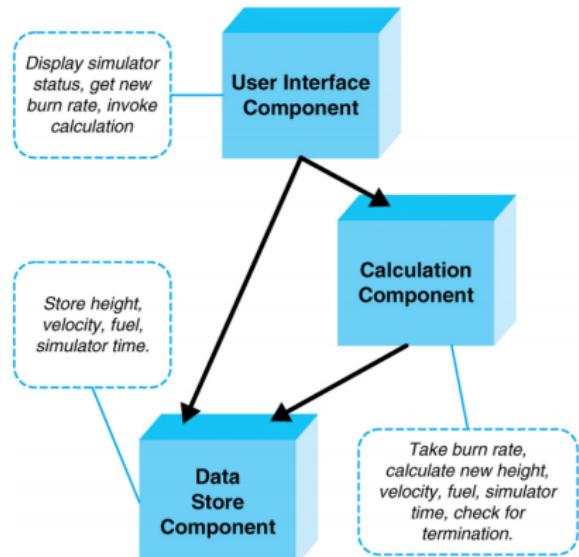
*The job of the **calculation** component is to, upon receipt of a burn-rate quantity, retrieve current values of height, velocity, and fuel from the data store component, update them with respect to the input burn-rate, and store the new values back. It also retrieves, increments, and stores back the simulator time. It is also responsible for notifying the calling component of whether the simulator has terminated, and with what state (landed safely, crashed, and so on).*

*The job of the **user interface** component is to display the current status of the lander using information from both the calculation and the data store components. While the simulator is running, it retrieves the new burn-rate value from the user, and invokes the calculation component."*

© N. Medvidovic

Неформальные графические модели

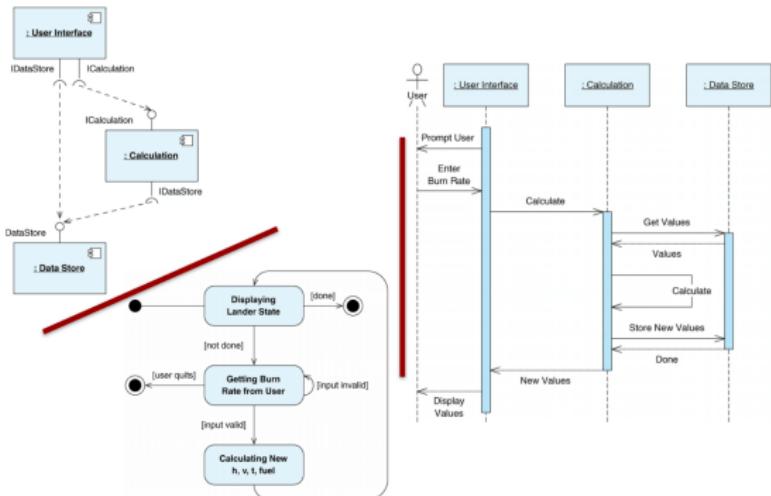
- ▶ Диаграммы, рисуемые в PowerPoint, Inkscape и подобном
- ▶ Могут быть красивыми, как правило, простые, очень гибкая нотация
- ▶ Неформальны, неоднозначны, не строги
 - ▶ Но часто воспринимаются наоборот
- ▶ Практически бесполезны для автоматической обработки



© N. Medvidovic

UML и SysML

- ▶ Несколько слабо связанных нотаций (“диаграмм”)
- ▶ Поддерживают много точек зрения, общеприняты, широкая поддержка инструментами
- ▶ Нет строгой семантики, сложно обеспечить консистентность, сложно расширять



© N. Medvidovic

AADL и другие текстовые формальные языки

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

```

data lander_state_data;
end lander_state_data;
bus lan_bus_type;
end lan_bus_type;

bus implementation lan_bus_type.ethernet
properties
    Transmission_Time => 1 ms .. 5 ms;
    Allowed_Message_Size => 1 b .. 1 kb;
end lan_bus_type.ethernet;
system calculation_type
features
    network : requires bus access
        lan_bus.calculation_to_datastore;
    request_get   : out event port;
    response_get  : in event data port lander_state_data;
    request_store : out event port lander_state_data;
    response_store : in event port;
end calculation_type;

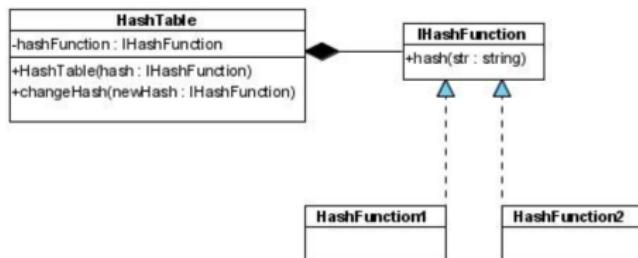
system implementation calculation_type.calculation
subcomponents
    the_calculation_processor :
        processor calculation_processor_type;
    the_calculation_process : process
        calculation_process_type.one_thread;

```

© N. Medvidovic

Вернёмся к визуальным моделям

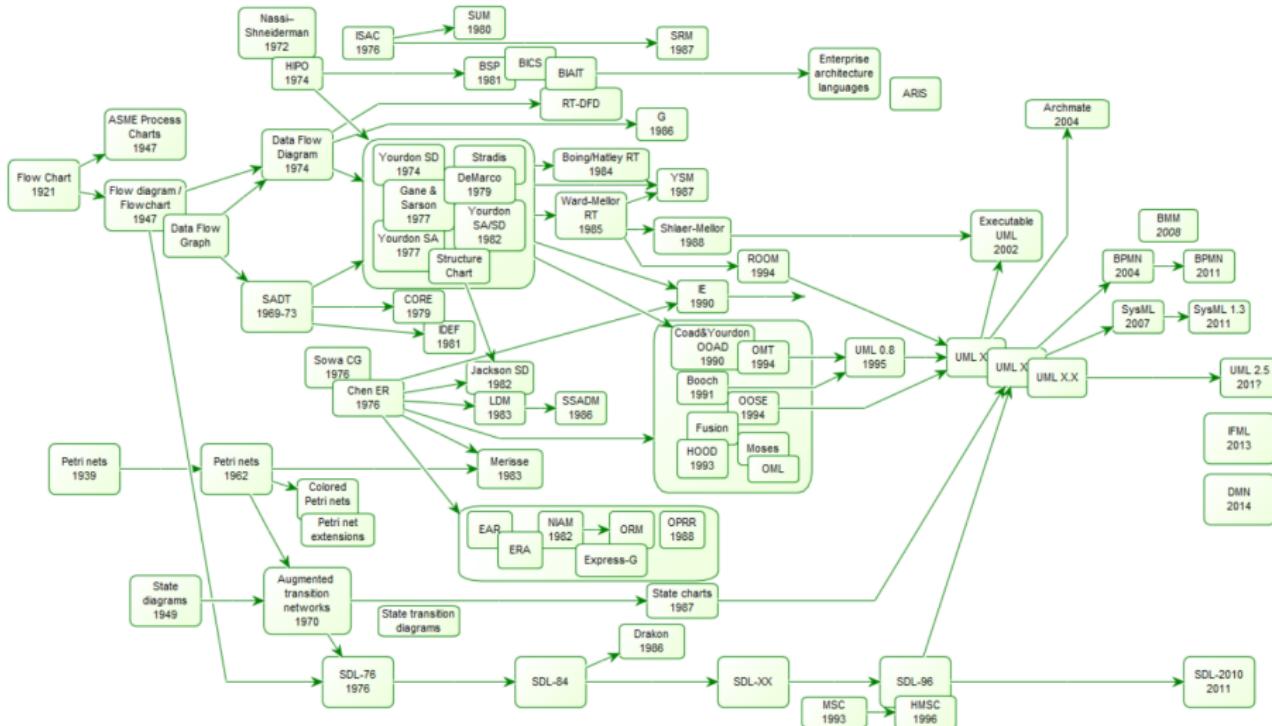
- ▶ **Метафора визуализации** — договорённость о том, как будут представляться сущности языка
- ▶ **Точка зрения моделирования** — какой аспект системы и для кого моделируется
- ▶ Бывают одноразовые модели, документация и графические исходники
 - ▶ **Семантический разрыв** — неспособность модели полностью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



Unified Modeling Language

- ▶ Семейство графических нотаций
 - ▶ 14 видов диаграмм
- ▶ Общая метамодель
- ▶ Стандарт под управлением Object Management Group
 - ▶ UML 1.1 — 1997 год
 - ▶ UML 2.0 — 2005 год
 - ▶ UML 2.5.1 — декабрь 2017 года
- ▶ Прежде всего, для проектирования ПО
 - ▶ После UML 2.0 стали появляться нотации и для инженеров
- ▶ Расширяем
 - ▶ Профили — механизм легковесного расширения
 - ▶ Метамоделирование

История



Виды диаграмм

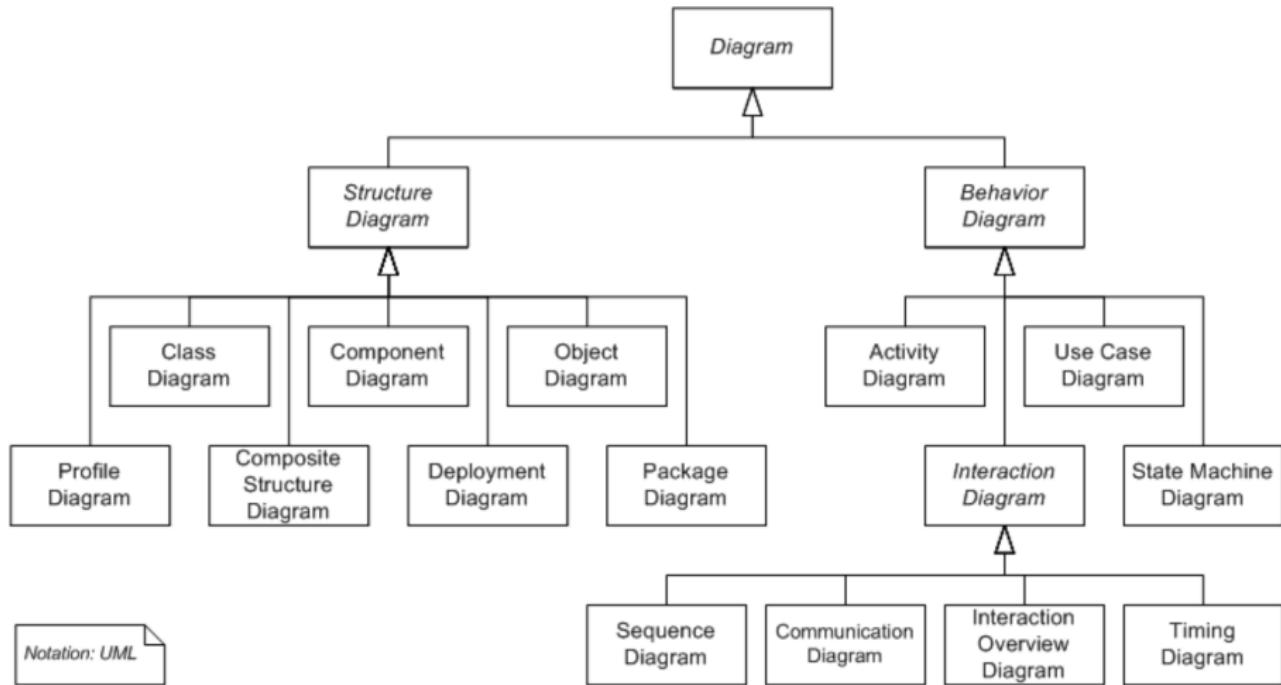
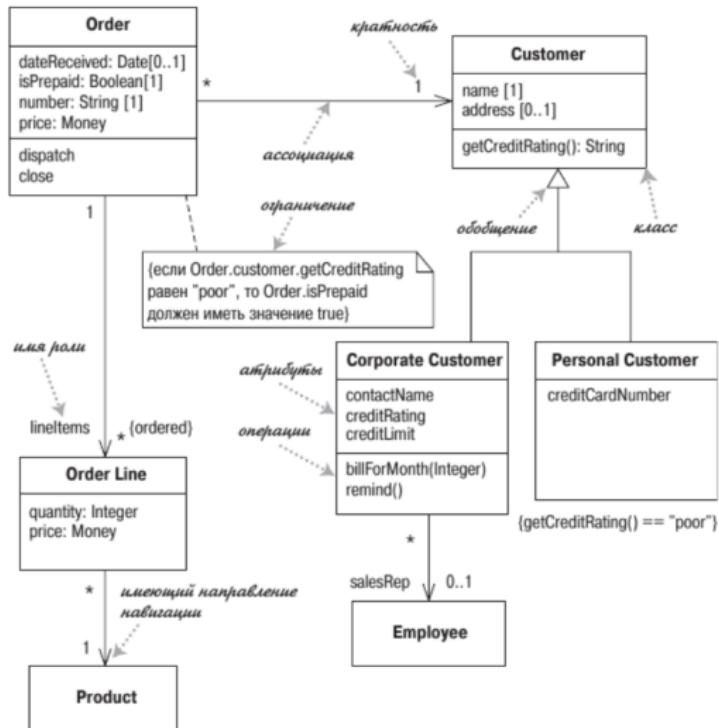


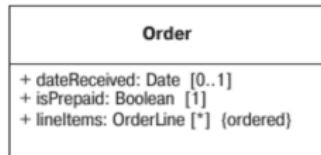
Диаграмма классов



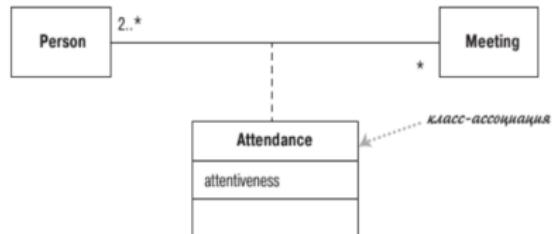
© М. Фаулер. “UML. Основы”

Свойства

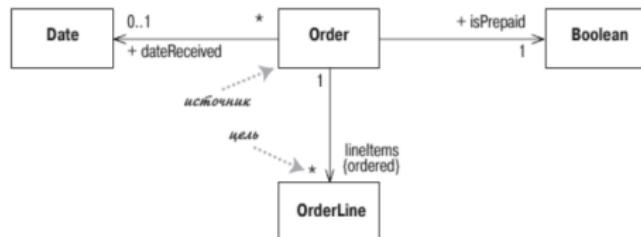
Атрибуты:



Ассоциация-класс:



Ассоциации:



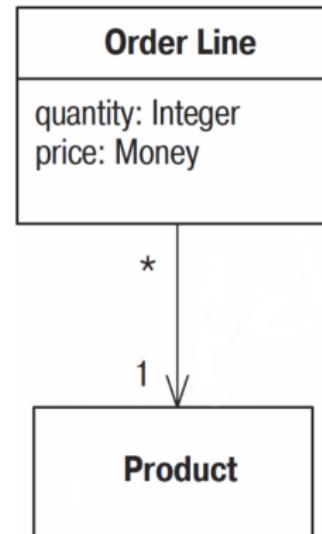
© М. Фаулер. “UML. Основы”

Синтаксис свойств

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

Как это связано с кодом

```
public class OrderLine {
    private int quantity;
    private Product product;
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    public Money getPrice() {
        return product.getPrice().multiply(quantity);
    }
}
```



Двунаправленные ассоциации



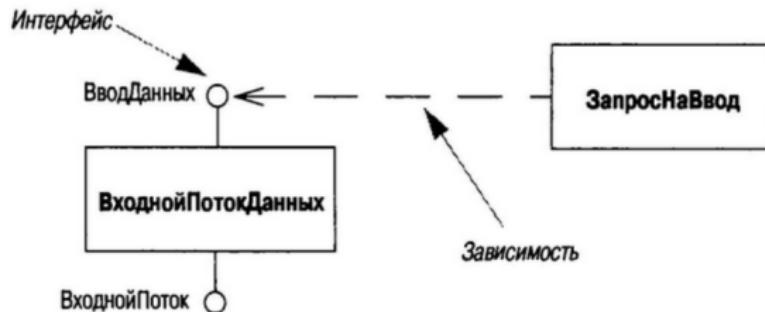
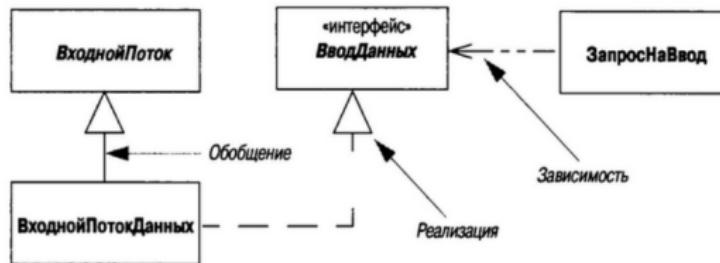
```

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;
    }
}
  
```

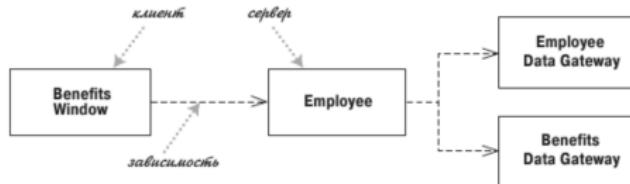
Интерфейсы



© М. Фаулер. "UML. Основы"

Зависимости

- ▶ call
- ▶ create
- ▶ instantiate
- ▶ derive
- ▶ realize
- ▶ responsibility
- ▶ refine
- ▶ trace



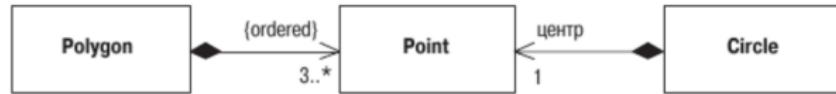
© М. Фаулер. “UML. Основы”

Агрегация и композиция

Агрегация:

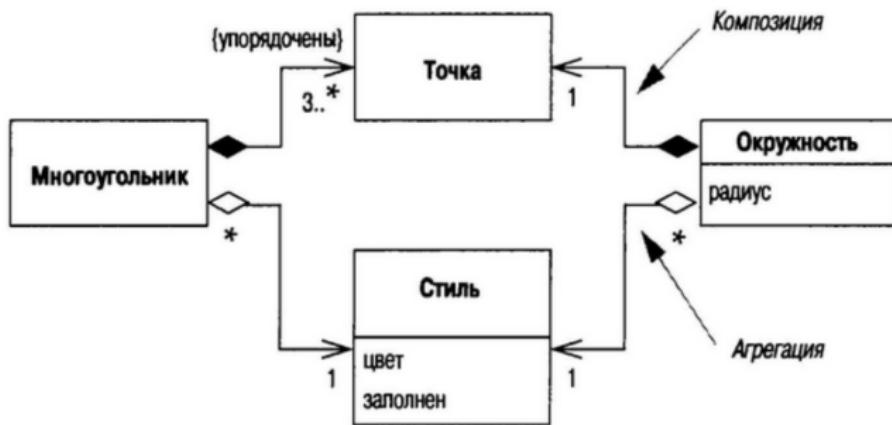


Композиция:



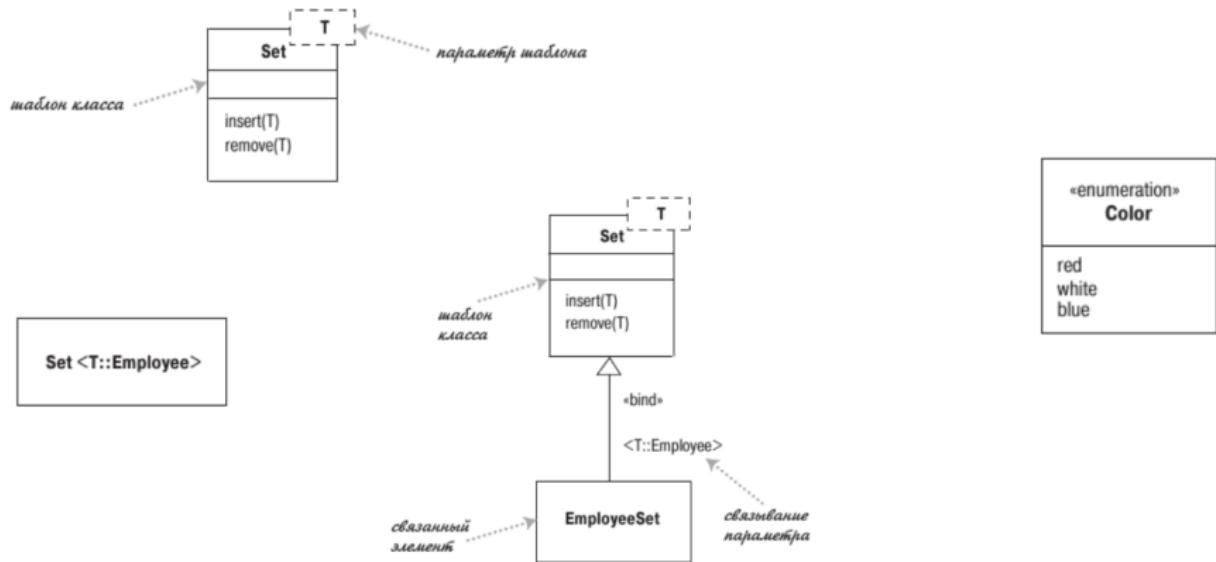
© М. Фаулер. "UML. Основы"

Агрегация и композиция, пример



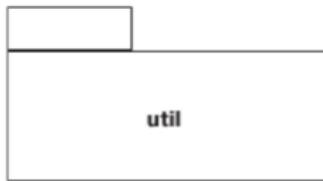
© М. Фаулер. “UML. Основы”

Шаблоны и перечисления



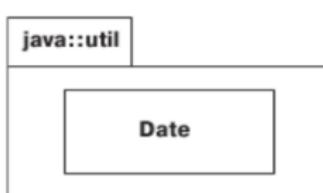
© М. Фаулер. “UML. Основы”

Диаграммы пакетов

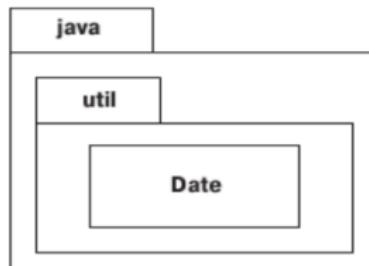


Содержимое, перечисленное в прямоугольнике

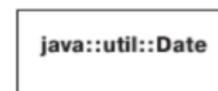
Содержимое в виде диаграммы в прямоугольнике



Полностью определенное имя пакета



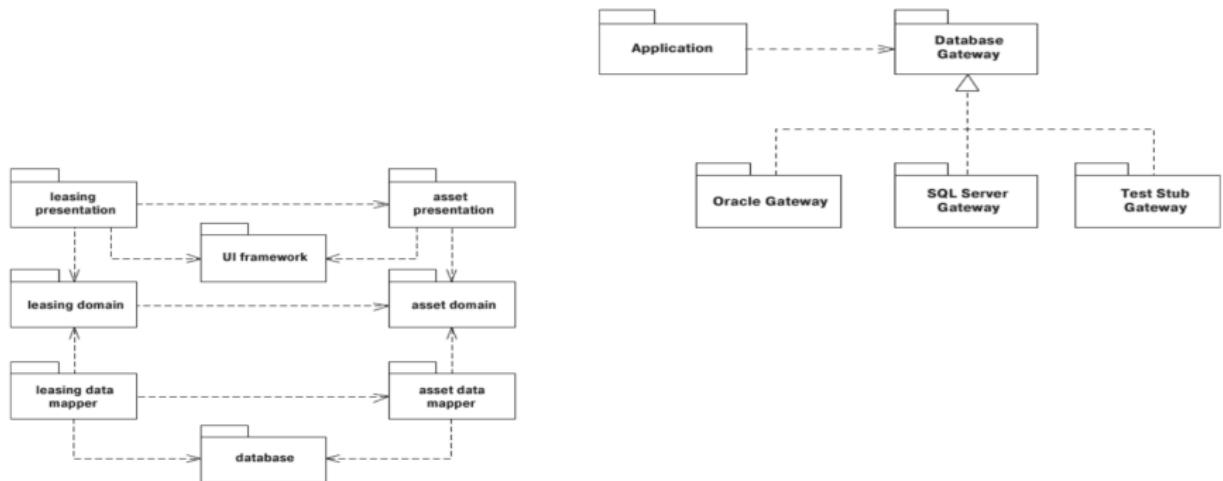
Вложенные пакеты



Полностью определенное имя класса

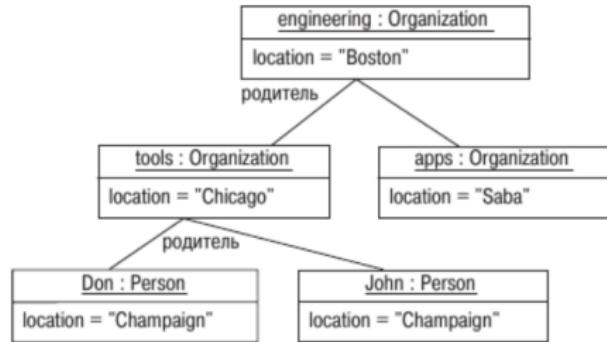
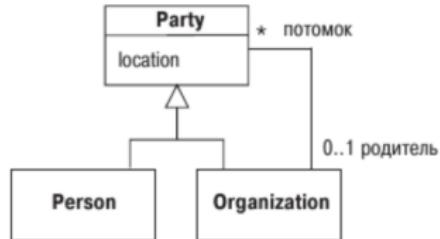
© М. Фаулер. “UML. Основы”

Диаграммы пакетов, зависимости



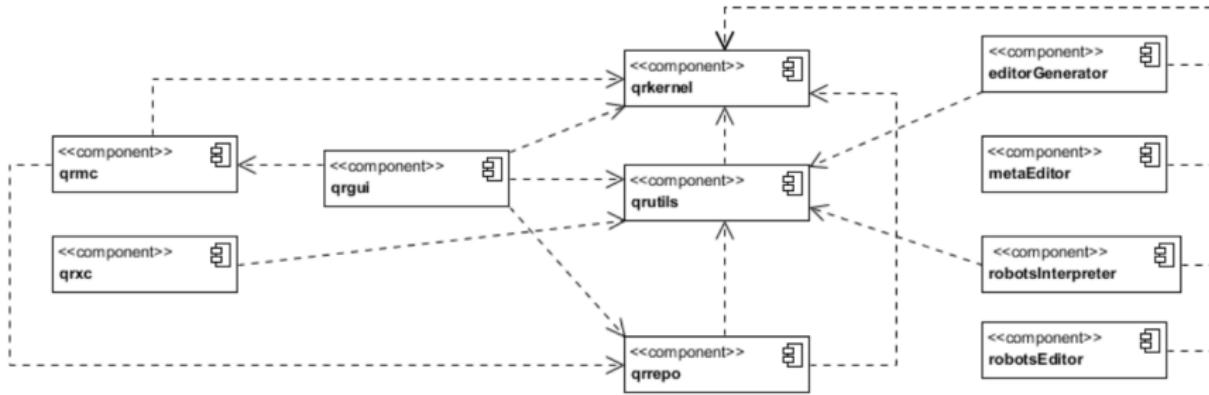
© М. Фаулер. “UML. Основы”

Диаграммы объектов

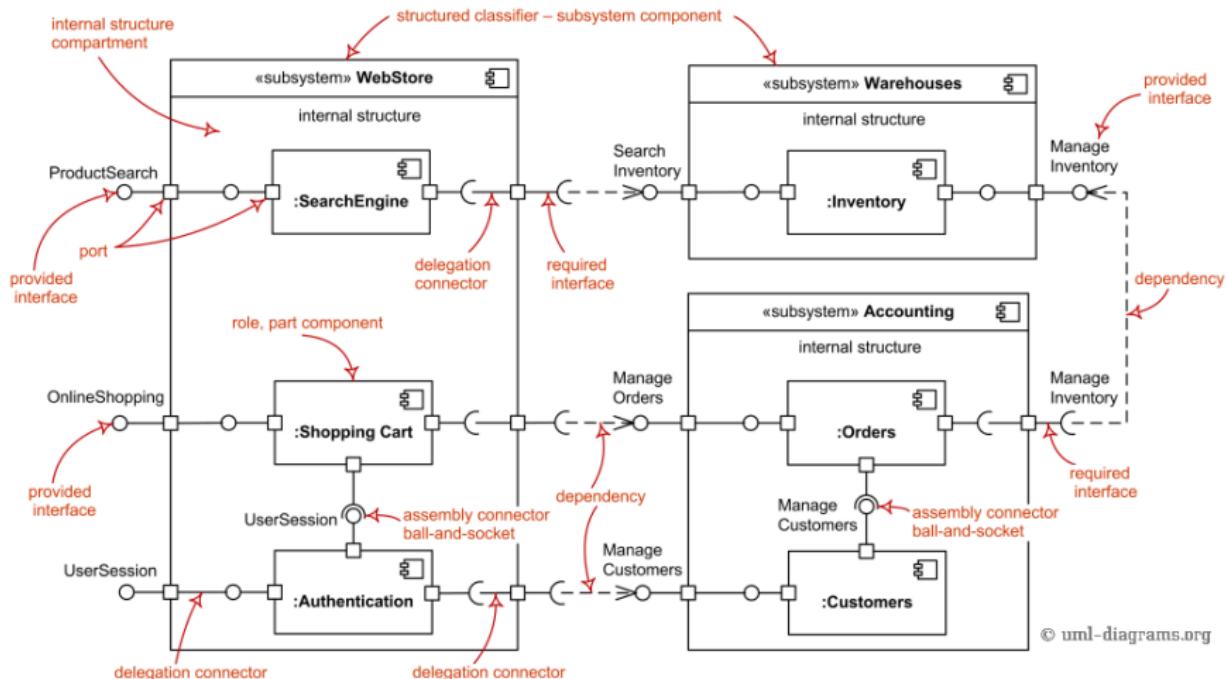


© М. Фаулер. “UML. Основы”

Диаграммы компонентов



Более подробно



© uml-diagrams.org

© <http://www.uml-diagrams.org>

Лекция 4: Моделирование и анализ

Юрий Литвинов
yurii.litvinov@gmail.com

24.09.2020г

Computer-Aided Software Engineering

- ▶ В 80-е годы термином CASE называли всё, что помогает разрабатывать ПО с помощью компьютера
 - ▶ Даже текстовые редакторы
- ▶ Теперь — прежде всего средства для визуального моделирования (UML-диаграммы, ER-диаграммы и т.д.)
- ▶ Отличаются от графических редакторов тем, что “понимают”, что в них рисуют
- ▶ Нынче чаще используются термины “MDE tool”, “UML tool” и т.д.

Типичная функциональность CASE-инструментов

- ▶ Набор визуальных редакторов
- ▶ Репозиторий
- ▶ Набор генераторов
- ▶ Текстовый редактор
- ▶ Редактор форм
- ▶ Средства обратного проектирования (reverse engineering)
- ▶ Средства верификации и анализа моделей
- ▶ Средства эмуляции и отладки
- ▶ Средства обеспечения командной разработки
- ▶ API для интеграции с другими инструментами
- ▶ Библиотеки шаблонов и примеров

Примеры CASE-инструментов

- ▶ “Рисовалки”
 - ▶ Visio
 - ▶ Dia
 - ▶ SmartDraw
 - ▶ LucidChart
 - ▶ Creately
- ▶ Полноценные CASE-системы
 - ▶ Enterprise Architect
 - ▶ Rational Software Architect
 - ▶ MagicDraw
 - ▶ Visual Paradigm
 - ▶ GenMyModel
- ▶ Забавные штуки
 - ▶ <https://www.websequencediagrams.com/>
 - ▶ <http://yuml.me/>
 - ▶ <http://plantuml.com/>

Моделирование требований

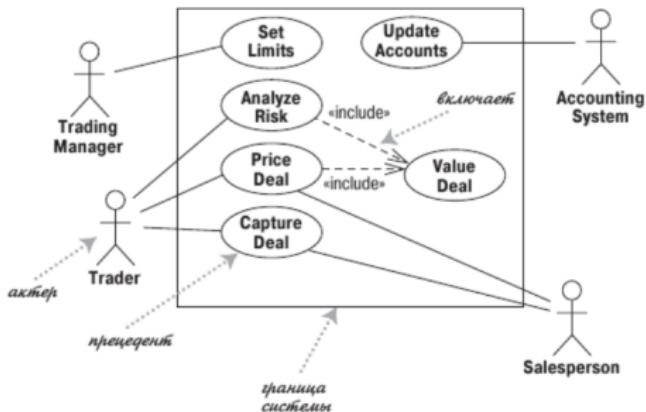
Первый этап разработки любой системы — сбор и анализ требований

- ▶ Понимание разработчиками решаемой задачи
- ▶ Соглашение между разработчиками, заказчиками и пользователями
 - ▶ Заказчики и пользователи часто разные люди с разными потребностями
- ▶ Чёткое обозначение границ системы
- ▶ Основа для планирования проекта
- ▶ Чаще всего словесное описание требований, реже формальные модели

Диаграмма случаев использования UML

Диаграмма прецедентов

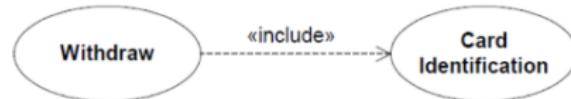
- ▶ Ивар Якобсон, 1992 год
- ▶ Акторы (или актёры, роли) — внешние сущности, использующие систему
 - ▶ Люди или другие программные системы
- ▶ Случаи использования (прецеденты) — цель использования системы актором
 - ▶ Раскрываются в набор сценариев, описываемых чаще текстом



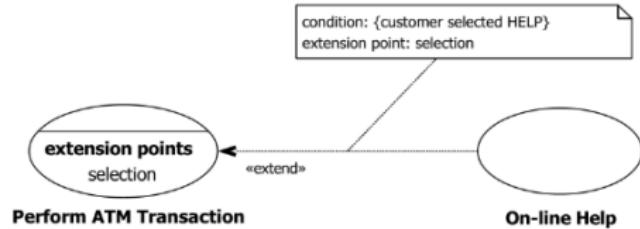
© М. Фаулер, UML. Основы

Include и Extend

Include:

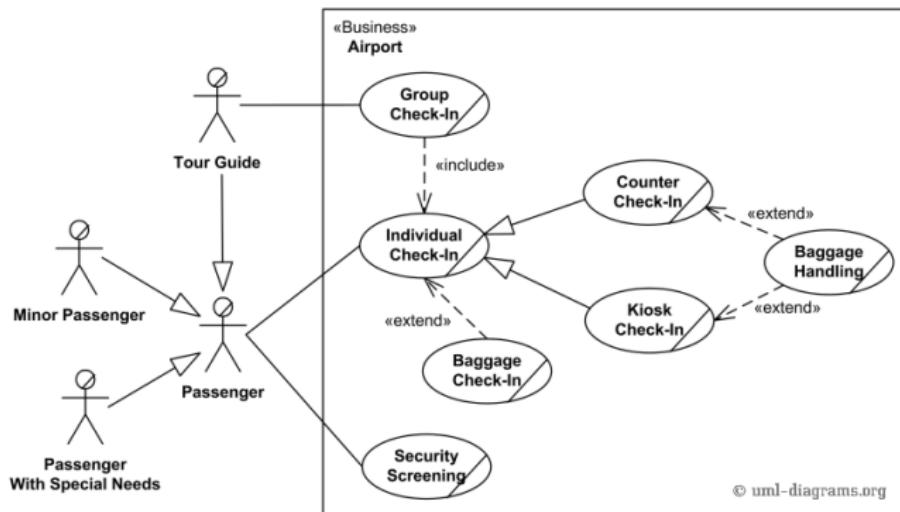


Extend:



© OMG, UML 2.5 Specification

Пример, check-in в аэропорту



© <http://www.uml-diagrams.org>

Сценарий использования, типичная структура

- ▶ Заголовок (цель основного актора)
- ▶ Заинтересованные лица, акторы, основной актор
- ▶ Предусловия
- ▶ Триггеры (активаторы)
- ▶ Основной порядок событий
- ▶ Альтернативные пути и расширения
- ▶ Постусловия

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:		
<ol style="list-style-type: none"> 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:		
<ol style="list-style-type: none"> 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 <ol style="list-style-type: none"> 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:		
<ol style="list-style-type: none"> 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) <ol style="list-style-type: none"> 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 		

Контекстная диаграмма IDEF0

- ▶ Обозначает границы системы и способы её взаимодействия с внешним миром
- ▶ Используется для моделирования не только ПО
- ▶ Каждая сторона имеет свой смысл
 - ▶ Слева — входные данные или материалы
 - ▶ Сверху — управление
 - ▶ Снизу — механизмы
 - ▶ Справа — выходные данные или продукты



© <http://ecm-journal.ru>

Диаграмма характеристик

Feature Diagram

- ▶ Представляет функциональность системы в виде дерева
- ▶ Используется в основном для моделирования семейств программных продуктов (Product lines)

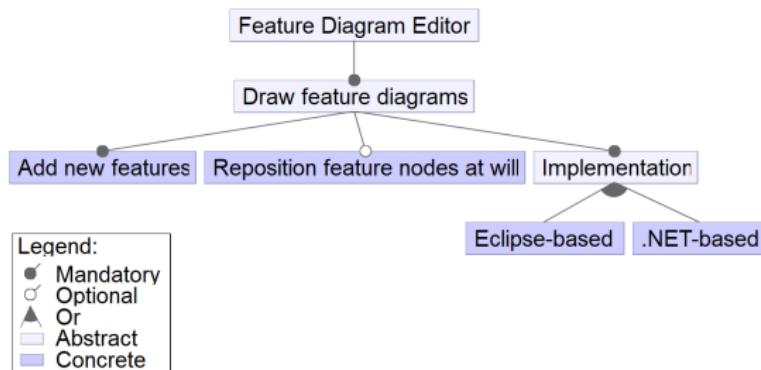
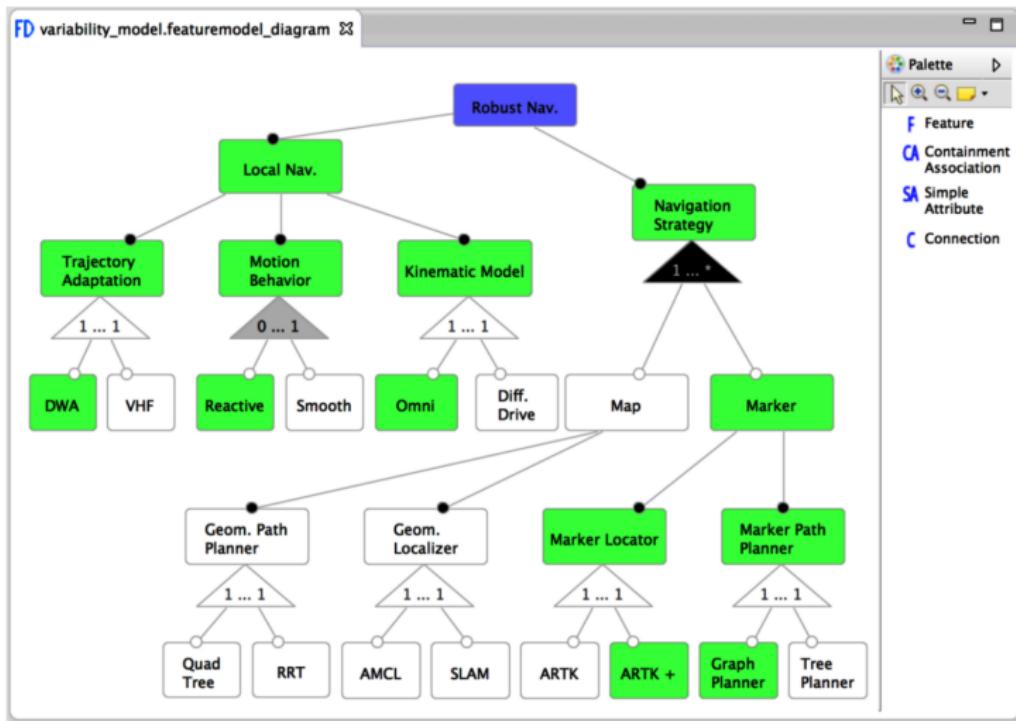


Диаграмма характеристик, пример



© D. Brugali

Feature Tree

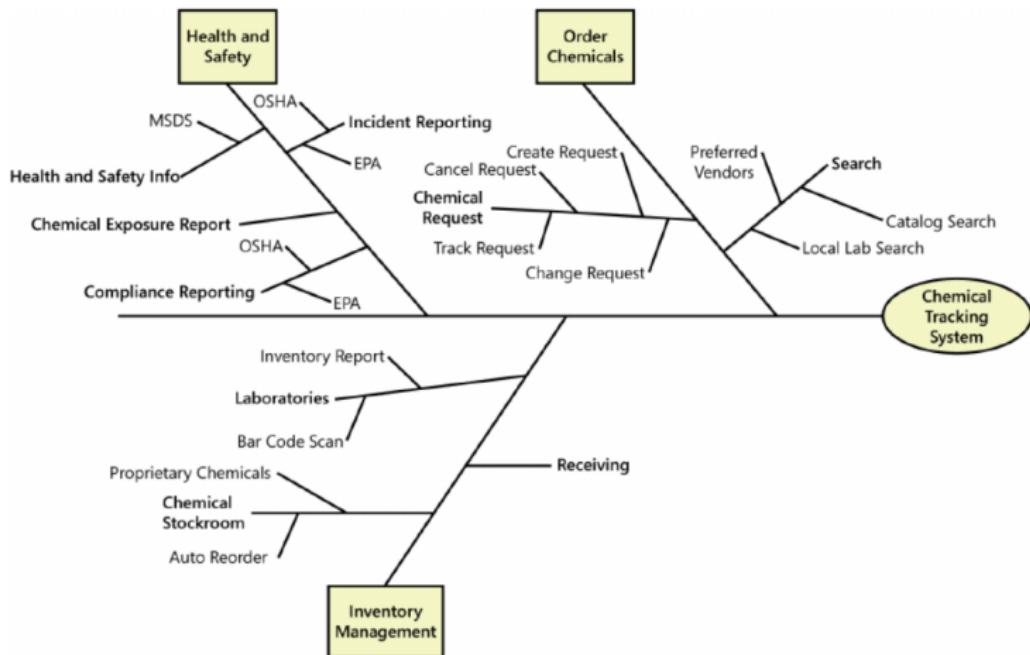
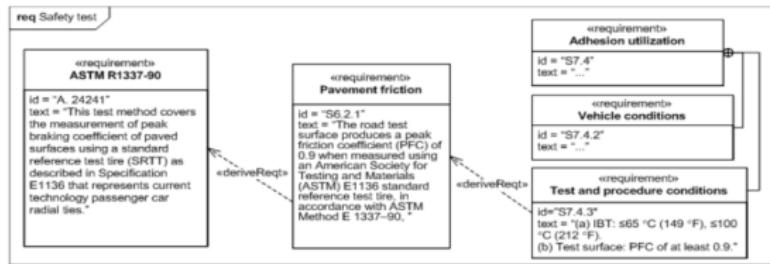


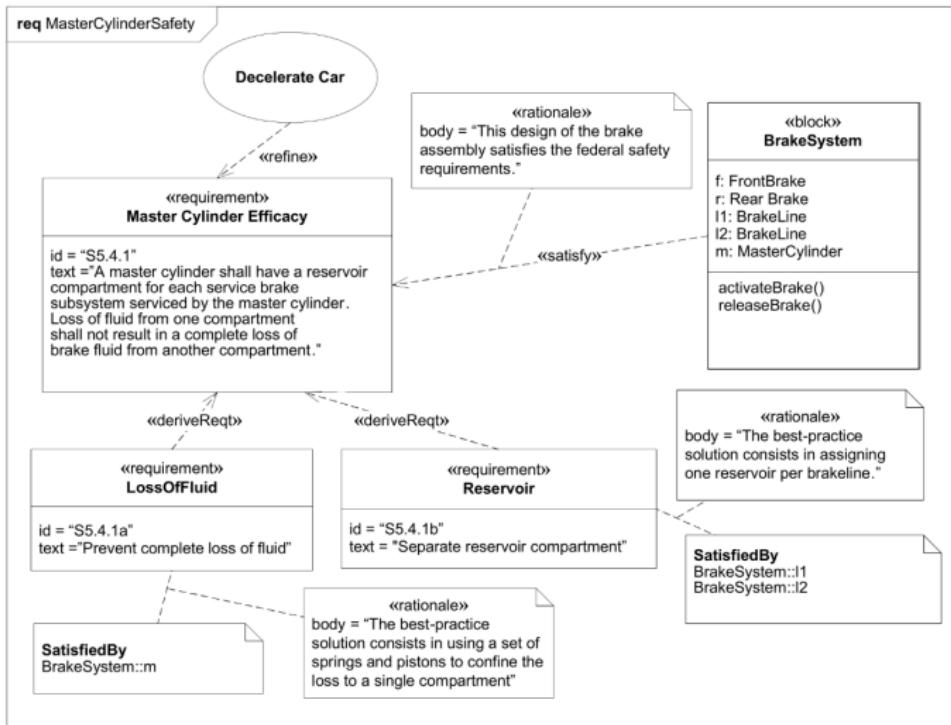
Диаграмма требований, SysML

- Более формальная нотация дерева фич



© OMG SysML 1.4 Specification

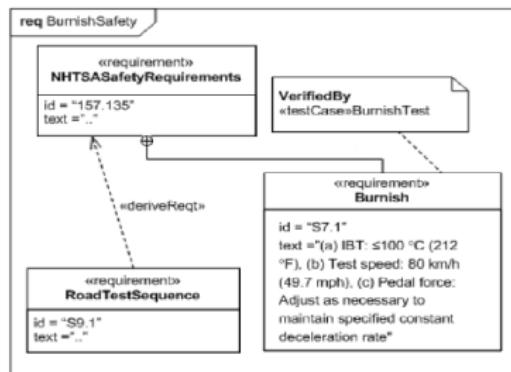
Диаграмма требований SysML, пример



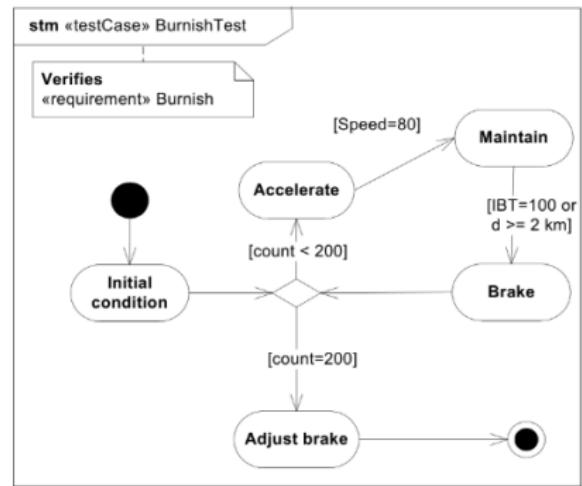
© OMG SysML 1.4 Specification

Диаграмма требований SysML и тесты

Требования:



Сценарий тестирования:

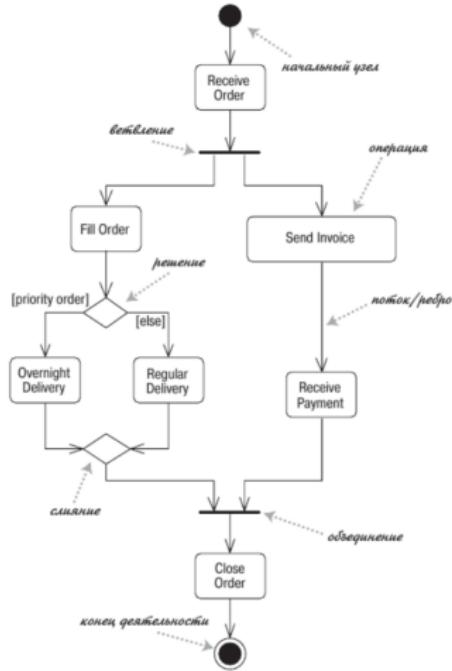


© OMG SysML 1.4 Specification

Диаграмма активностей UML

Диаграммы деятельности

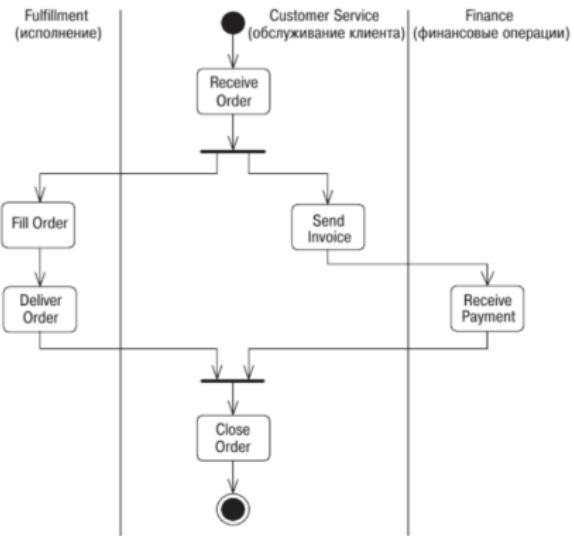
- ▶ Используются для моделирования бизнес-процессов, тоже на первых этапах
 - ▶ Может быть визуализацией сценария использования
- ▶ Иногда — для моделирования алгоритма
- ▶ Расширенные блок-схемы
- ▶ Семантика на основе сетей Петри



© М. Фаулер, UML. Основы

Диаграмма активностей, разделы

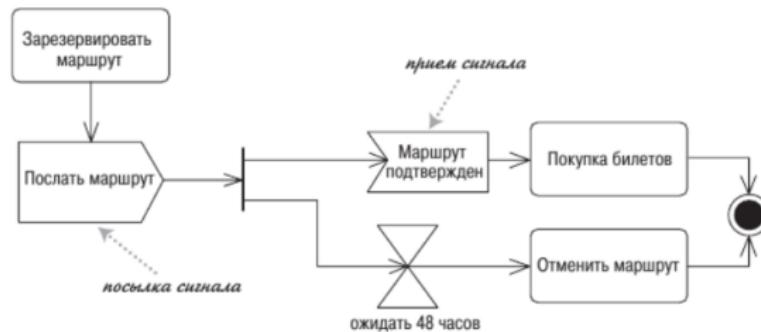
- ▶ Раздел представляет отдел организации (или организацию), отвечающий за часть работы
- ▶ Визуализирует поток работ между отделами



© М. Фаулер, UML. Основы

Диаграмма активностей, сигналы

- ▶ Для визуализации асинхронных процессов
- ▶ Сигналом может быть посылка документа, запрос и т.д.

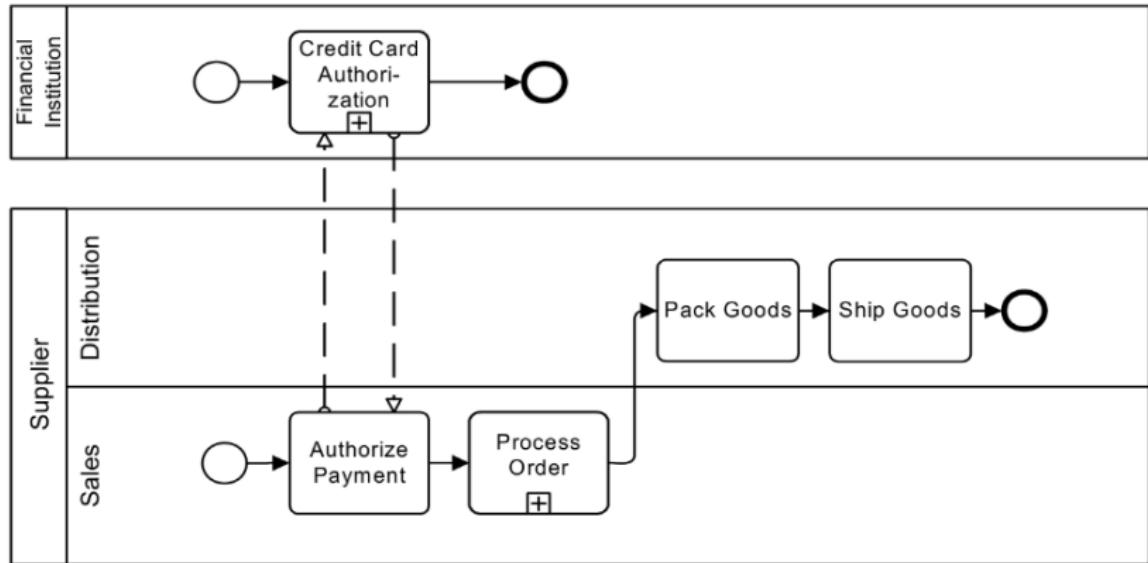


© М. Фаулер, UML. Основы

Business Process Model and Notation

- ▶ Версия 1.0 в 2004 году, текущая (2.0) — в 2011
- ▶ Для описания бизнес-процессов
 - ▶ Сильно продвинутые диаграммы активностей
 - ▶ Позволяют описывать группы взаимодействующих процессов
 - ▶ Исполнимая семантика
 - ▶ Правила генерации в BPEL
 - ▶ Business Process Execution Language

Пример диаграммы



© OMG BPMN 2.0 Specification

События

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

© <https://ru.wikipedia.org/wiki/BPMN>

Операторы ветвления



Оператор исключающего ИЛИ, управляемый данными



Оператор исключающего ИЛИ, управляемый событиями



Оператор включающего ИЛИ



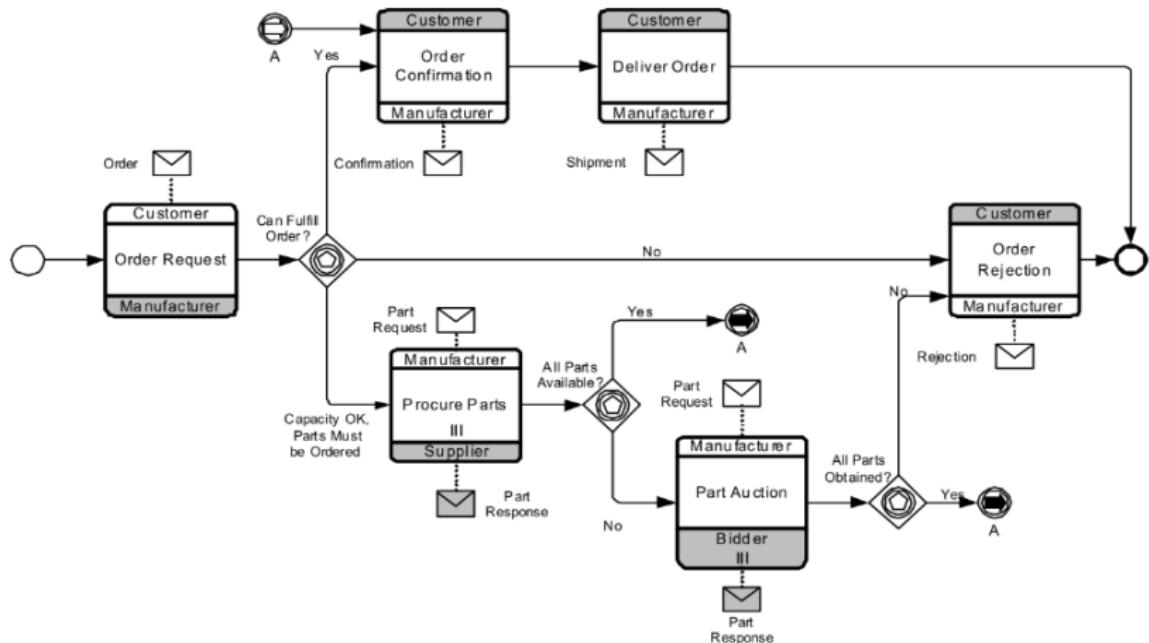
Оператор И



Сложный оператор

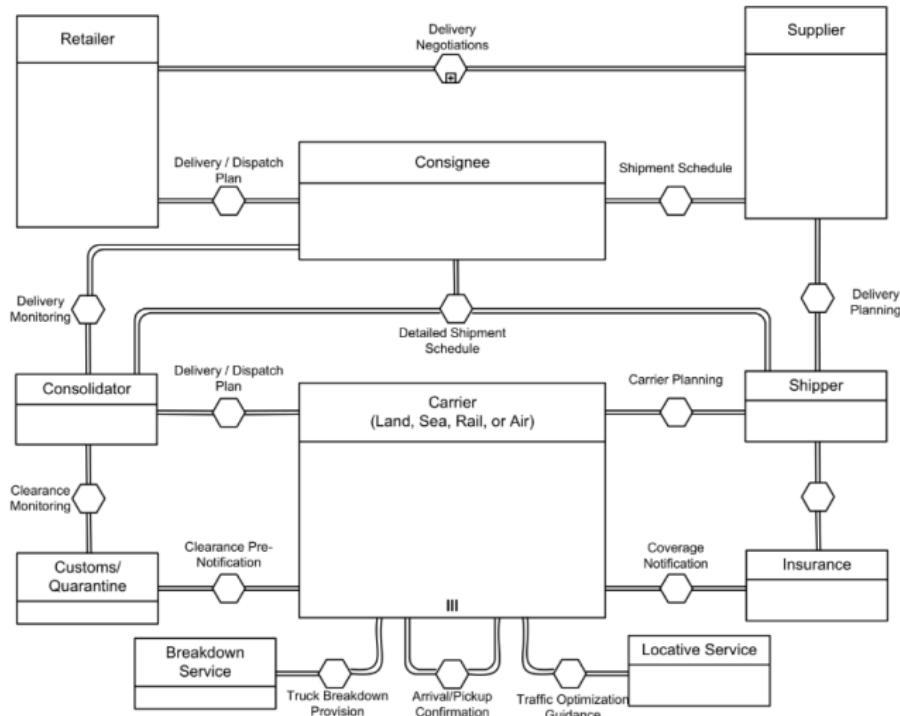
© <https://ru.wikipedia.org/wiki/BPMN>

Диаграмма хореографии



© OMG BPMN 2.0 Specification

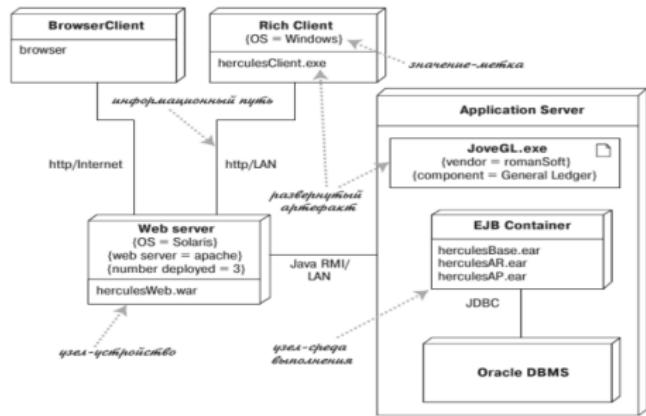
Диаграмма диалогов



© OMG BPMN 2.0 Specification

Диаграмма развёртывания UML

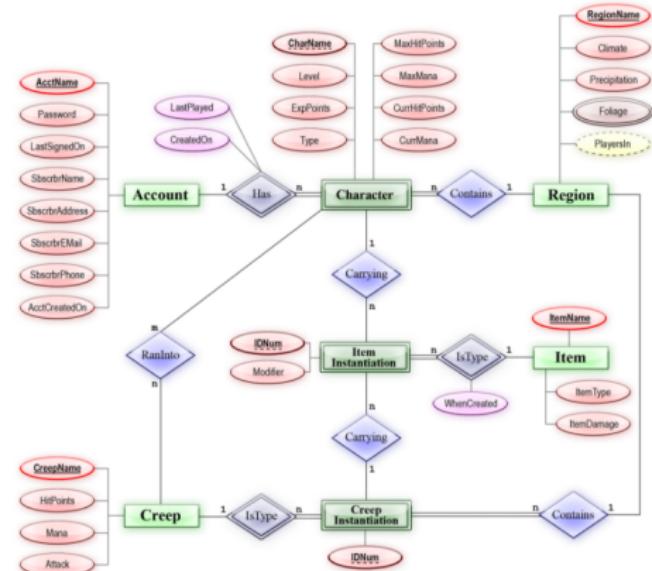
- ▶ Показывает отображение компонентов и физических артефактов на реальные (или виртуальные) устройства
- ▶ Бывает полезна на начальных этапах проектирования, даже до диаграмм компонентов



© М. Фаулер, UML. Основы

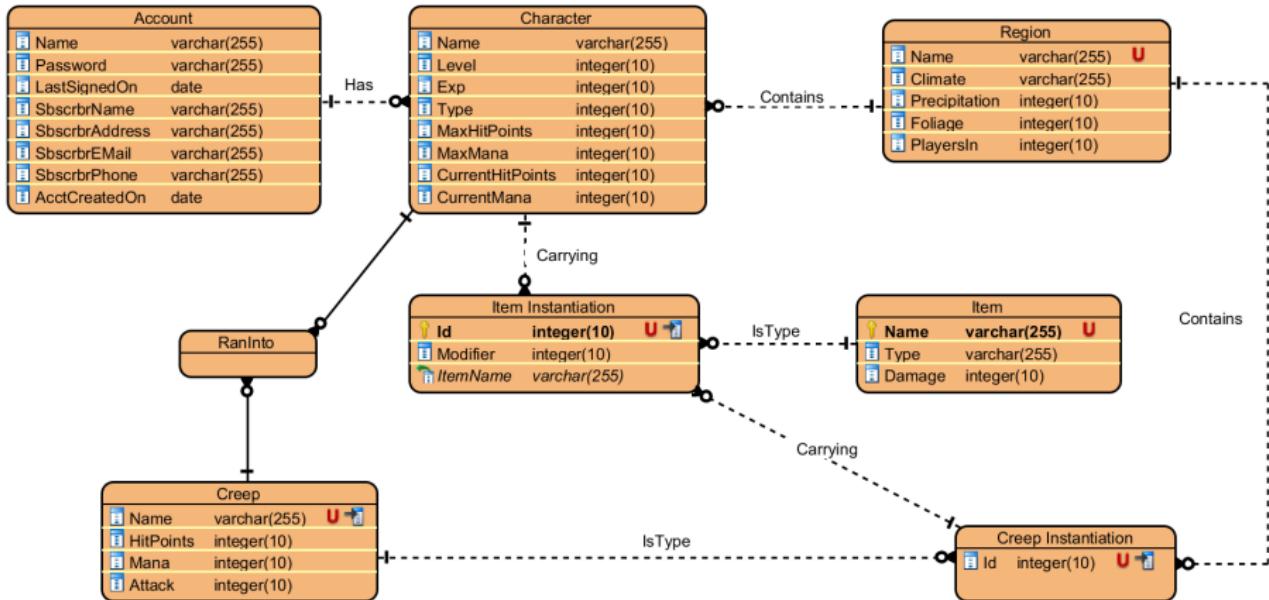
Диаграммы “Сущность-связь”

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

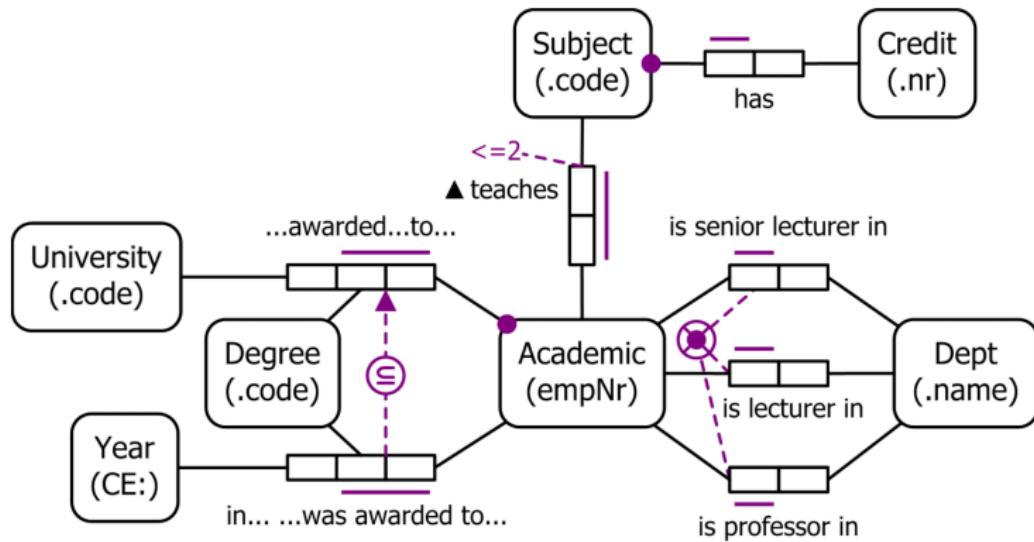


© <https://ru.wikipedia.org>

Нотация “Вороньей лапки”



Object-Role Modeling

© <http://www.orm.net>

Лекция 5: Моделирование поведения

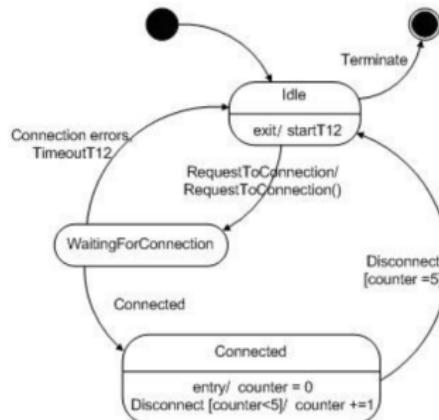
Юрий Литвинов
yurii.litvinov@gmail.com

01.10.2020г

Диаграммы конечных автоматов

Диаграммы состояний

- ▶ Состояния объекта как часть жизненного цикла
- ▶ Моделирование реактивных объектов
 - ▶ Например, сетевое соединение
 - ▶ Или знакомый пример с торговым автоматом
- ▶ Имеют исполнимую семантику
- ▶ Д. Харел, 1987



Диаграммы конечных автоматов, синтаксис

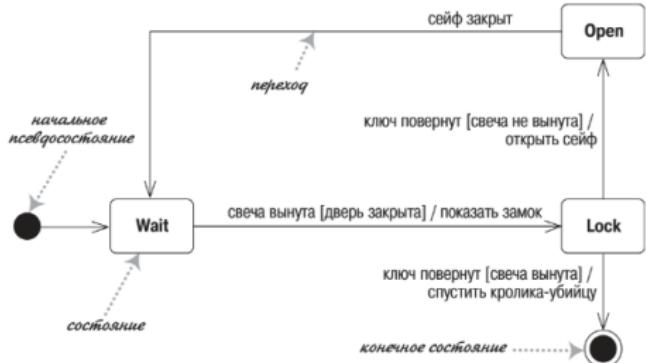
▶ Состояние

- ▶ entry activity
- ▶ exit activity
- ▶ do activity
- ▶ внутренний переход

▶ Событие

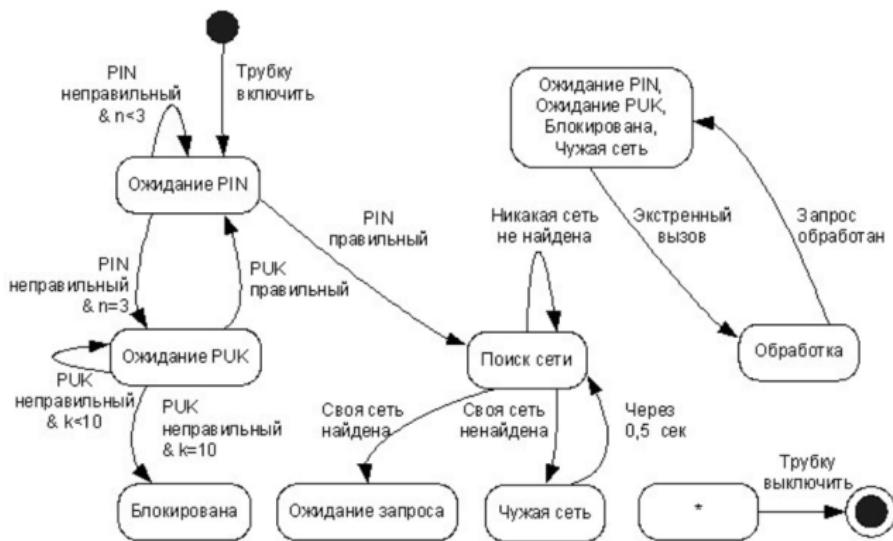
▶ Переход

- ▶ [<trigger> ',' <trigger>]* '['<guard>']' [/<behavior-expression>]]



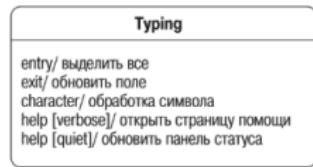
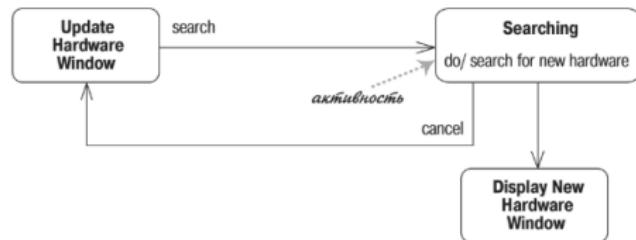
© М. Фаулер, UML. Основы

Пример, мобильный телефон

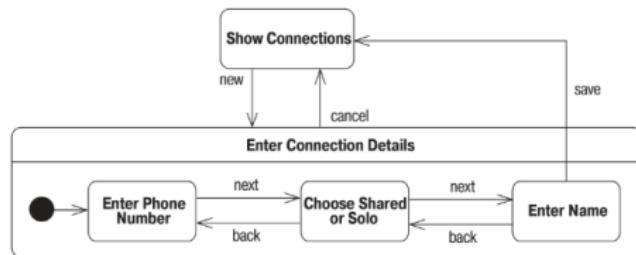


Диаграммы конечных автоматов, прочие вещи

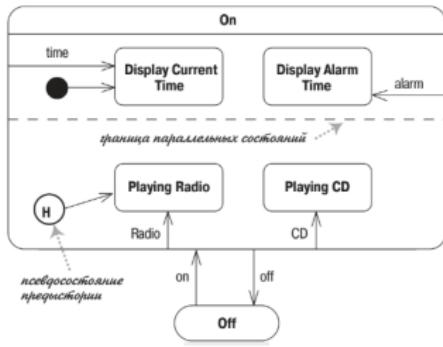
Активности:



Вложенные состояния:



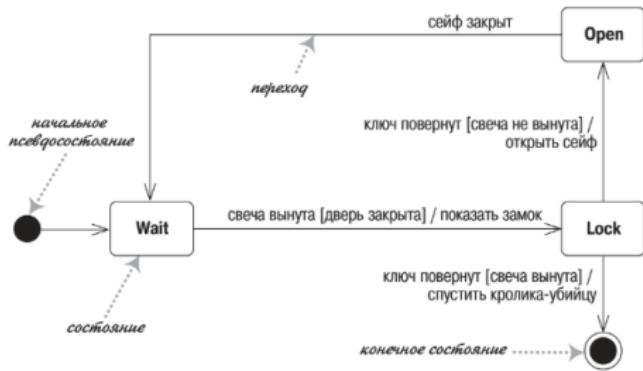
Параллельные состояния, псевдосостояние истории:



© М. Фаулер, UML. Основы

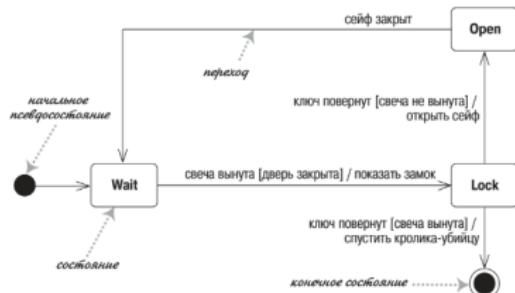
Генерация кода

```
public void handleEvent(PanelEvent anEvent) {
    switch (currentState) {
        case PanelState.Open:
            switch (anEvent) {
                case PanelEvent.SafeClosed:
                    currentState = PanelState.Wait;
            }
            break;
        case PanelState.Wait:
            switch (anEvent) {
                case PanelEvent.CandleRemoved:
                    if (isDoorOpen) {
                        revealLock();
                        currentState = PanelState.Lock;
                    }
            }
            break;
        case PanelState.Lock:
            switch (anEvent) {
                case PanelEvent.KeyTurned:
                    if (isCandleIn) {
                        openSafe();
                        currentState = PanelState.Open;
                    } else {
                        releaseKillerRabbit();
                        currentState = PanelState.Final;
                    }
            }
            break;
    }
}
```



© М. Фаулер, UML. Основы

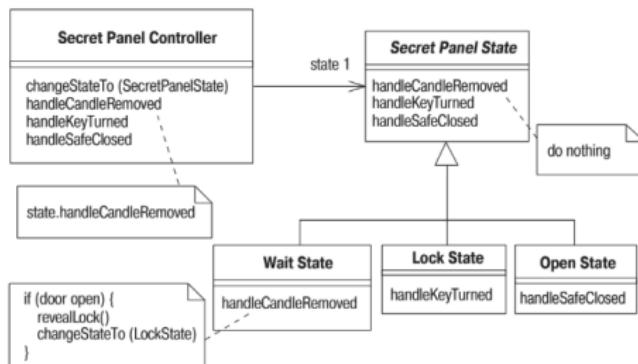
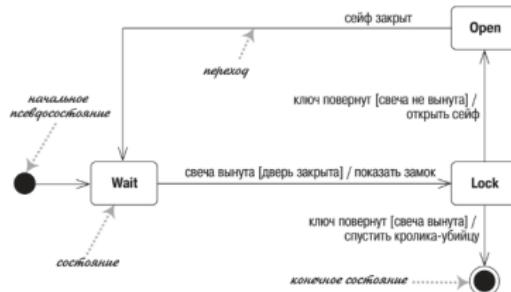
Таблица состояний



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

© М. Фаулер, UML. Основы

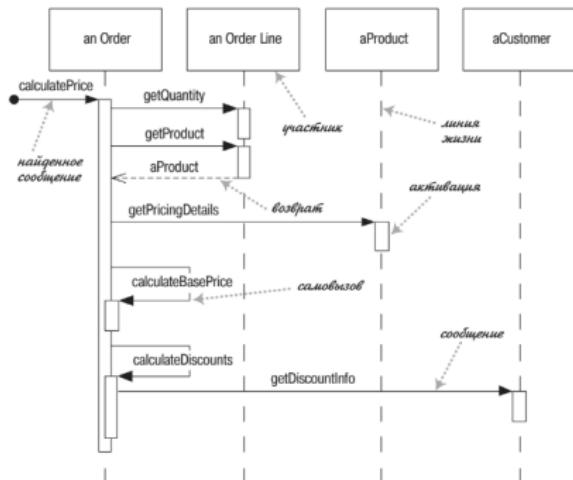
Паттерн “Состояние”



© М. Фаулер, UML. Основы

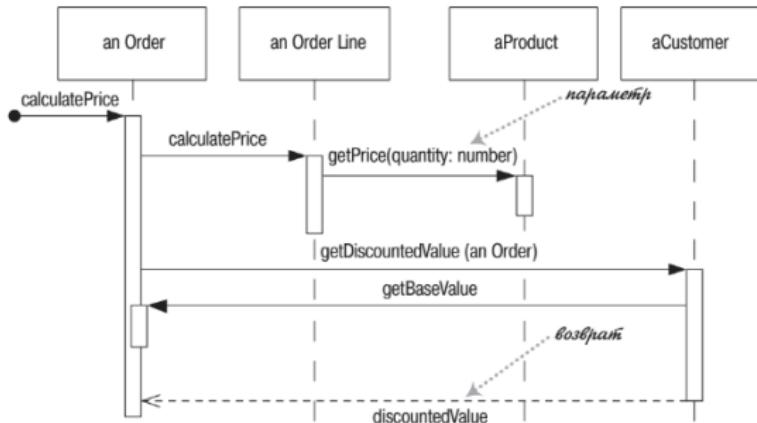
Диаграммы последовательностей

- ▶ Применяются для визуализации взаимодействия между объектами
 - ▶ Особо удобно для асинхронных вызовов
 - ▶ Телекоммуникационные протоколы
- ▶ Могут применяться на этапе анализа предметной области
- ▶ Могут применяться для составления плана тестирования
- ▶ И даже для визуализации логов работающей системы



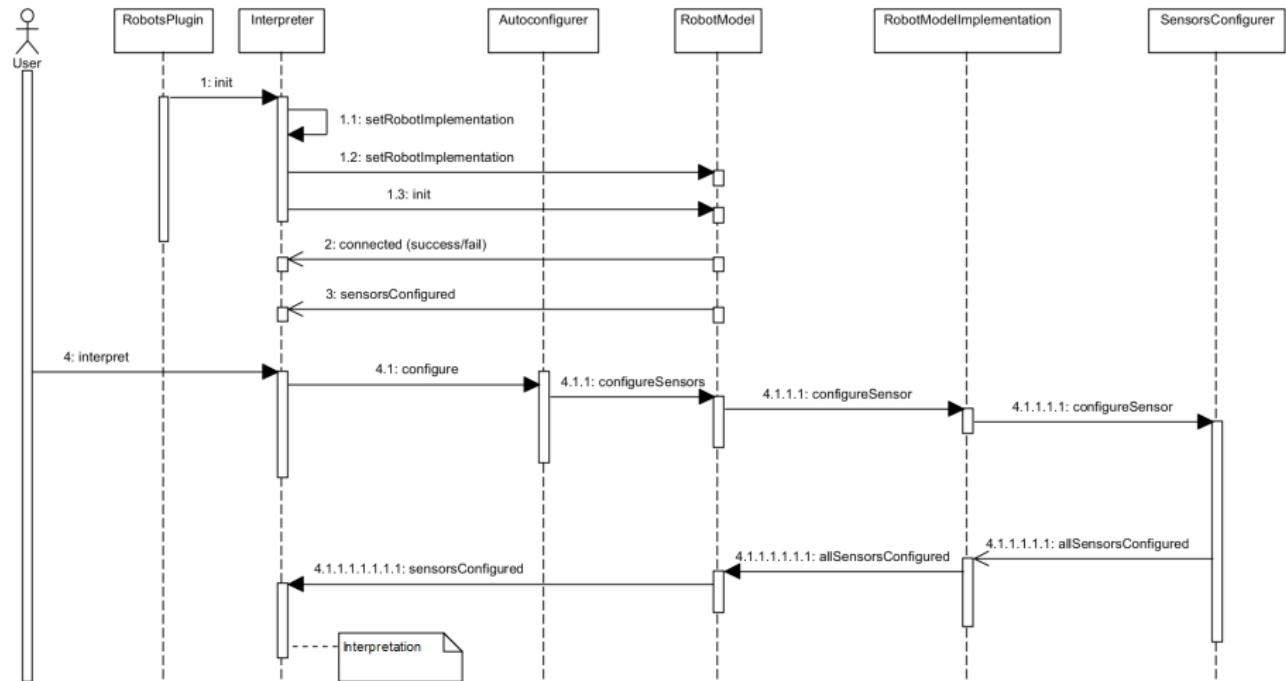
© М. Фаулер, UML. Основы

Ещё немного о синтаксисе

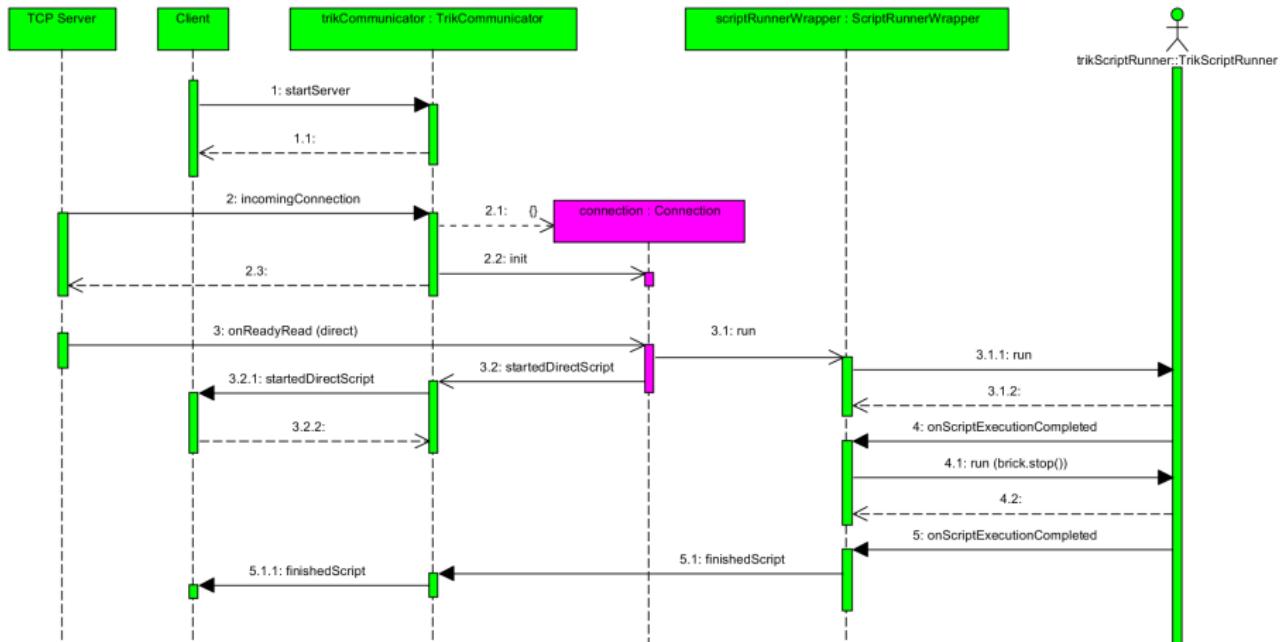


© М. Фаулер, UML. Основы

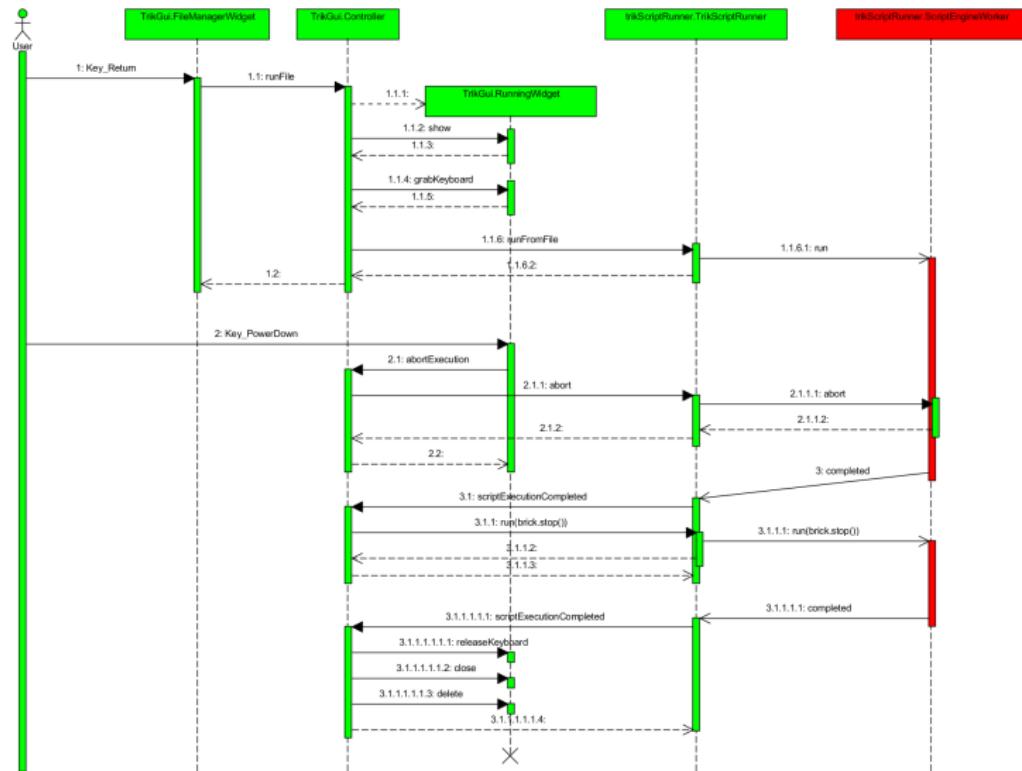
Пример



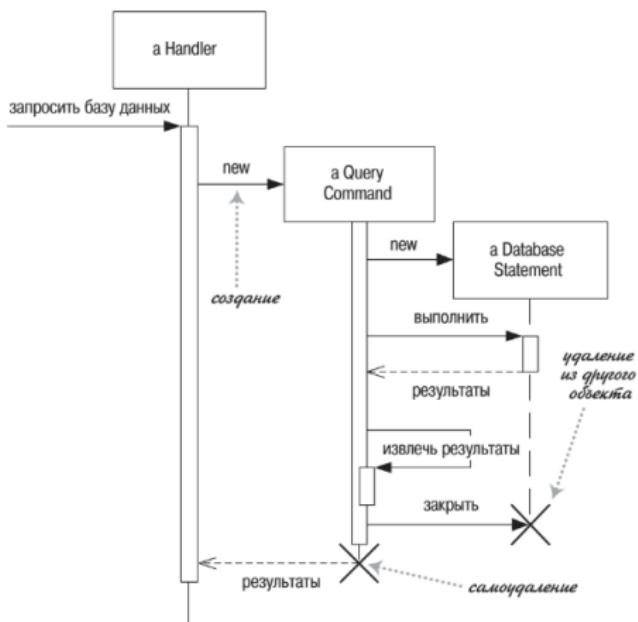
Ещё пример



И ещё пример



Создание и удаление объектов

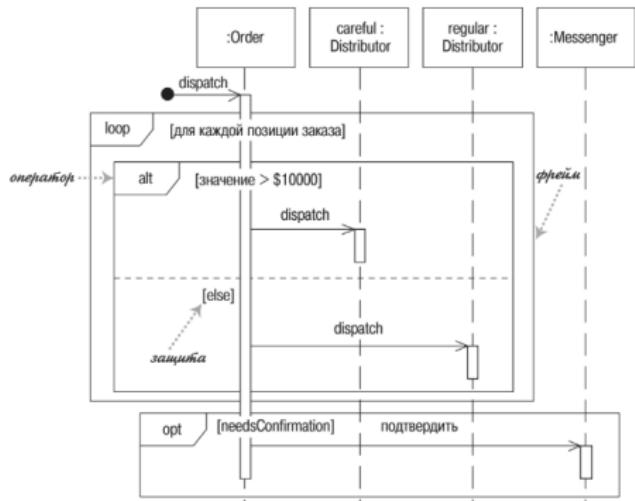


© М. Фаулер, UML. Основы

Фреймы

```

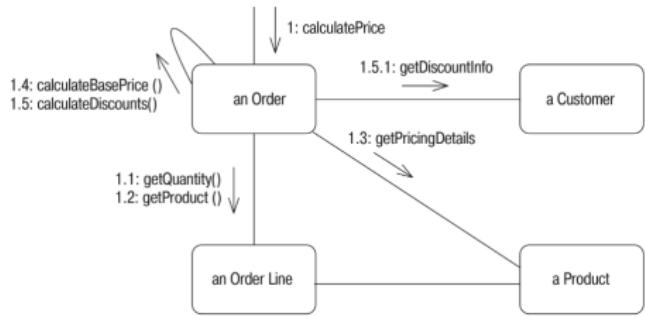
foreach (lineitem)
  if (product.value > $10K)
    careful.dispatch
  else
    regular.dispatch
  end if
end for
if (needsConfirmation)
  messenger.confirm
  
```



© М. Фаулер, UML. Основы

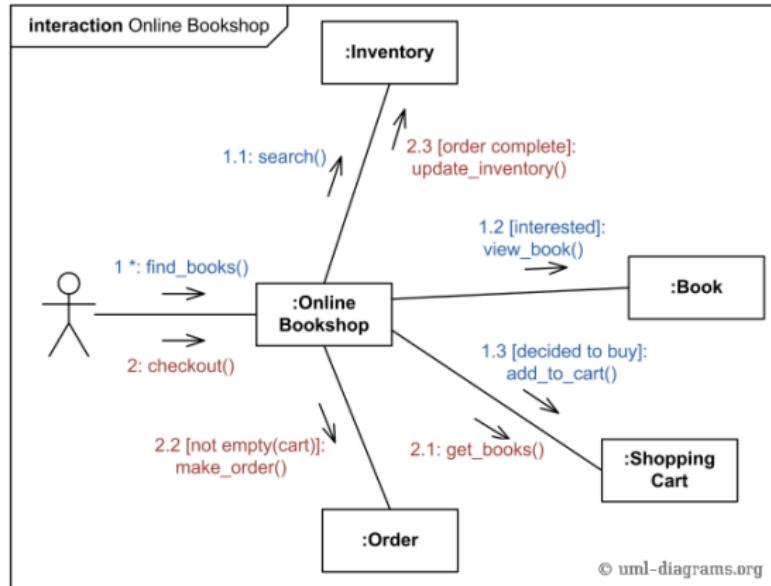
Коммуникационные диаграммы

- ▶ Применяются для визуализации взаимодействия между объектами
 - ▶ Более легковесный аналог диаграмм последовательностей
 - ▶ Тоже отображают один сценарий взаимодействия



© М. Фаулер, UML. Основы

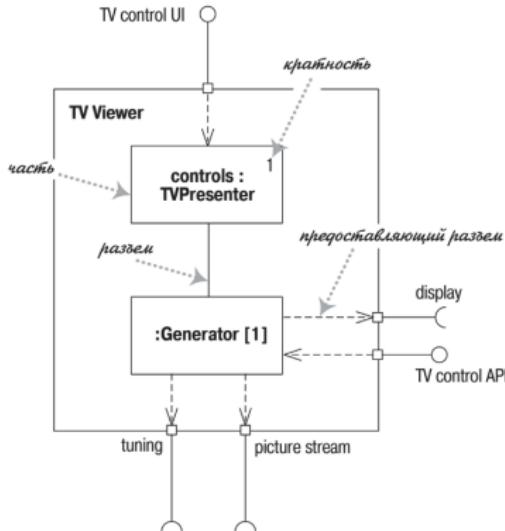
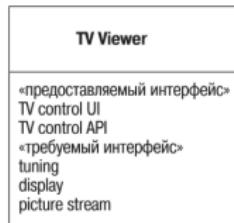
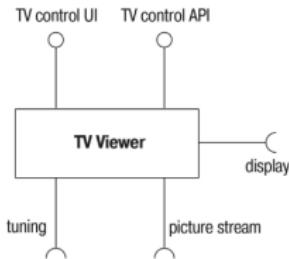
Коммуникационные диаграммы, пример



© <http://www.uml-diagrams.org/>

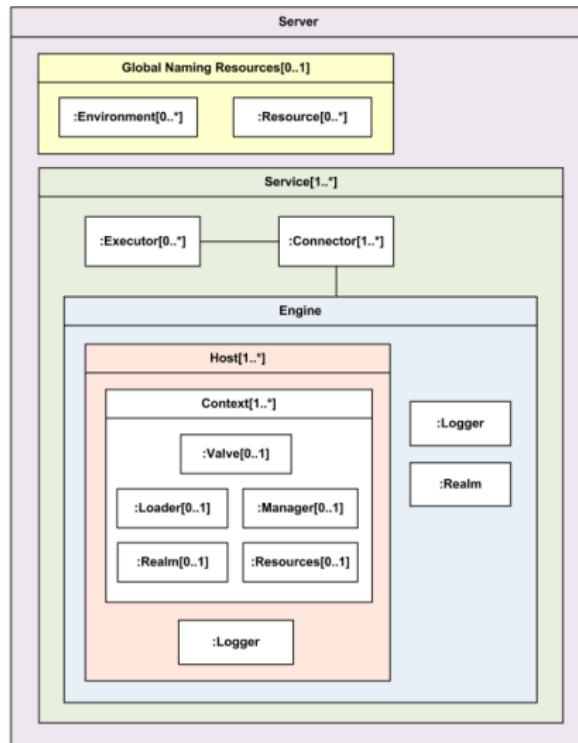
Диаграммы составных структур

- ▶ По сути, продвинутые диаграммы компонентов
- ▶ Внутри компоненты не другие компоненты, а части (роли)



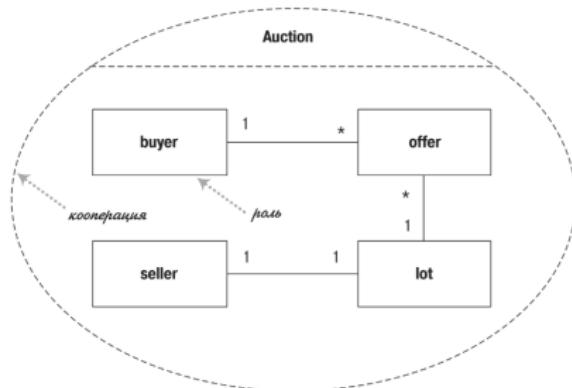
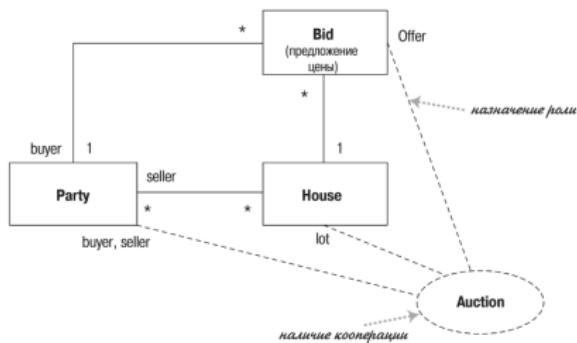
© М. Фаулер, UML. Основы

Диаграммы составных структур, пример

© <http://www.uml-diagrams.org/>

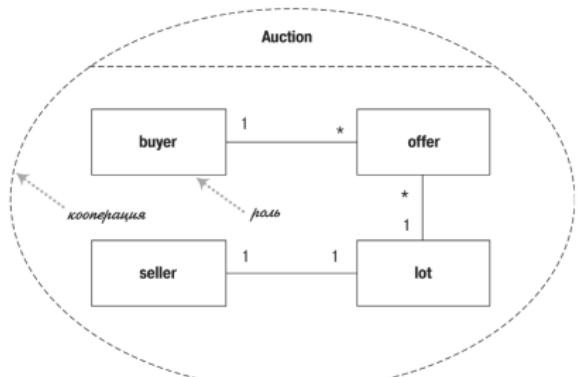
Диаграммы коопераций

- ▶ Показывают взаимодействие между объектами (ролями) в рамках одного сценария использования



© М. Фаулер, UML. Основы

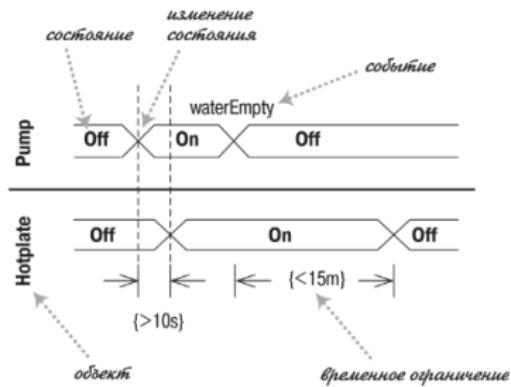
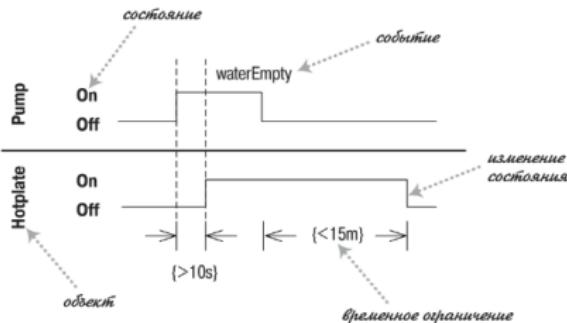
Диаграммы коопераций, последовательности



© М. Фаулер, UML. Основы

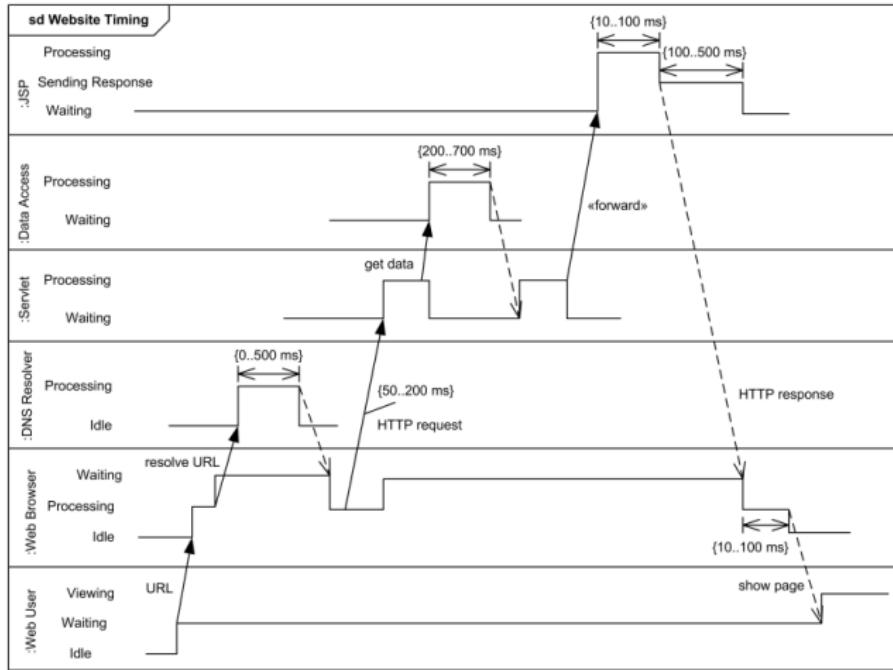
Временные диаграммы

- ▶ Для моделирования временных ограничений в системах реального времени



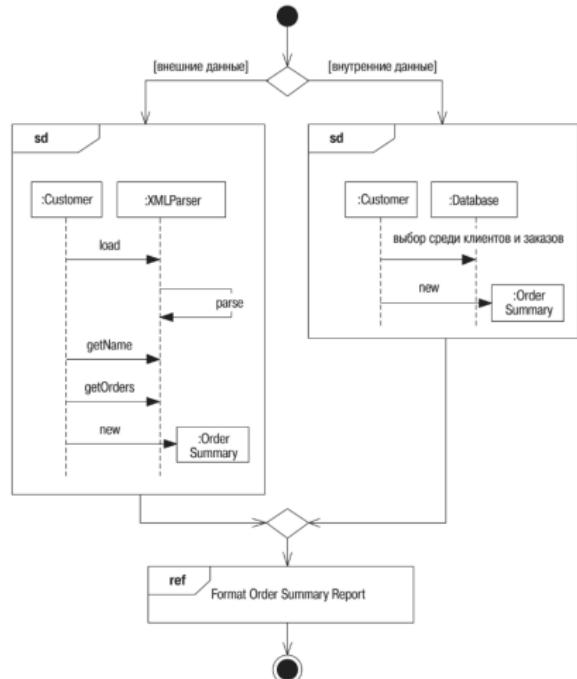
© М. Фаулер, UML. Основы

Временная диаграмма, пример

© <http://www.uml-diagrams.org/>

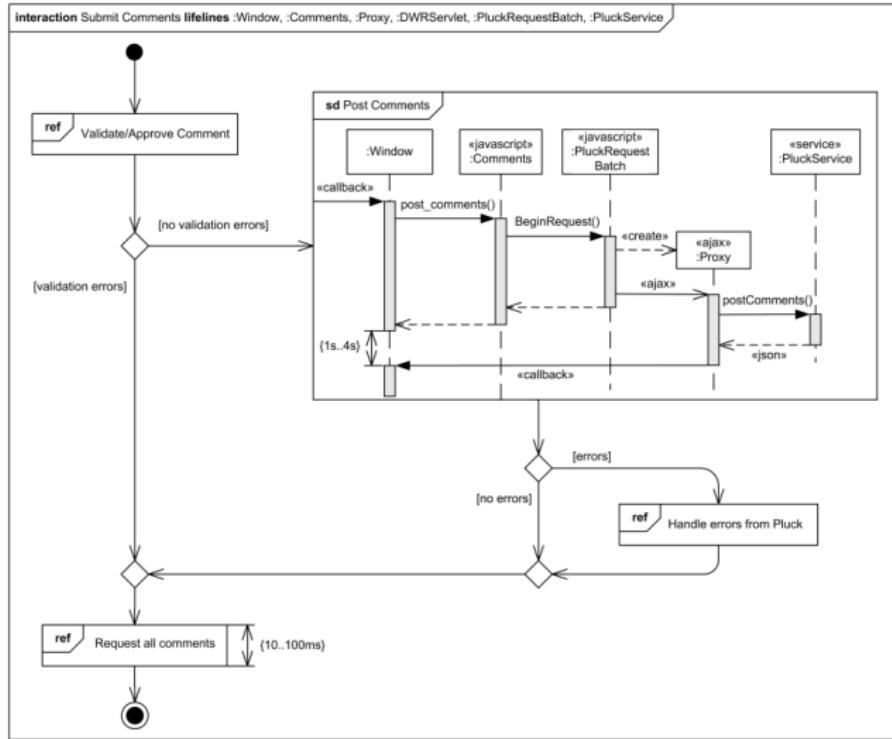
Диаграммы обзора взаимодействия

- ▶ Диаграммы активностей + диаграммы последовательностей
- ▶ Применяются при наличии взаимодействия со сложной логикой, когда фреймы неудобны



© М. Фаулер, UML. Основы

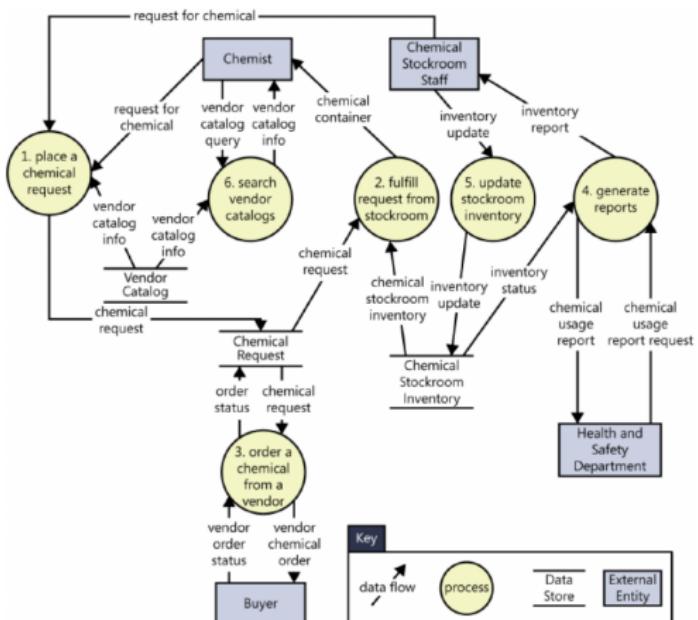
Диаграмма обзора взаимодействия, пример

© <http://www.uml-diagrams.org/>

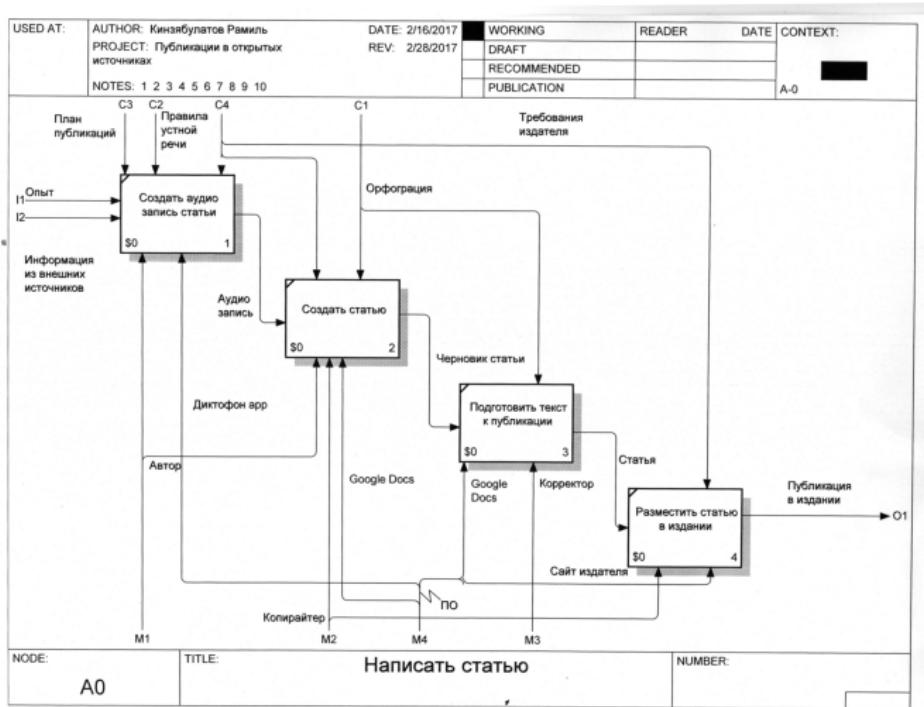
Диаграммы потоков данных

DFD

- ▶ Показывают обмен данными в системе
- ▶ Внешние сущности, процессы внутри системы, потоки данных



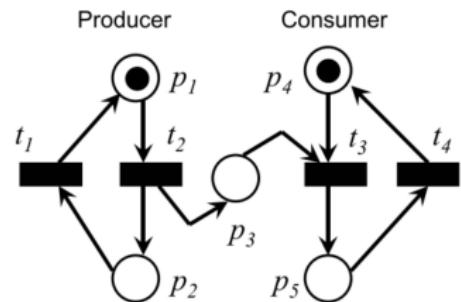
Диаграммы IDEF0



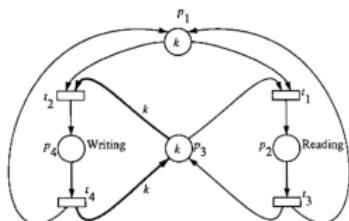
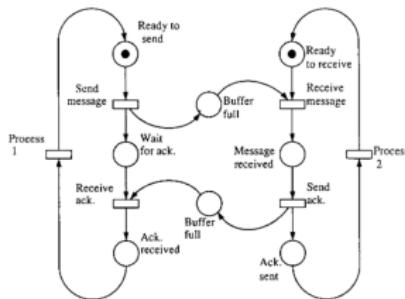
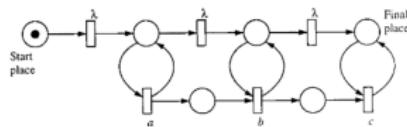
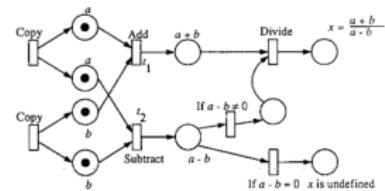
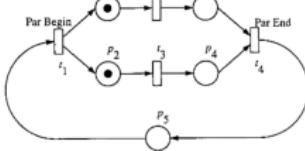
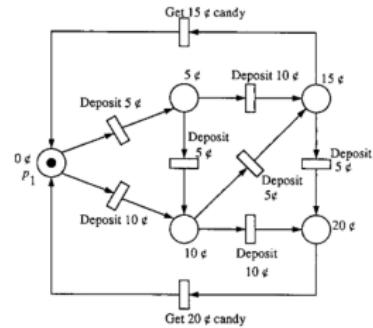
© <https://habrahabr.ru/post/322832/>

Сети Петри

- ▶ Тройка (P, T, ϕ)
 - ▶ Множество мест
 - ▶ Множество переходов
 - ▶ Функция потока
 $\phi : (P \times T) \cup (T \times P) \rightarrow N$
- ▶ Маркировка: $\mu : P \rightarrow N$
- ▶ Срабатывание (firing): $\mu \xrightarrow{t} \mu' :$
 $\mu'(p) = \mu(p) - \phi(p, t) + \phi(t, p), \forall p \in P$



Зачем это



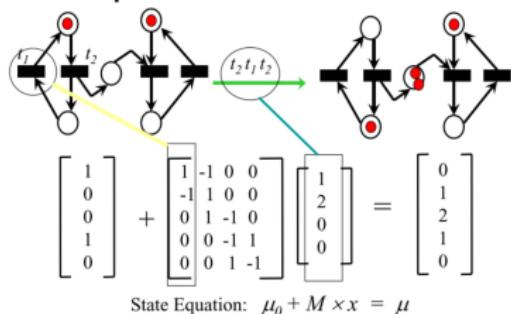
© Murata Tadao. Petri nets: Properties, analysis and applications

Свойства, которые можно проверить

- ▶ Поведенческие свойства:
 - ▶ Достигимость
 - ▶ Ограничность (безопасность)
 - ▶ Живость (L0 - L4)
 - ▶ “Реверсабельность” и “домашнее состояние”
 - ▶ ...
- ▶ Структурные свойства
 - ▶ Структурная живость
 - ▶ Полная контролируемость
 - ▶ Структурная ограниченность
 - ▶ ...

Способы анализа

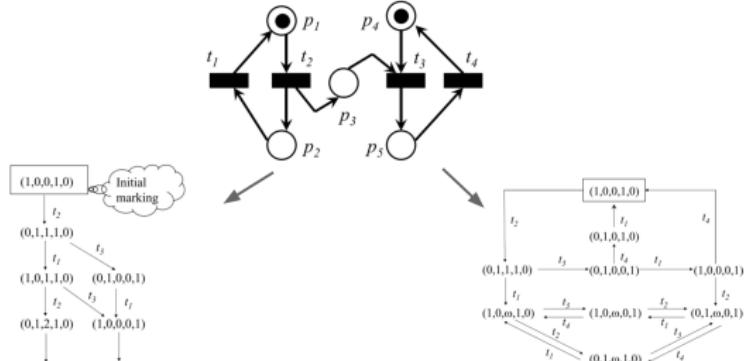
► Алгебраический



► Структурный

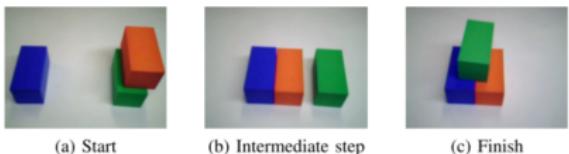
► Редукцией

► Пространства состояний



Пример использования (1)

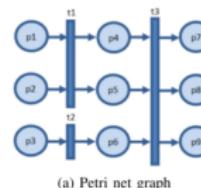
- ▶ Сеть Петри как форма представления знаний, генерится автоматически по демонстрации
 - ▶ Поиск по дереву достижимости для определения пути решения задачи или подзадачи
 - ▶ “как программисту вскипятить чайник, если в нём уже есть вода?”



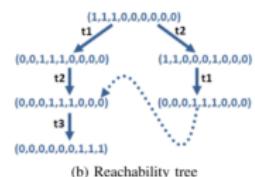
(a) Start

(b) Intermediate step

(c) Finish



(a) Petri net graph



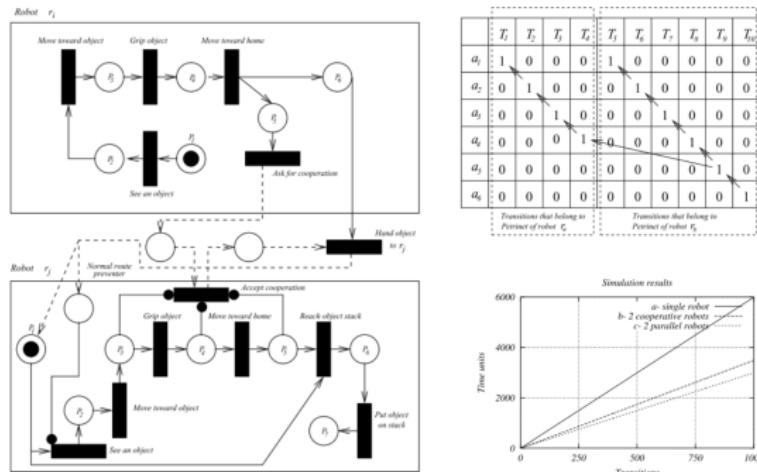
(b) Reachability tree

© Robot Task Learning from Demonstration

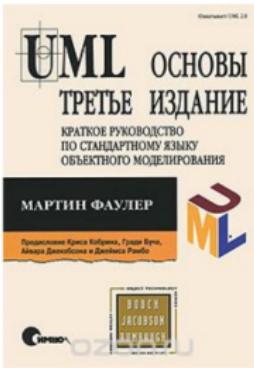
Using Petri Nets

Пример использования (2)

- ▶ Сеть Петри как план работы, состоящий из элементарных действий гетерогенных агентов
- ▶ Автоматическое распределение работ



Книжка



М. Фаулер, UML. Основы. Краткое руководство по стандартному языку объектного моделирования. СПб., Символ-Плюс, 2011. 192 С.

Лекция 6: Структурные шаблоны

Юрий Литвинов
yurii.litvinov@gmail.com

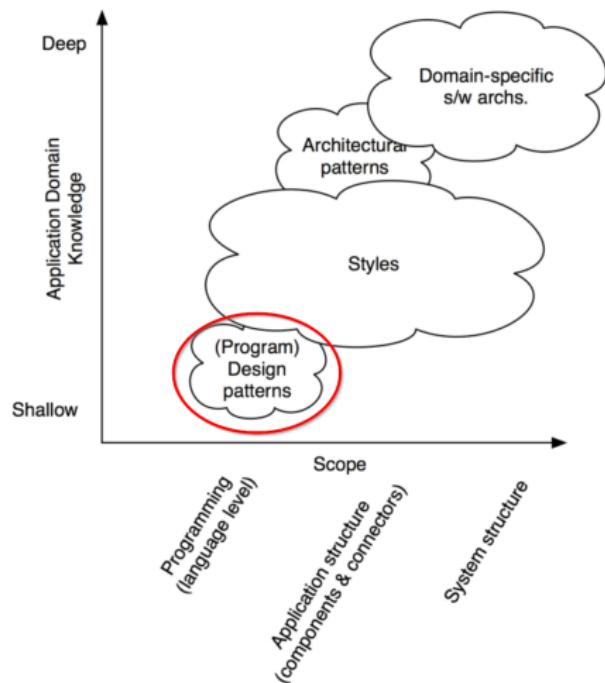
08.10.2020г

Паттерны проектирования

Шаблон проектирования — это повторимая архитектурная конструкция, являющаяся решением некоторой типичной технической проблемы

- ▶ Подходит для класса проблем
- ▶ Обеспечивает переиспользуемость знаний
- ▶ Позволяет унифицировать терминологию
- ▶ В удобной для изучения форме
- ▶ НЕ конкретный рецепт или указания к действию

Паттерны и архитектурные стили



© N. Medvidovic

Книжка про паттерны

Must read!

Приемы объектно-ориентированного проектирования. Паттерны проектирования

Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес

Design Patterns: Elements of Reusable Object-Oriented Software



Начнём с примера

Текстовый редактор

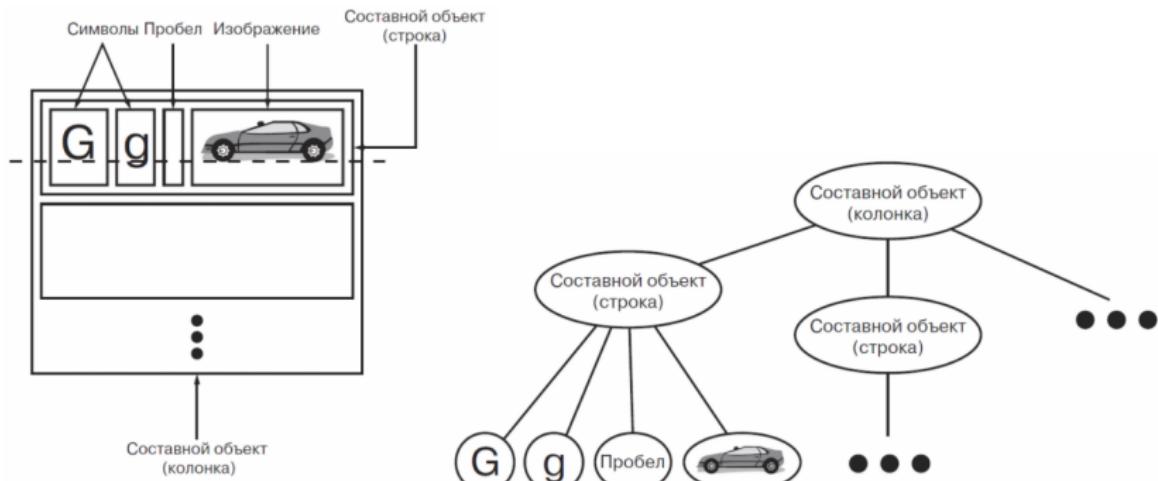
WYSIWYG-редактор, основные вопросы:

- ▶ Структура документа
- ▶ Форматирование
- ▶ Создание привлекательного интерфейса пользователя
- ▶ Поддержка стандартов внешнего облика программы
- ▶ Операции пользователя, undo/redo
- ▶ Проверка правописания и расстановка переносов

Структура документа

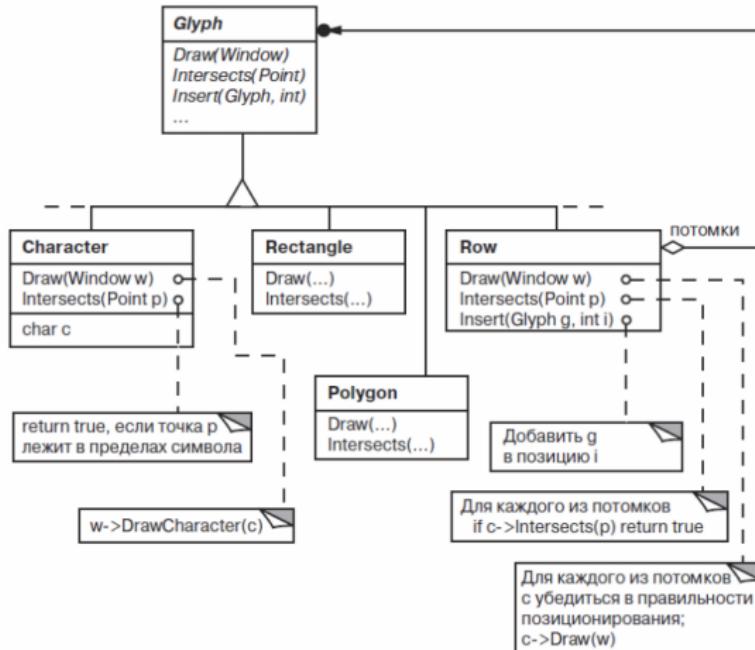
- ▶ Документ — множество графических элементов
 - ▶ Организация в физическую структуру
 - ▶ Средства UI для манипулирования структурой
- ▶ Требования к внутреннему представлению
 - ▶ Отслеживание внутренней структуры документа
 - ▶ Генерирование визуального представления
 - ▶ Отображение позиций экрана на внутреннее представление
- ▶ Ограничения
 - ▶ Текст и графика едины
 - ▶ Простой и составной элементы едины

Рекурсивная композиция



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

Диаграмма классов: глифы

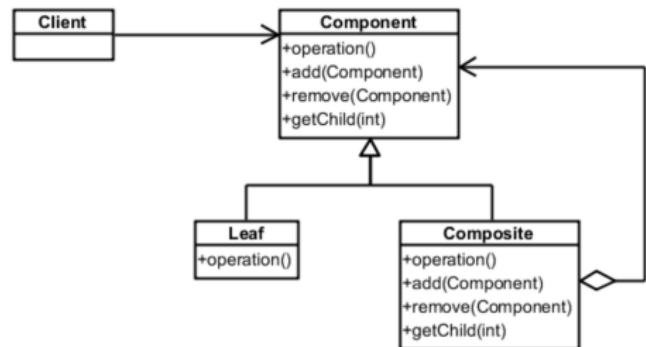


© Э. Гамма и др., Приемы объектно-ориентированного проектирования

Паттерн “Компоновщик”

Composite

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



“Компоновщик” (Composite), детали реализации

- ▶ Ссылка на родителя
 - ▶ Может быть полезна для простоты обхода
 - ▶ “Цепочка обязанностей”
 - ▶ Но дополнительный инвариант
 - ▶ Обычно реализуется в Component
- ▶ Разделяемые поддеревья и листья
 - ▶ Позволяют сильно экономить память
 - ▶ Проблемы с навигацией к родителям и разделяемым состоянием
 - ▶ Паттерн “Приспособленец”
- ▶ Идеологические проблемы с операциями для работы с потомками
 - ▶ Не имеют смысла для листа
 - ▶ Можно считать Leaf Composite-ом, у которого всегда 0 потомков
 - ▶ Операции add и remove можно объявить и в Composite, тогда придётся делать cast
 - ▶ Иначе надо бросать исключения в add и remove

“Компоновщик”, детали реализации (2)

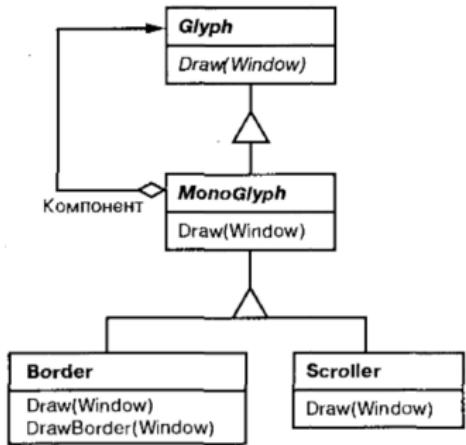
- ▶ Операция `getComposite()` – более аккуратный аналог `cast-a`
- ▶ Где определять список потомков
 - ▶ В `Composite`, экономия памяти
 - ▶ В `Component`, единообразие операций
 - ▶ “Список” вполне может быть хеш-таблицей, деревом или чем угодно
- ▶ Порядок потомков может быть важен, может нет
- ▶ Кеширование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

Усовершенствование UI

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

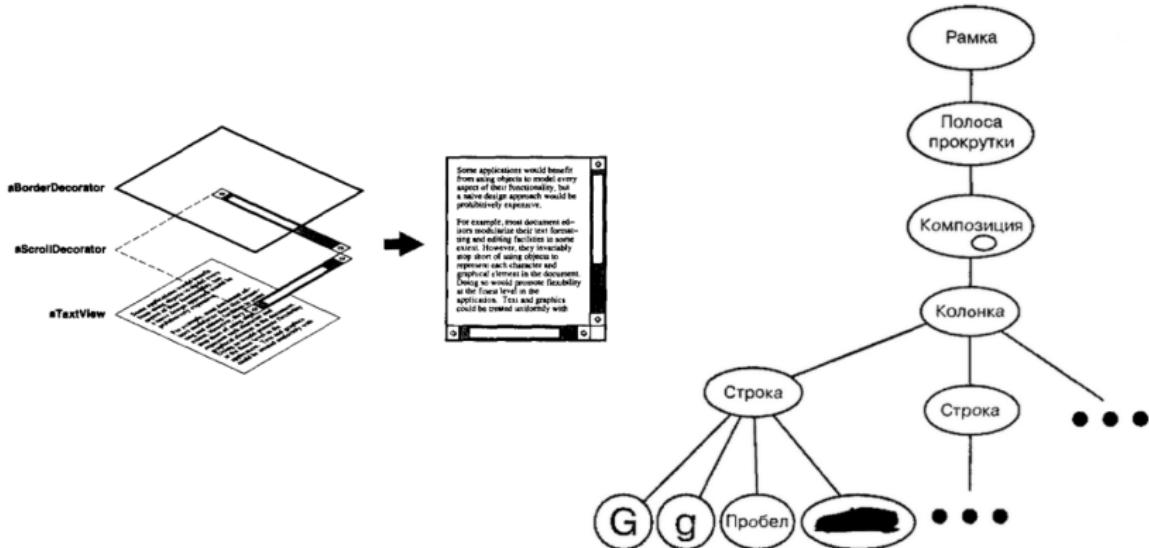
Моноглиф

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



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

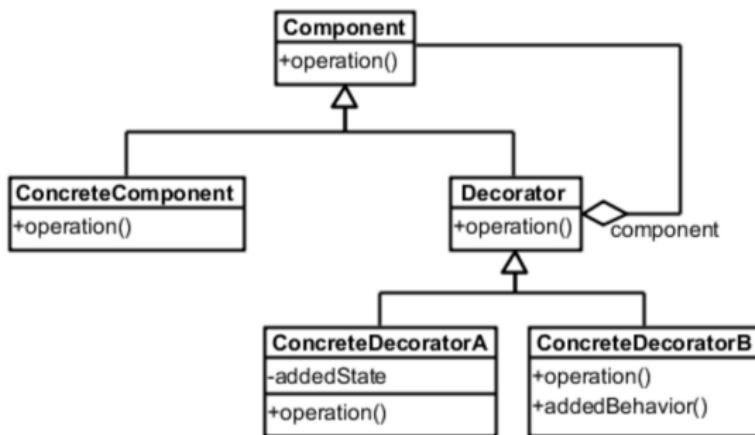
Структура глифов



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

Паттерн “Декоратор”

Decorator

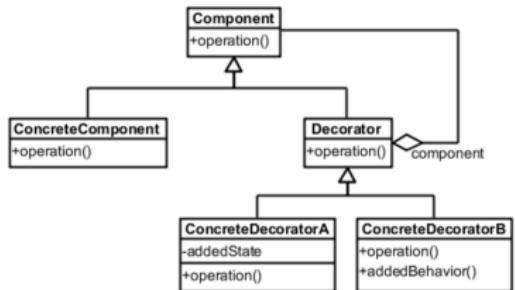


Декоратор, особенности

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

“Декоратор” (Decorator), детали реализации

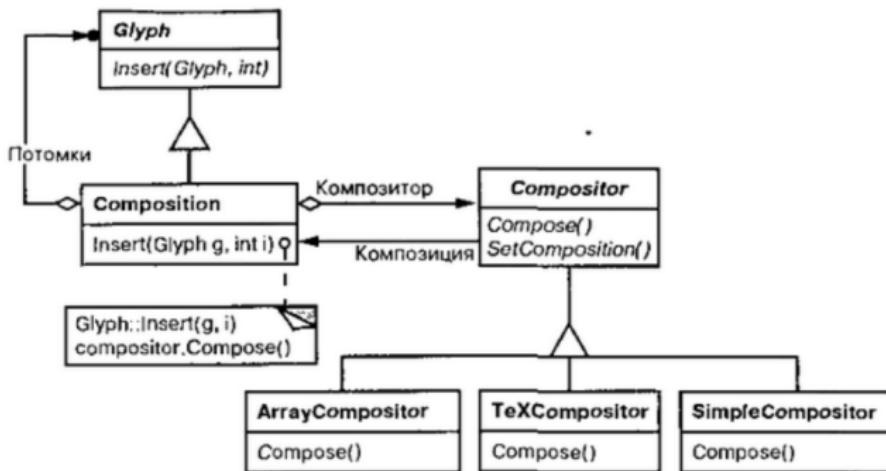
- ▶ Интерфейс декоратора должен соответствовать интерфейсу декорируемого объекта
 - ▶ Иначе получится “Адаптер”
- ▶ Если конкретный декоратор один, абстрактный класс можно не делать
- ▶ Component должен быть по возможности небольшим (в идеале, интерфейсом)
 - ▶ Иначе лучше паттерн “Стратегия”
 - ▶ Или самодельный аналог, например, список “расширений”, которые вызываются декорируемым объектом вручную перед операцией или после неё



Форматирование текста

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

Compositor и Composition

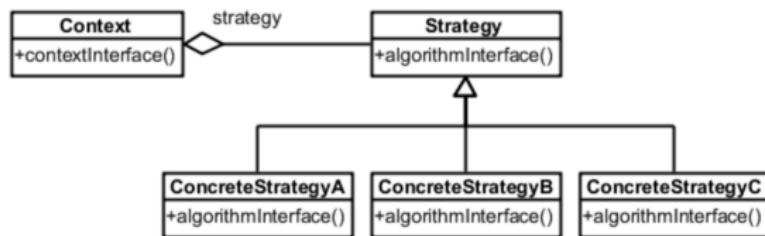


© Э. Гамма и др., Приемы объектно-ориентированного проектирования

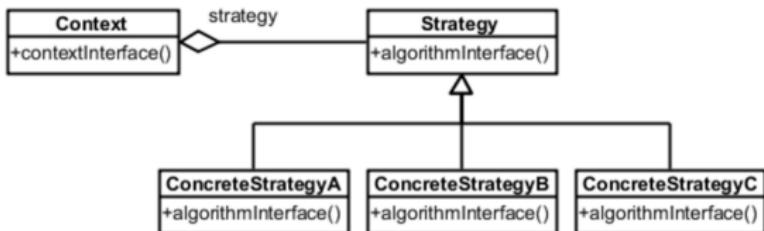
Паттерн “Стратегия”

Strategy

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



“Стратегия” (Strategy), детали реализации



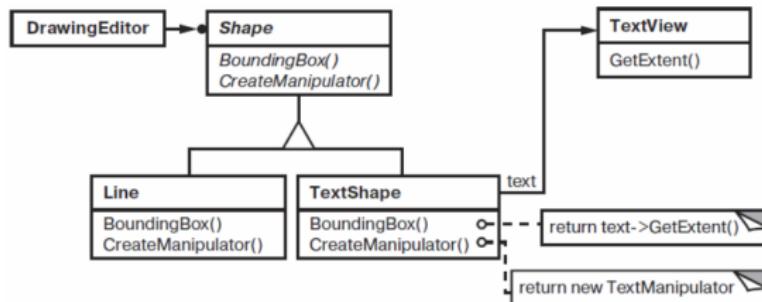
- ▶ Передача контекста вычислений в стратегию
 - ▶ Как параметры метода — уменьшает связность, но некоторые параметры могут быть стратегии не нужны
 - ▶ Передавать сам контекст в качестве аргумента — в Context интерфейс для доступа к данным

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

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

Проблема неподходящих интерфейсов

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

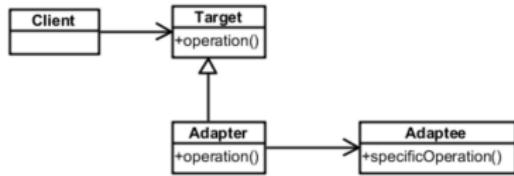


© Э. Гамма и др., Приемы объектно-ориентированного проектирования

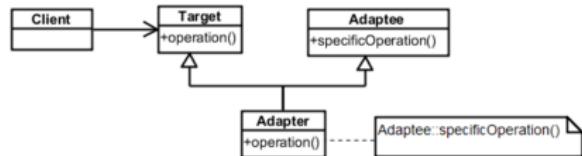
Паттерн “Адаптер”

Adapter

- ▶ Адаптер объекта:



- ▶ Адаптер класса:



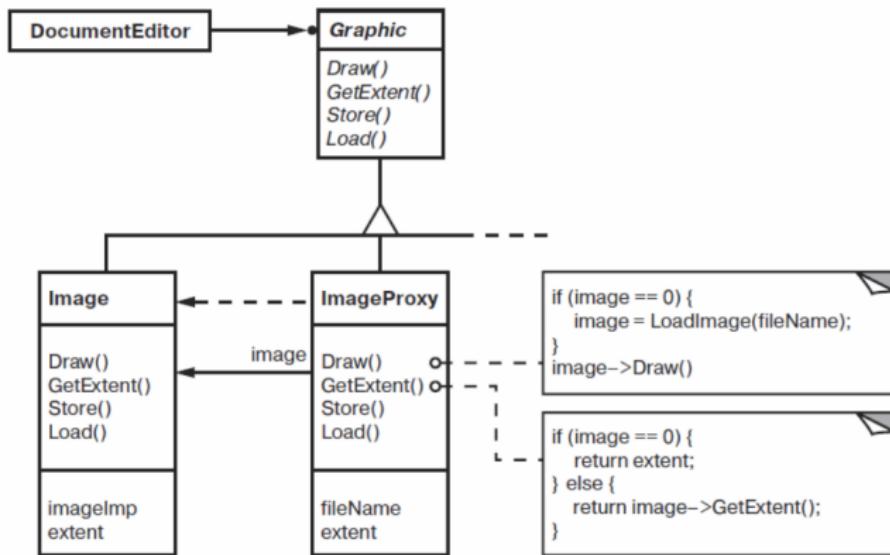
- ▶ Нужно множественное наследование
 - ▶ private-наследование в C++

Управление доступом к объектам

- ▶ Встраивание в документ графических объектов
 - ▶ Затраты на создание могут быть значительными
 - ▶ Хотим отложить их на момент использования
- ▶ Использование заместителей объектов

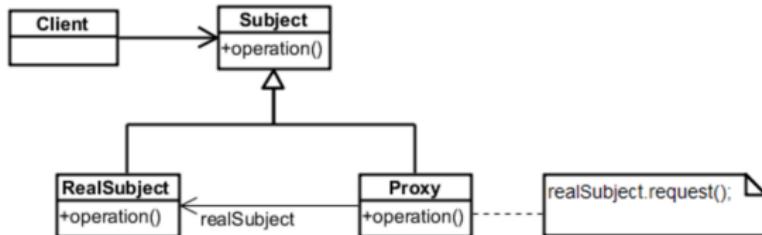


Отложенная загрузка изображения



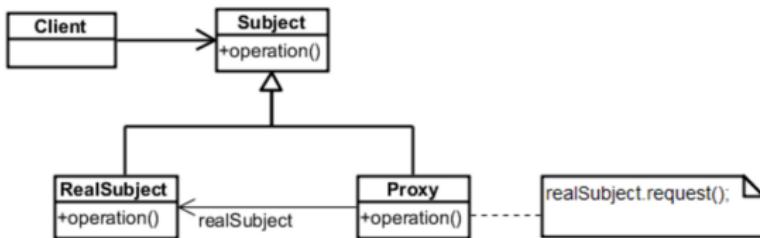
Паттерн “Заместитель”

Proxy



- ▶ Замещение удалённых объектов
- ▶ Создание “тяжёлых” объектов по требованию
- ▶ Контроль доступа
- ▶ Умные указатели
 - ▶ Подсчёт ссылок
 - ▶ Ленивая загрузка/инициализация
 - ▶ Работа с блокировками
 - ▶ Копирование при записи

“Заместитель”, детали реализации



- ▶ Перегрузка оператора доступа к членам класса (для C++)
 - ▶ Умные указатели так устроены
 - ▶ C++ вызывает операторы `->` по цепочке
 - ▶ `object->do()` может быть хоть
`((object.operator->()).operator->()).do()`
 - ▶ Не подходит, если надо различать операции

“Заместитель”, детали реализации (2)

- ▶ Реализация “вручную” всех методов проксируемого объекта
 - ▶ Сотня методов по одной строчке каждый
 - ▶ C#/F#: **public void do()** => **realSubject.do()**;
 - ▶ Препроцессор/генерация
 - ▶ Технологии наподобие WCF
- ▶ Проксируемого объекта может не быть в памяти

Паттерн “Фасад”

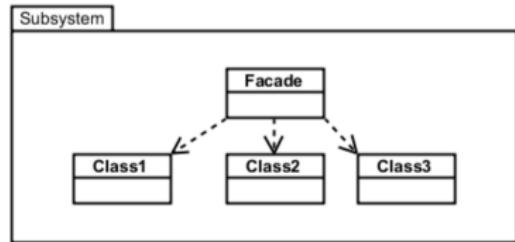
Facade

- ▶ Простой интерфейс к сложной системе
- ▶ Отделение подсистем от клиента и друг от друга
- ▶ Многоуровневая архитектура

“Фасад” (Facade), детали реализации

► Абстрактный Facade

- ▶ Существенно снижает связность клиента с подсистемой



► Открытые и закрытые классы подсистемы

- ▶ Пространства имён и пакеты помогают, но требуют дополнительных соглашений
 - ▶ Пространство имён details
- ▶ Инкапсуляция целой подсистемы — это хорошо

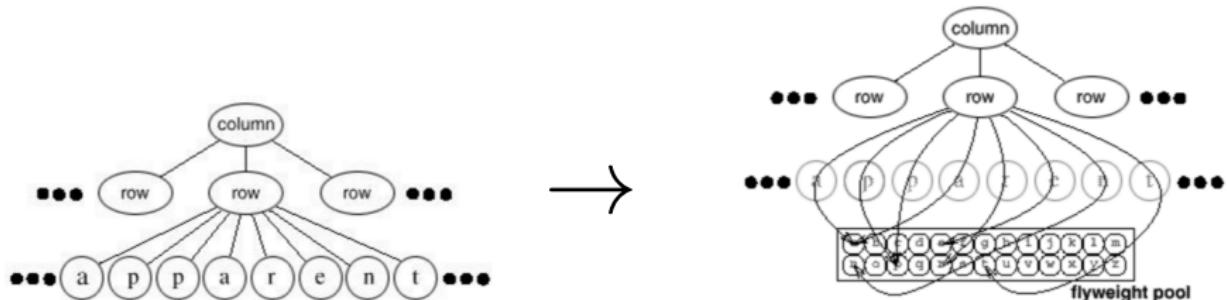
Паттерн “Приспособленец” (Flyweight)

Предназначается для эффективной поддержки множества мелких объектов

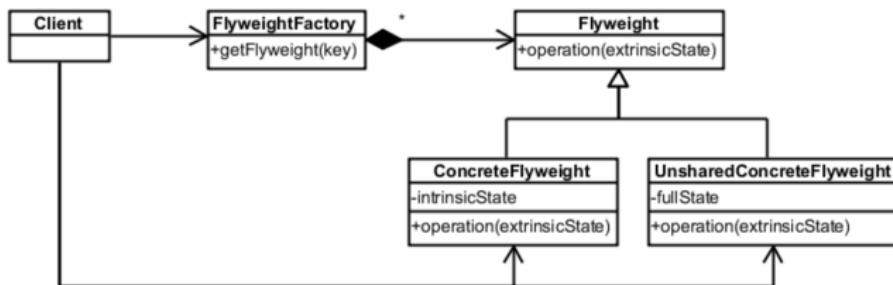
Пример:

- ▶ Есть текстовый редактор
- ▶ Хочется работать с каждым символом как с объектом
 - ▶ Единообразие алгоритмов форматирования и внутренней структуры документа
 - ▶ Более красивая и ООПшная реализация
 - ▶ Паттерн “Компоновщик”, структура “Символ” → “Строка” → “Страница”
- ▶ Наивная реализация привела бы к чрезмерной расточительности по времени работы и по памяти, потому что документы с миллионами символов не редкость

“Приспособленец”, пример

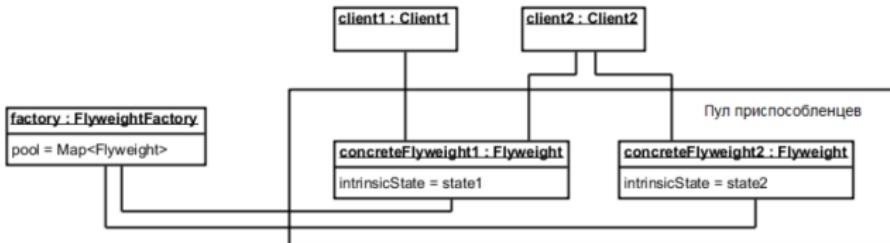


“Приспособленец”, общая схема



- ▶ *Flyweight* — определяет интерфейс, через который приспособленцы могут получать внешнее состояние
- ▶ *ConcreteFlyweight* — реализует интерфейс *Flyweight* и может иметь внутреннее состояние, не зависят от контекста
- ▶ *UnsharedConcreteFlyweight* — неразделяемый “приспособленец”, хранящий всё состояние в себе, бывает нужен, чтобы собирать иерархические структуры из *Flyweight*-ов (“Компоновщик”)
- ▶ *FlyweightFactory* — содержит пул приспособленцев, создаёт их и управляет их жизнью

"Приспособленец", диаграмма объектов



- ▶ Клиенты могут быть разных типов
- ▶ Клиенты могут разделять приспособленцев
 - ▶ Один клиент может иметь несколько ссылок на одного приспособленца
- ▶ Во время выполнения клиенты имеют право не знать про фабрику

Когда применять

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

Тонкости реализации

- ▶ Внешнее состояние — по сути, отдельный объект, поэтому если различных внешних состояний столько же, сколько приспособленцев, смысла нет
 - ▶ Один объект-состояние покрывает сразу несколько приспособленцев
 - ▶ Например, объект "Range" может хранить параметры форматирования для всех букв внутри фрагмента
- ▶ Клиенты не должны инстанцировать приспособленцев сами, иначе трудно обеспечить разделение
 - ▶ Имеет смысл иметь механизм для удаления неиспользуемых приспособленцев
 - ▶ Если их может быть много
- ▶ Приспособленцы немутабельны и Value Objects (с правильно переопределённой операцией сравнения)
 - ▶ Про hashCode() тоже надо не забыть

Лекция 7: Структурные и порождающие шаблоны

Юрий Литвинов
yurii.litvinov@gmail.com

15.10.2020г

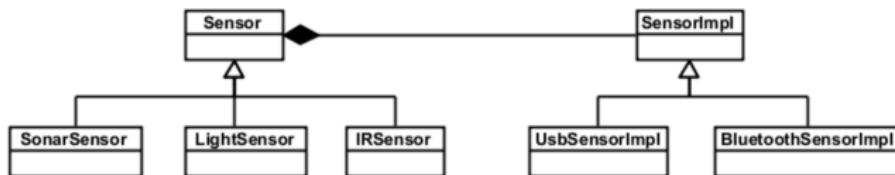
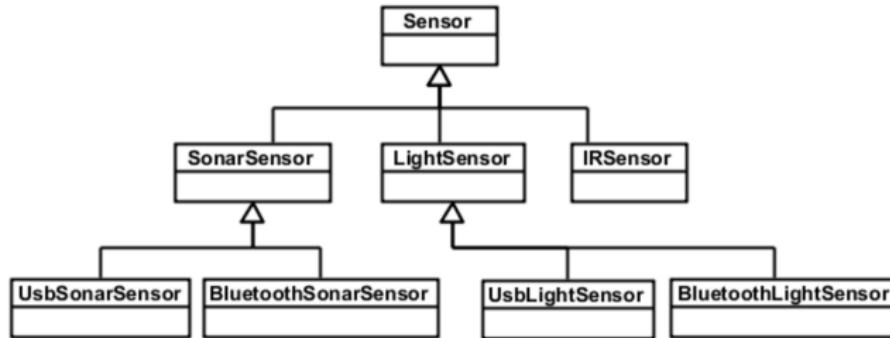
Паттерн “Мост” (Bridge)

Отделяет абстракцию от реализации

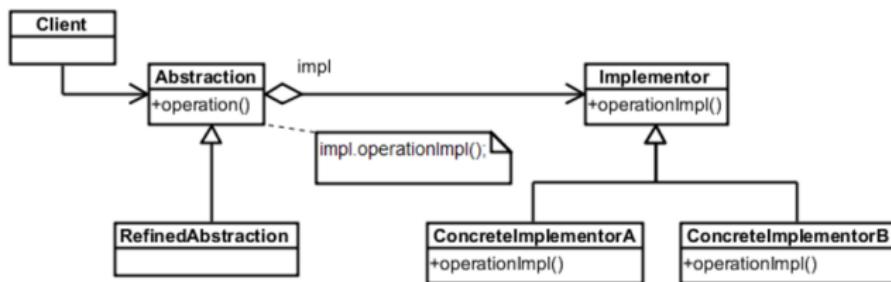
Пример:

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

“Мост”, пример



“Мост”, общая схема



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

Когда применять

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

Тонкости реализации

Создание правильного Implementor-a

- ▶ Самой абстракцией в конструкторе, в зависимости от переданных параметров
 - ▶ Как вариант — выбор реализации по умолчанию и замена её по ходу работы
- ▶ Принимать реализацию извне (как параметр конструктора, или, реже, как значение в сеттер)
- ▶ Фабрика/фабричный метод
 - ▶ Позволяет спрятать платформозависимые реализации, чтобы не зависеть от них всех при сборке

Pointer To Implementation (PImpl)

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

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

- ▶ Позитивно сказывается на времени компиляции программ на C++
- ▶ Позволяет менять реализацию независимо
 - ▶ Сохраняя бинарную совместимость

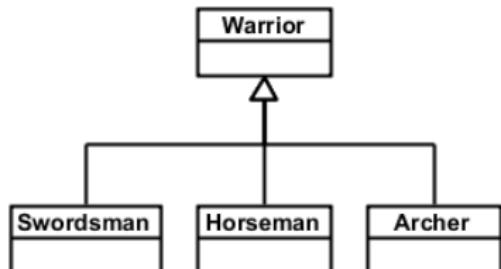
Как: предварительное объявление класса-реализации, полное определение — в .cpp-файле вместе с методами абстракции

Часто используется в реализации библиотек (например, Qt)

“Фабричный метод” МОТИВАЦИЯ

Игра-стратегия

- ▶ Воины
 - ▶ Мечники
 - ▶ Конница
 - ▶ Лучники
- ▶ Общее поведение
- ▶ Общие характеристики



Виртуальный конструктор

```
enum WarriorId { SwordsmanId, ArcherId, HorsemanId };
```

```
class Warrior
```

```
{
```

```
public:
```

```
    Warrior(WarriorId id)
```

```
{
```

```
    if (id == SwordsmanId) p = new Swordsman;  

    else if (id == ArcherId) p = new Archer;  

    else if (id == HorsemanId) p = new Horseman;  

    else assert( false);
```

```
}
```

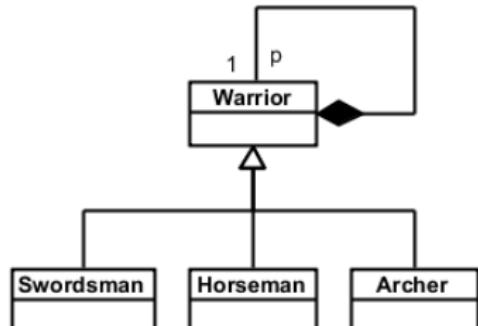
```
    virtual void info() { p->info(); }
```

```
    virtual ~Warrior() { delete p; p = 0; }
```

```
private:
```

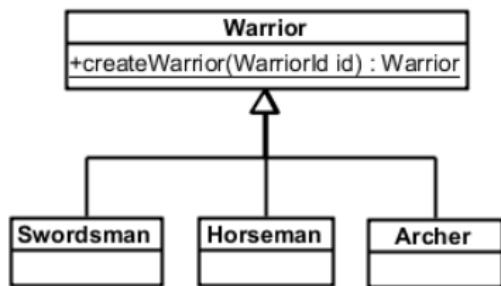
```
    Warrior* p;
```

```
};
```



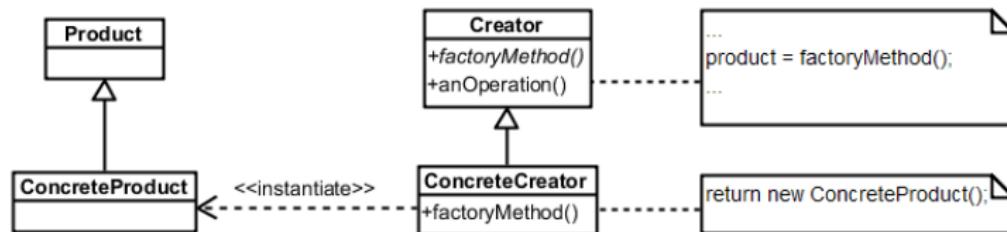
Фабричный метод

- ▶ Базовый класс знает про остальные
- ▶ switch в createWarrior()



Паттерн “Factory Method”

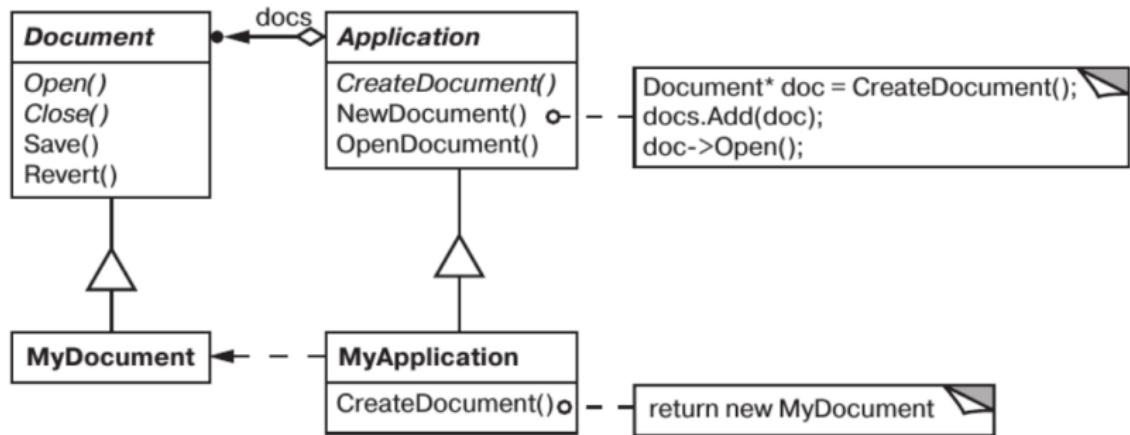
Factory Method



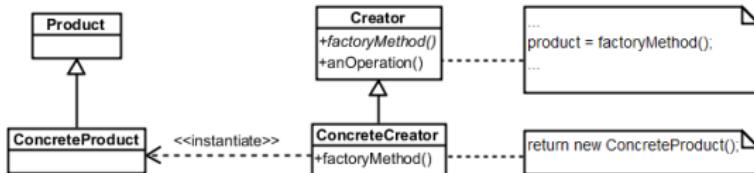
► Применимость:

- ▶ классу заранее неизвестно, объекты каких классов ему нужно создавать
- ▶ объекты, которые создает класс, специфицируются подклассами
- ▶ класс делегирует свои обязанности одному из нескольких вспомогательных подклассов

Пример, текстовый редактор



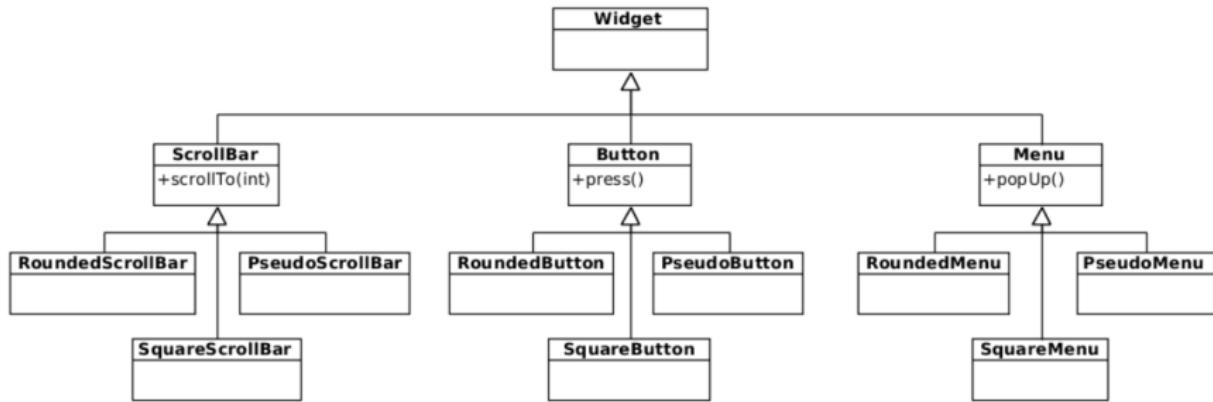
“Фабричный метод”, детали реализации



- ▶ Абстрактный Creator или реализация по умолчанию
 - ▶ Второй вариант может быть полезен для расширяемости
- ▶ Параметризованные фабричные методы
- ▶ Если язык поддерживает инстанциацию по прототипу (JavaScript, Smalltalk), можно хранить порождаемый объект
- ▶ Creator не может вызывать фабричный метод в конструкторе
- ▶ Можно сделать шаблонный Creator

"Абстрактная фабрика", мотивация

- ▶ Хотим поддержать разные стили UI
 - ▶ Гибкая поддержка в архитектуре
 - ▶ Удобное добавление новых стилей



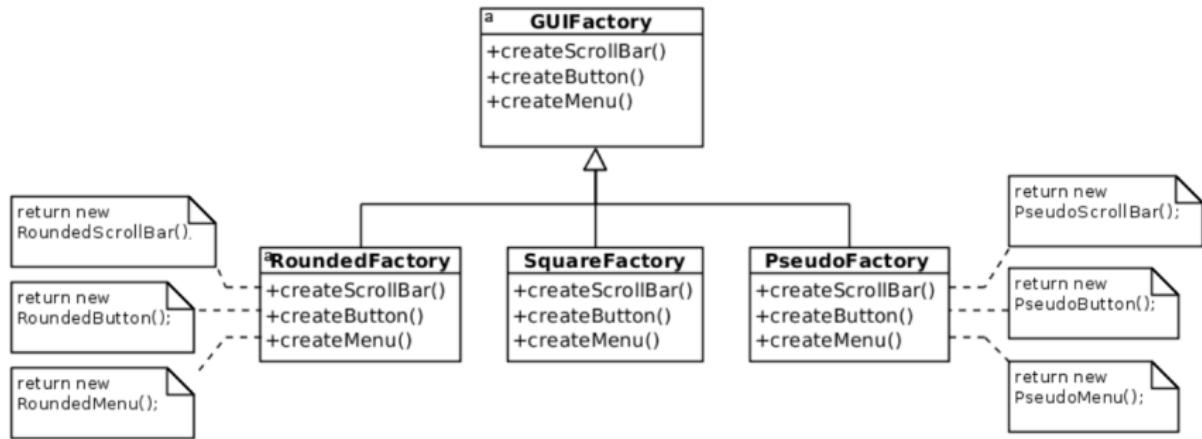
Создание виджетов

ScrollBar* bar = **new** RoundedScrollBar;

vs

ScrollBar* bar = guiFactory->createScrollBar();

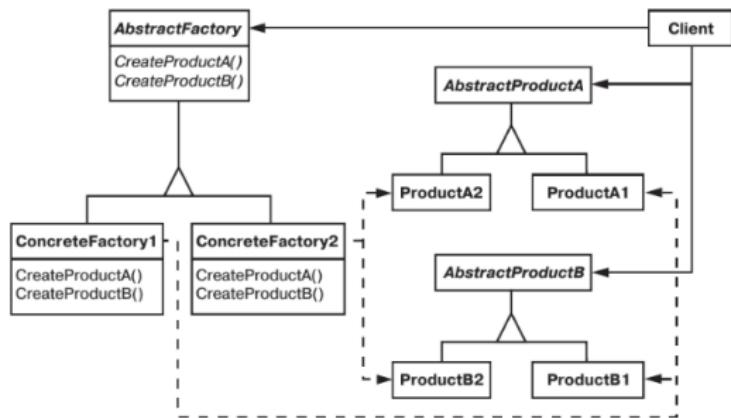
Фабрика виджетов



Паттерн “Абстрактная фабрика”

Abstract Factory

- ▶ Изолирует конкретные классы
- ▶ Упрощает замену семейств продуктов
- ▶ Гарантирует сочетаемость продуктов
- ▶ Поддержать новый вид продуктов непросто

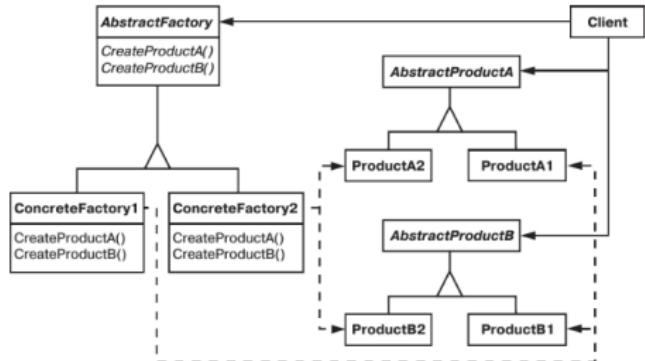


"Абстрактная фабрика", применимость

- ▶ Система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты
- ▶ Система должна конфигурироваться одним из семейств составляющих ее объектов
- ▶ Взаимосвязанные объекты должны использоваться вместе
- ▶ Хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию

“Абстрактная фабрика”, детали реализации

- ▶ Хорошо комбинируются с паттерном “Одиночка”
- ▶ Если семейств продуктов много, то фабрика может инициализироваться прототипами, тогда не надо создавать сотню подклассов
- ▶ Прототип на самом деле может быть классом (например, Class в Java)
- ▶ Если виды объектов часто меняются, может помочь параметризация метода создания
 - ▶ Может пострадать типобезопасность



Паттерн “Одиночка”

Singleton

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

Singleton
-uniqueInstance
-singletonData
-Singleton()
+instance()
+singletonOperation()
+getSingletonData()

“Одиночка”, наивная реализация

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

“Одиночка”, простая многопоточная реализация

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
  
    private Singleton () {}  
  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

"Одиночка", плохая многопоточная реализация

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

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;  
    }  
}
```

"Multiton"

- ▶ Реестр одиночек, обеспечивает уникальность объекта по ключу
 - ▶ Сам создаёт объекты
 - ▶ Не даёт возможности зарегистрировать объект извне

Multiton
-instances : Map<Key, Multiton>
-Multiton()
+instance() : Multiton

“Одиночка”, критика

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

Паттерн “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

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”

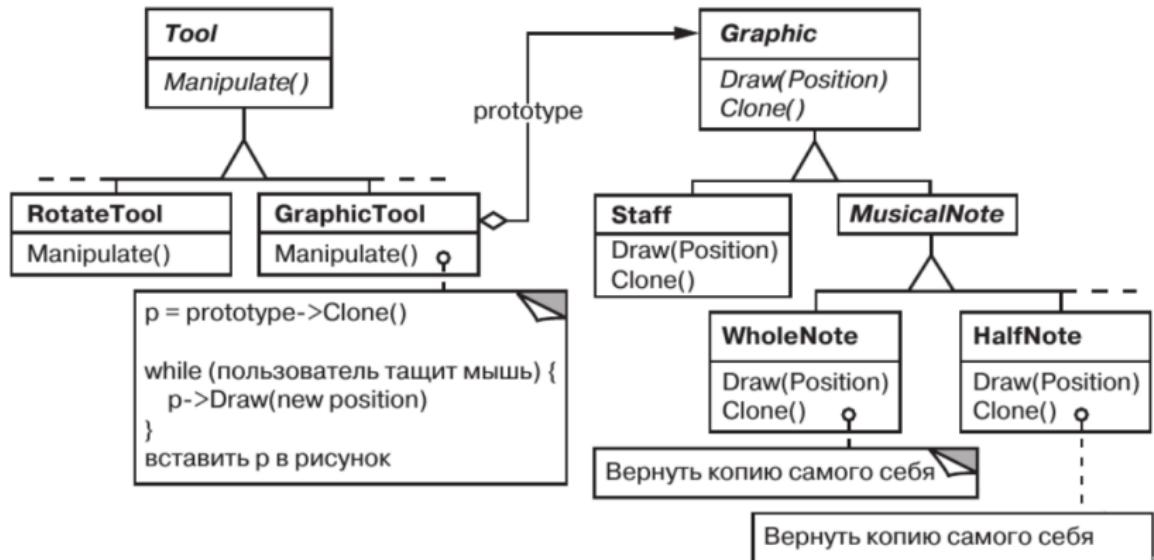
Решение: пул потоков в .NET

- ▶ 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 выполнять 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”

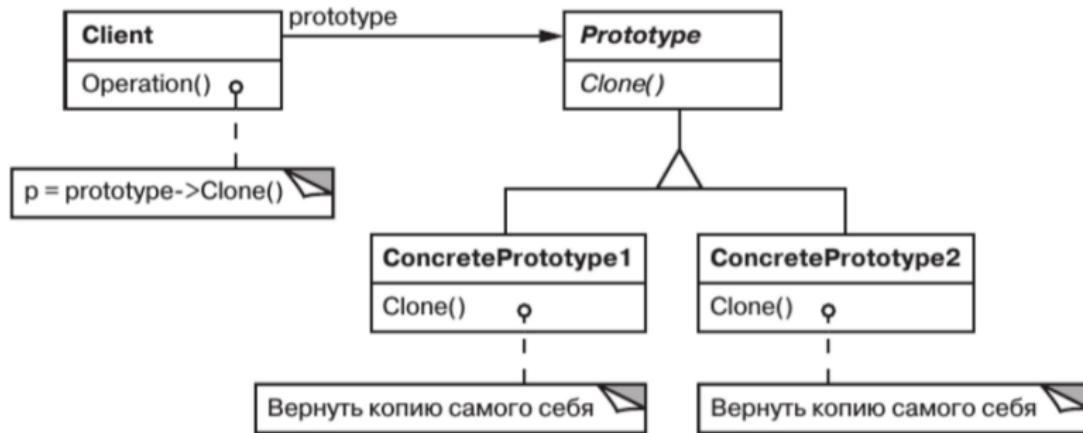
- ▶ Применяется, когда объекты создавать сложно, но каждый объект нужен лишь ненадолго
- ▶ Желательно, чтобы на поддержание объектов в пуле не требовалось много ресурсов, либо объектов в пуле было мало
 - ▶ Например, создать 50000 сетевых соединений “заранее” может быть плохой идеей
- ▶ Следует применять 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

“Прототип”, мотивация



Патерн “Прототип”

Prototype



“Прототип”, детали реализации

- ▶ Паттерн интересен только для языков, где мало runtime-информации о типе (C++)
- ▶ Реестр прототипов, обычно ассоциативное хранилище
- ▶ Операция Clone
 - ▶ Глубокое и мелкое копирование
 - ▶ В случае, если могут быть круговые ссылки
 - ▶ Сериализовать/десериализовать объект (но помнить про идентичность)
- ▶ Инициализация клона
 - ▶ Передавать параметры в Clone — плохая идея

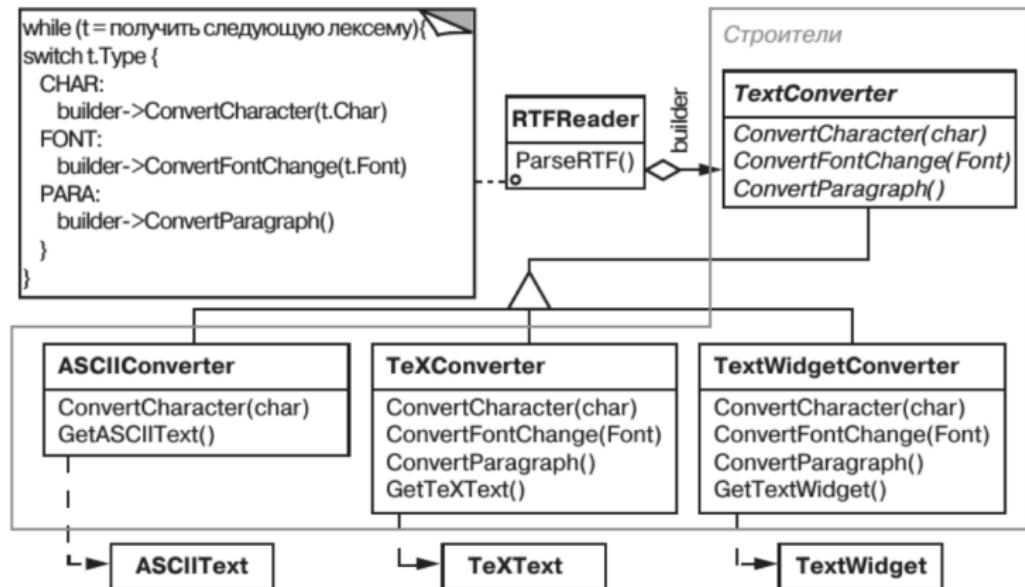
Лекция 8: Поведенческие шаблоны

Юрий Литвинов
yuriii.litvinov@gmail.com

22.10.2020г

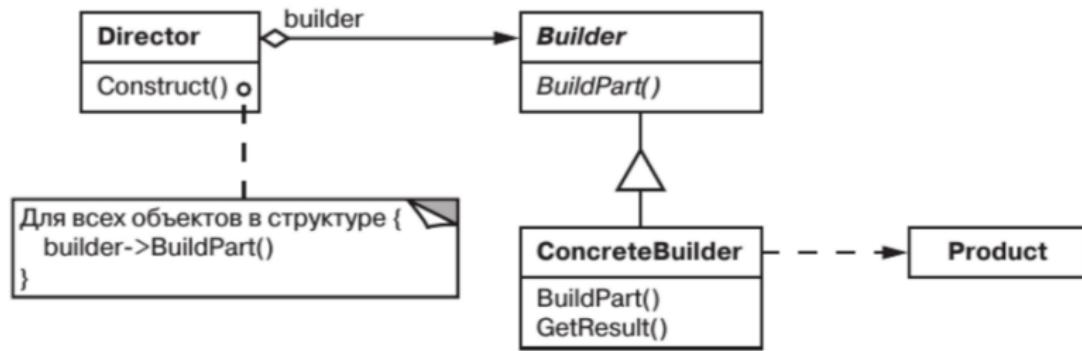
“Строитель”, мотивация

Конвертер текста

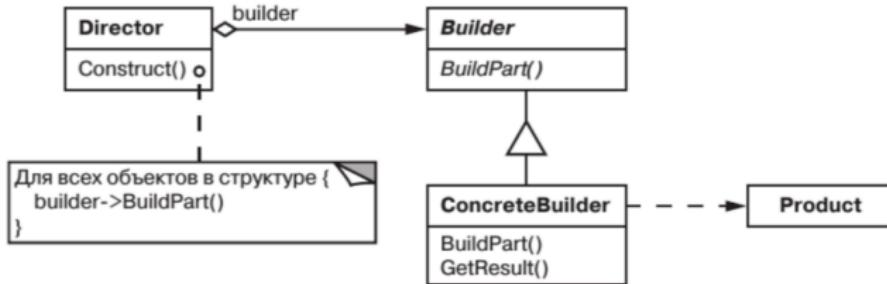


Патерн “Строитель”

Builder



“Строитель” (Builder), детали реализации



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

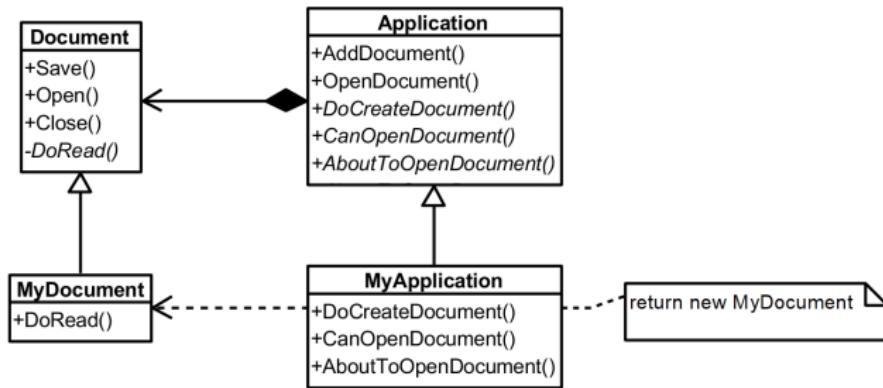
“Строитель”, примеры

- ▶ `StringBuilder`
- ▶ `Guava`, подсистема работы с графами

```
MutableNetwork<Webpage, Link> webSnapshot =
```

```
    NetworkBuilder.directed()  
        .allowsParallelEdges(true)  
        .nodeOrder(ElementOrder.natural())  
        .expectedNodeCount(100000)  
        .expectedEdgeCount(1000000)  
        .build();
```

Паттерн “Шаблонный метод”, мотивация



- ▶ Алгоритм, общий для всех потомков
- ▶ Детали реализации операций — в потомках
- ▶ Задание точек расширения

Шаблонный метод, пример

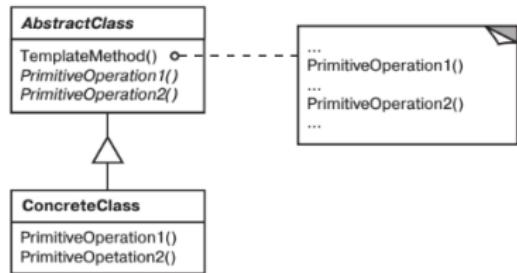
```
void Application::OpenDocument (const char* name) {  
    if (!CanOpenDocument(name)) {  
        return;  
    }
```

```
    Document* doc = DoCreateDocument();
```

```
    if (doc) {  
        _docs->AddDocument(doc);  
        AboutToOpenDocument(doc);  
        doc->Open();  
        doc->DoRead();  
    }  
}
```

"Шаблонный метод" (Template Method), детали реализации

- ▶ Сам шаблонный метод, как правило, невиртуальный
- ▶ Лучше использовать соглашения об именовании, например, называть операции с Do
- ▶ Примитивные операции могут быть виртуальными или чисто виртуальными
 - ▶ Лучше их делать protected
 - ▶ Чем их меньше, тем лучше

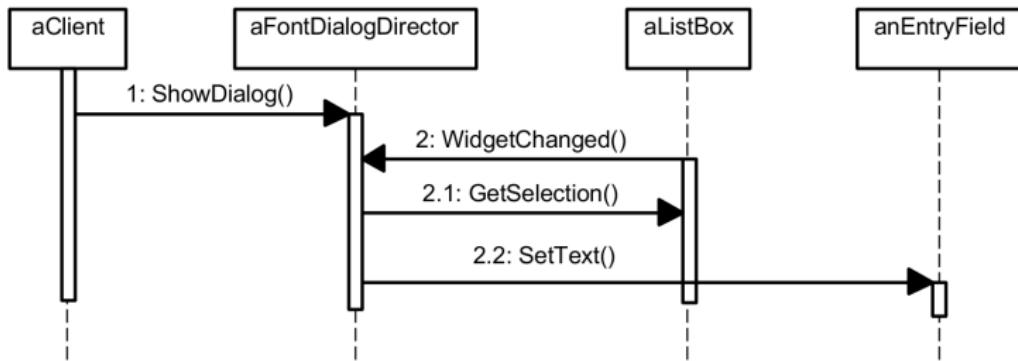
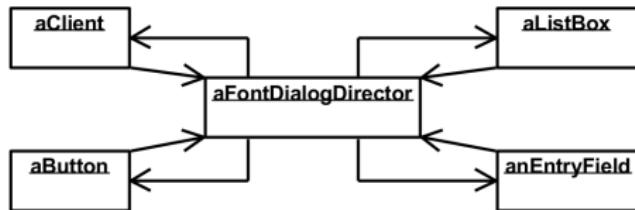


“Посредник” (Mediator), мотивация

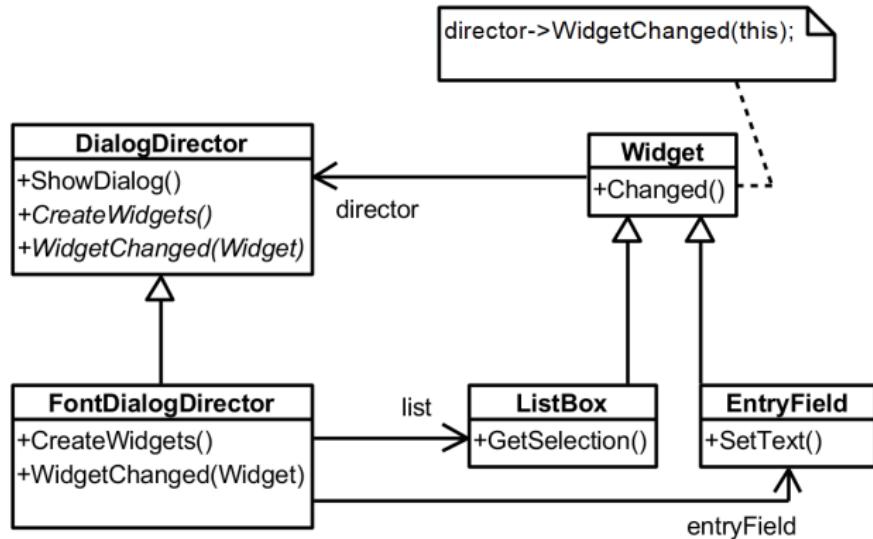
- ▶ Большое количество связей между объектами
- ▶ Объекты знают слишком много
- ▶ Снижается переиспользуемость



Решение: централизация

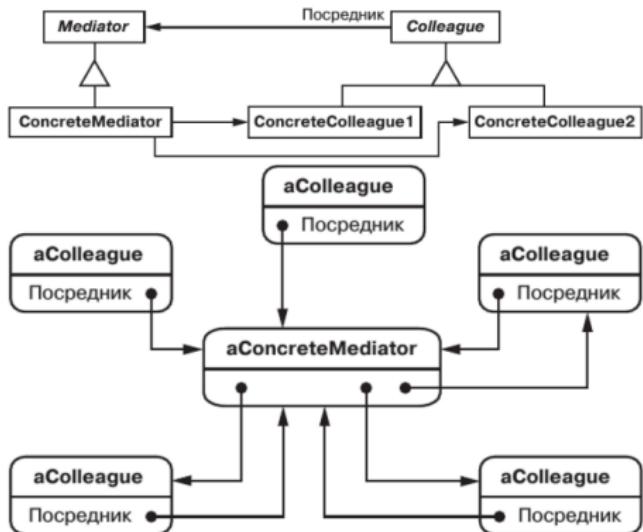


Что получилось



“Посредник” (Mediator), детали реализации

- ▶ Абстрактный класс “Mediator” часто не нужен
- ▶ Паттерн “Наблюдатель”: медиатор подписывается на события в коллегах
- ▶ Наоборот: коллеги вызывают методы медиатора



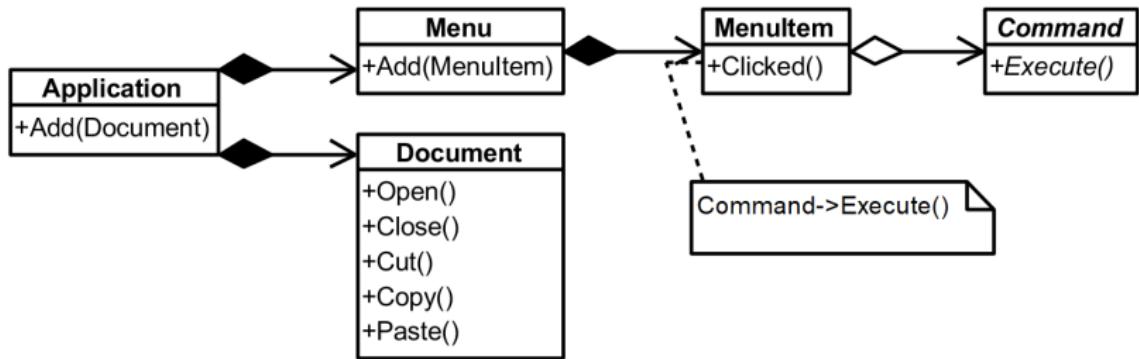
Посредник, достоинства и недостатки

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

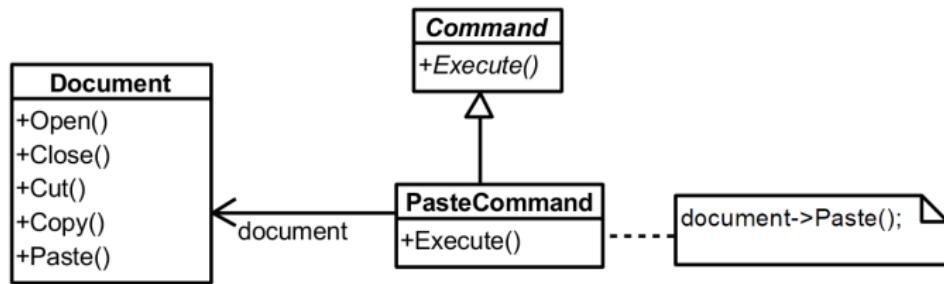
Паттерн “Команда”, мотивация

- ▶ Хотим отделить инициацию запроса от его исполнения
- ▶ Хотим, чтобы тот, кто “активирует” запрос, не знал, как он исполняется
- ▶ При этом хотим, чтобы тот, кто знает, когда исполнится запрос, не знал, когда он будет активирован
- ▶ Но зачем?
 - ▶ Команды меню приложения
 - ▶ Палитры инструментов
 - ▶ ...
- ▶ “Просто вызвать действие” не получится, вызов функции жёстко свяжет инициатора и исполнителя

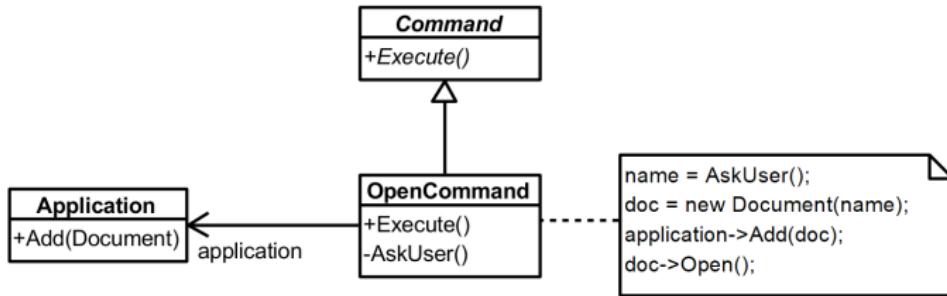
Решение: обернём действие в объект



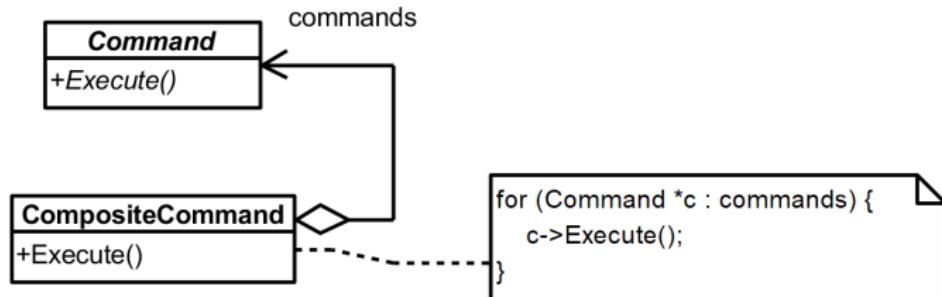
Команда вставки



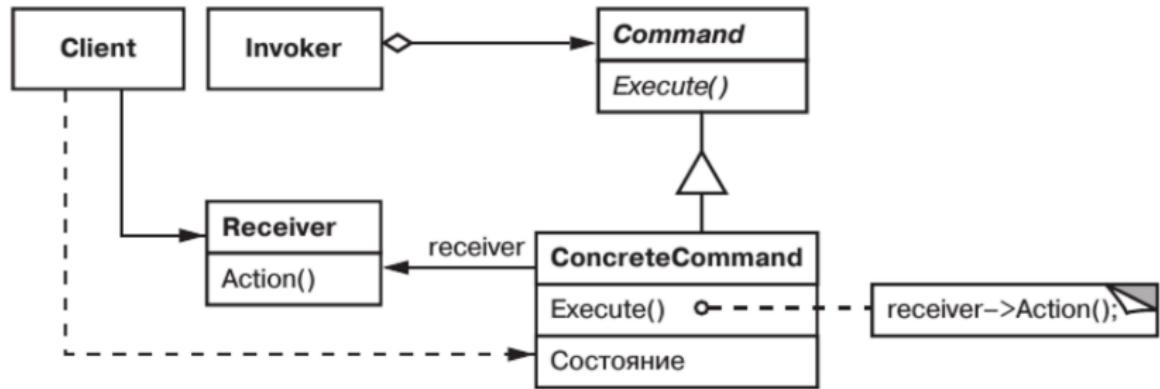
Команда открытия документа



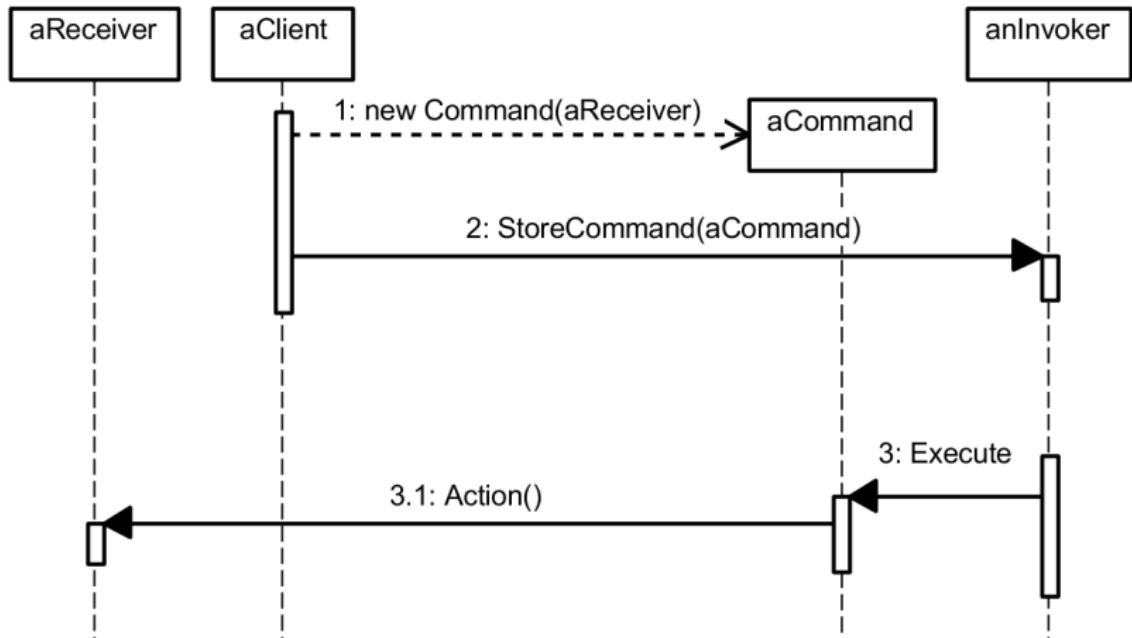
Составная команда



Паттерн "Команда"



Взаимодействие объектов



Команда, применимость

- ▶ Параметризовать объекты выполняемым действием
- ▶ Определять, ставить в очередь и выполнять запросы в разное время
- ▶ Поддержать отмену операций
- ▶ Структурировать систему на основе высокоуровневых операций, построенных из примитивных
- ▶ Поддержать протоколирование изменений

“Команда” (Command), детали реализации

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

"Команда", пример

- ▶ Qt, класс QAction:

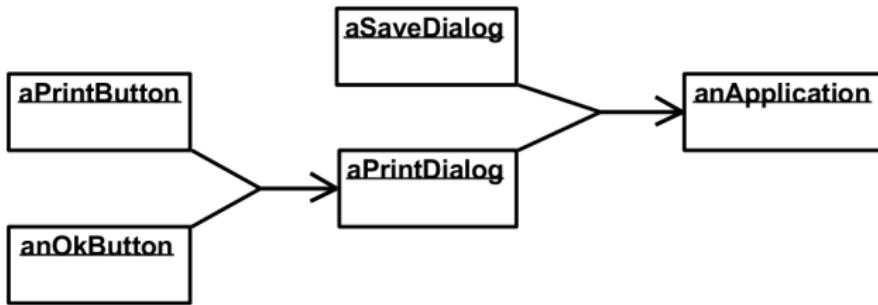
```
const QIcon openIcon = QIcon(":/images/open.png");
QAction *openAct = new QAction(openIcon, tr("&Open..."), this);
```

```
openAct->setShortcuts(QKeySequence::Open);
openAct->setStatusTip(tr("Open an existing file"));
```

```
connect(openAct, &QAction::triggered, this, &MainWindow::open);
```

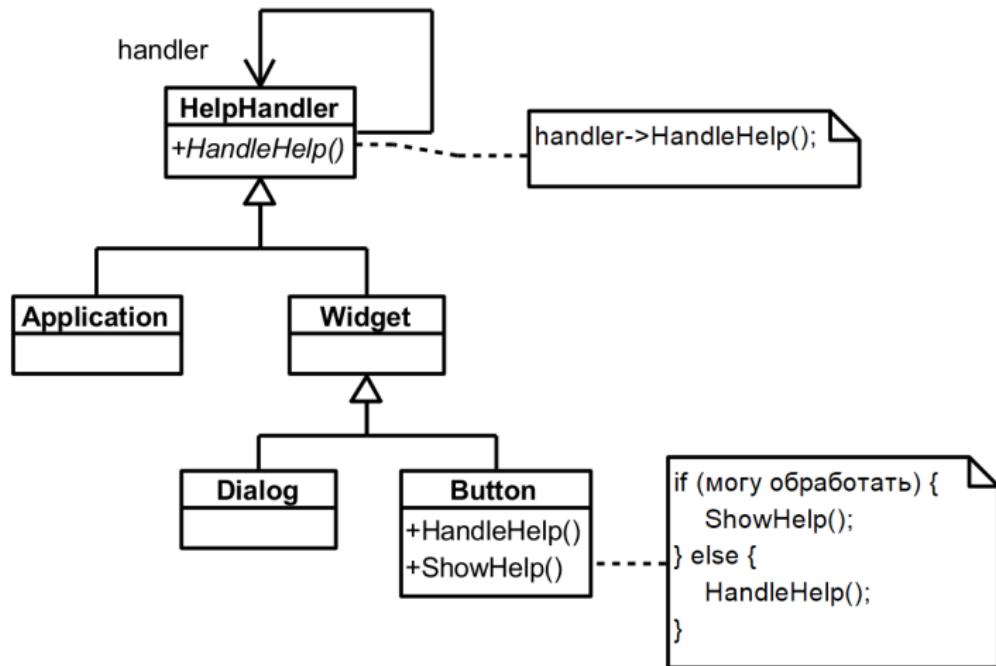
```
fileMenu->addAction(openAct);
fileToolBar->addAction(openAct);
```

Паттерн “Цепочка ответственности”, мотивация



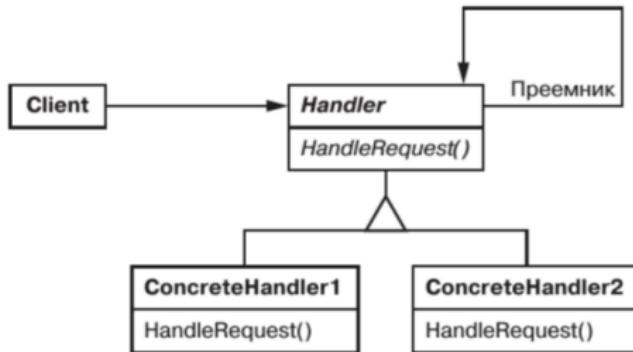
- ▶ Организация контекстной справки
- ▶ Если у элемента справки нет, запрос передаётся контейнеру
- ▶ Заранее неизвестно, кто в итоге обработает запрос

Как это выглядит на диаграмме классов



“Цепочка ответственности” (Chain of Responsibility), детали реализации

- ▶ Не обязательно реализовывать связи в цепочке специально
 - ▶ На самом деле, чаще используются существующие связи
- ▶ По умолчанию в Handler передавать запрос дальше (если ссылки на преемника всё-таки есть)
- ▶ Если возможных запросов несколько, их надо как-то различать
 - ▶ Явно вызывать методы — нерасширяемо
 - ▶ Использовать объекты-запросы



“Цепочка ответственности”, плюсы и минусы

- ▶ Ослабление связности
- ▶ Дополнительная гибкость при распределении обязанностей
- ▶ Получение не гарантировано

Когда использовать:

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

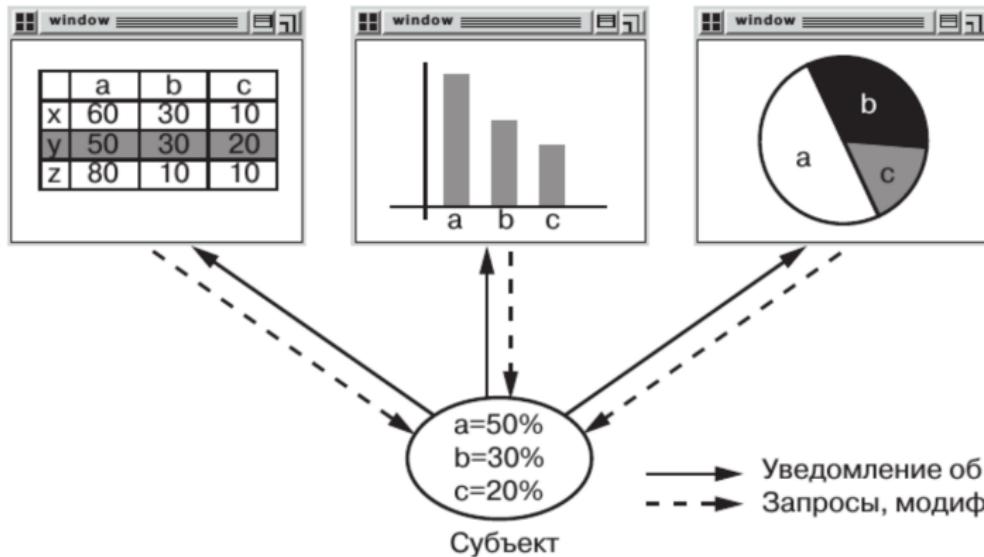
“Цепочка ответственности”, примеры

- ▶ Распространение исключений
- ▶ Распространение событий в оконных библиотеках:

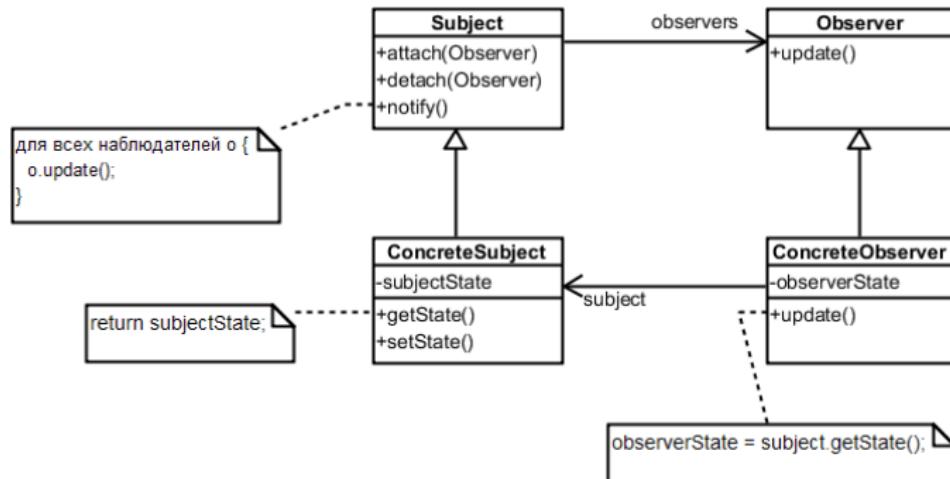
```
void MyCheckBox::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        // handle left mouse button here
    } else {
        // pass on other buttons to base class
        QCheckBox::mousePressEvent(event);
    }
}
```

Паттерн “Наблюдатель”, мотивация

Наблюдатели



Паттерн “Наблюдатель”



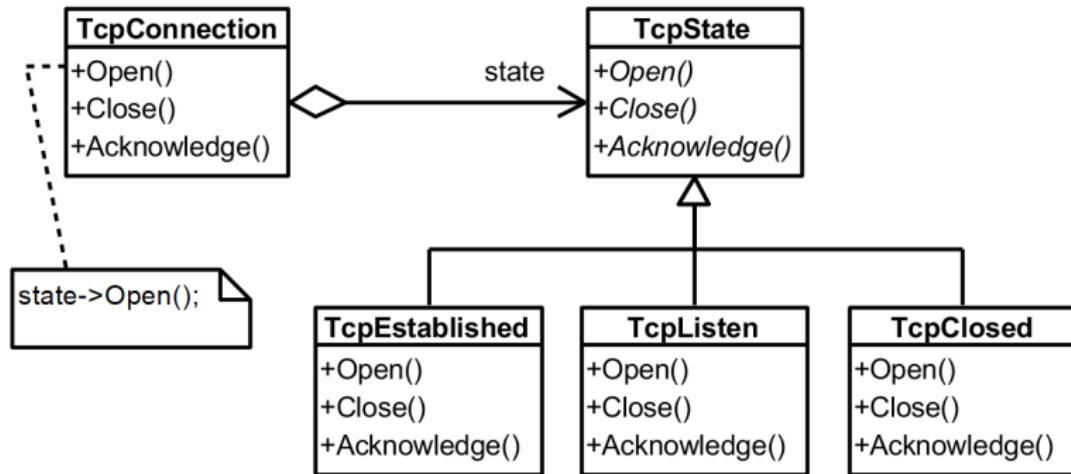
“Наблюдатель” (Observer), детали реализации

- ▶ Во многих языках поддержан “из коробки” (через механизм событий)
- ▶ Могут использоваться хеш-таблицы для отображения субъектов и наблюдателей
 - ▶ Так делает WPF в .NET, есть даже языковая поддержка в C#
- ▶ Необходимость идентифицировать субъект
- ▶ Кто инициирует нотификацию
 - ▶ Операции, модифицирующие субъект
 - ▶ Клиент, после серии модификаций субъекта

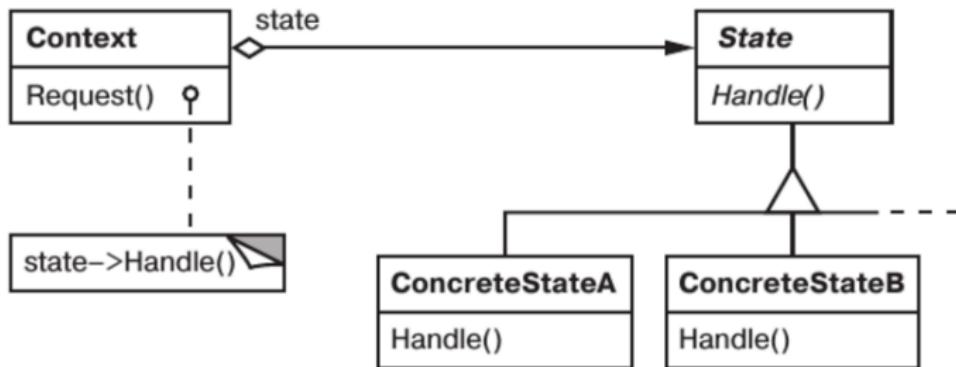
“Наблюдатель” (Observer), детали реализации (2)

- ▶ Ссылки на субъектов и наблюдателей
 - ▶ Простой способ организовать утечку памяти в C# или грохнуть программу в C++
- ▶ Консистентность субъекта при отправке нотификации
 - ▶ Очевидно, но легко нарушить, вызвав метод предка в потомке
 - ▶ “Шаблонный метод”
 - ▶ Документировать, кто когда какие события бросает
- ▶ Передача сути изменений — pull vs push
- ▶ Фильтрация по типам событий
- ▶ Менеджер изменений (“Посредник”)

Паттерн “Состояние”, мотивация



Паттерн “Состояние”



“Состояние” (State), детали реализации

- ▶ Переходы между состояниями — в Context или в State?
- ▶ Таблица переходов
 - ▶ Трудно добавить действия по переходу
- ▶ Создание и уничтожение состояний
 - ▶ Создать раз и навсегда
 - ▶ Создавать и удалять при переходах

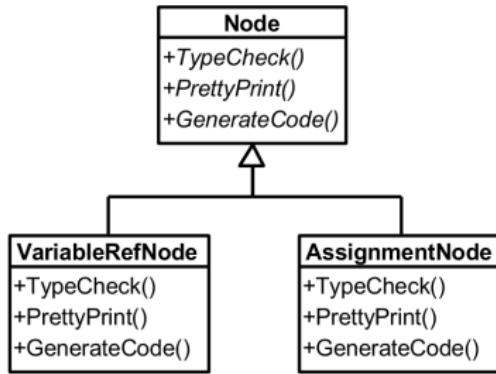
“Состояние” результаты

- ▶ Локализует зависящее от состояния поведение
- ▶ Делает явными переходы между состояниями
- ▶ Объекты состояния можно разделять

Когда применять:

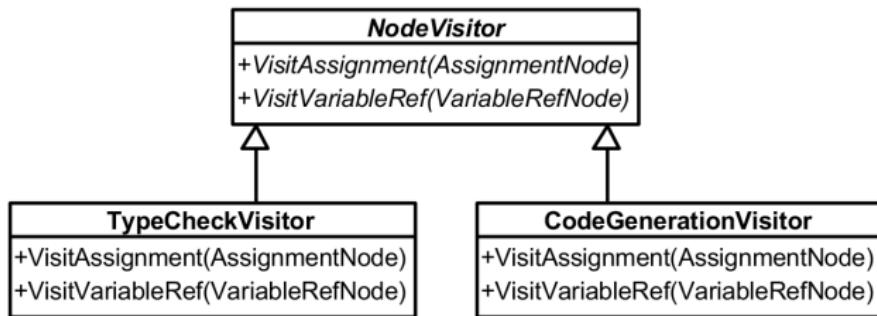
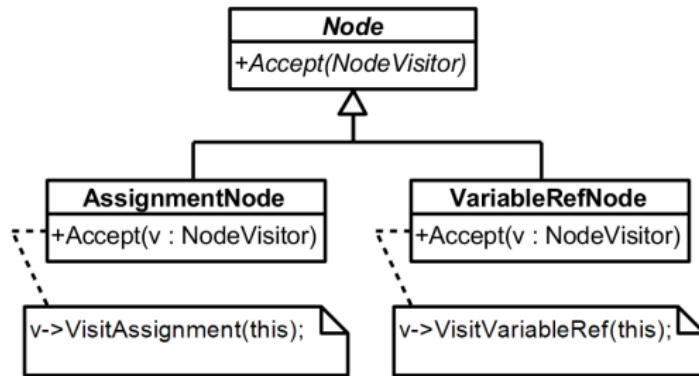
- ▶ Поведение объекта зависит от его состояния и должно изменяться во время выполнения
- ▶ Обилие условных операторов, в которых выбор ветви зависит от состояния

Паттерн “Посетитель”, мотивация

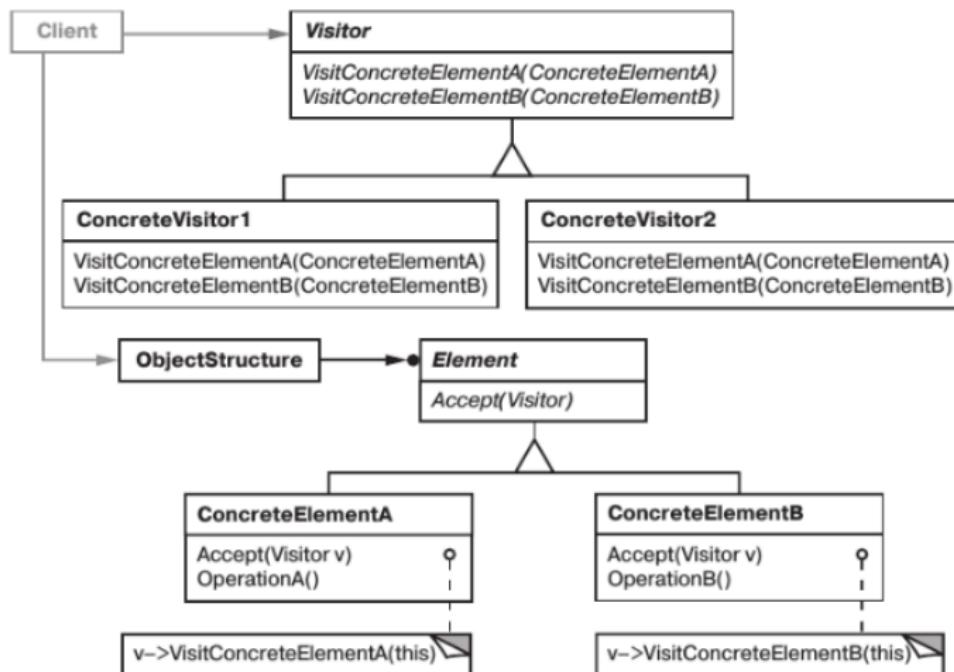


- ▶ Синтаксическое дерево
- ▶ Много разных типов узлов
- ▶ Много разных операций, которые над ними можно выполнять

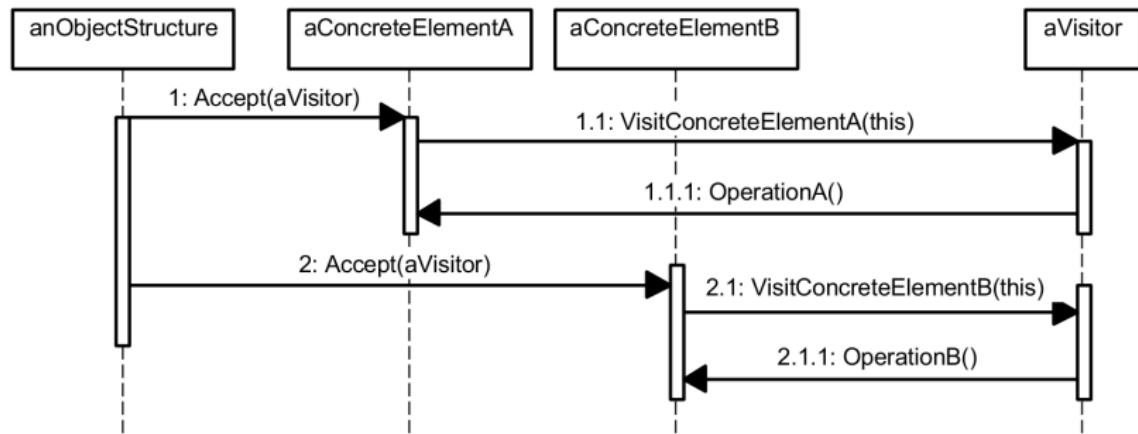
Паттерн “Посетитель”, решение



Паттерн “Посетитель”



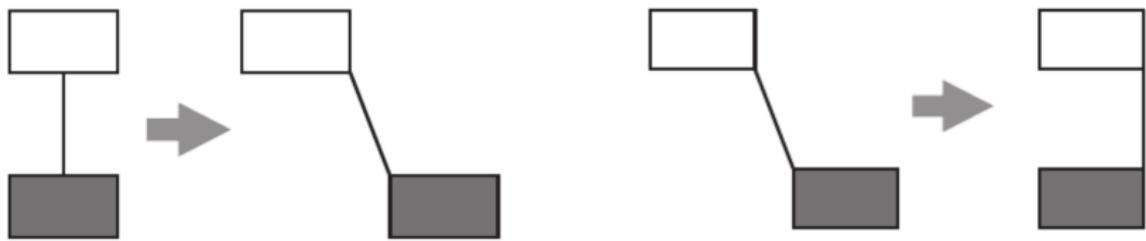
Двойная диспетчеризация



"Посетитель" (Visitor), детали реализации

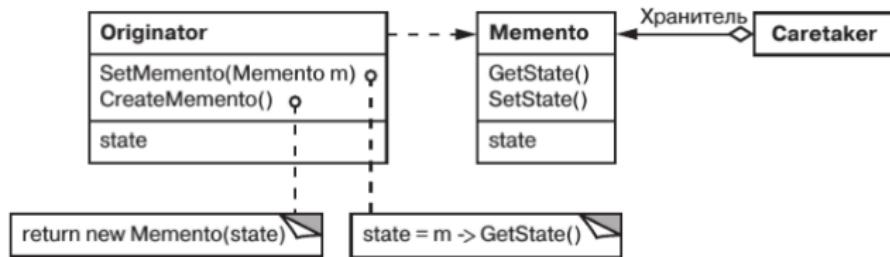
- ▶ Использовать перегрузку методов Visit(...)
- ▶ Чаще всего сама коллекция отвечает за обход, но может быть итератор
- ▶ Может даже сам Visitor, если обход зависит от результата операции
- ▶ Аккумулирование состояния
- ▶ Несколько нарушает инкапсуляцию
- ▶ Просто добавлять новые операции, но сложно добавлять новые классы

Паттерн “Хранитель”, мотивация



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

Паттерн “Хранитель”

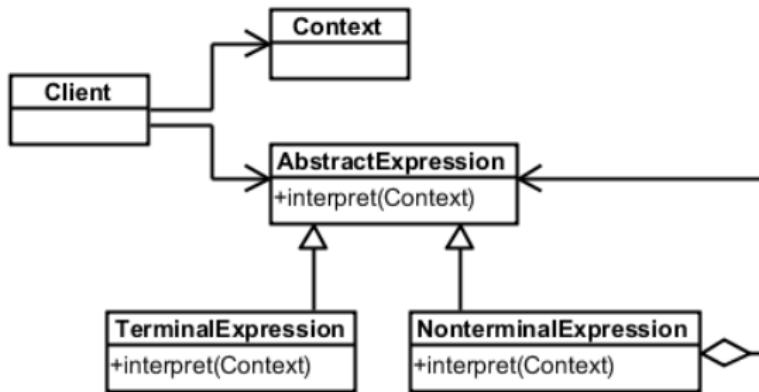


“Хранитель” (Memento), детали реализации

- ▶ Два интерфейса: “широкий” для хозяев и “узкий” для остальных объектов
 - ▶ Требуется языковая поддержка
- ▶ Можно хранить только дельты состояний

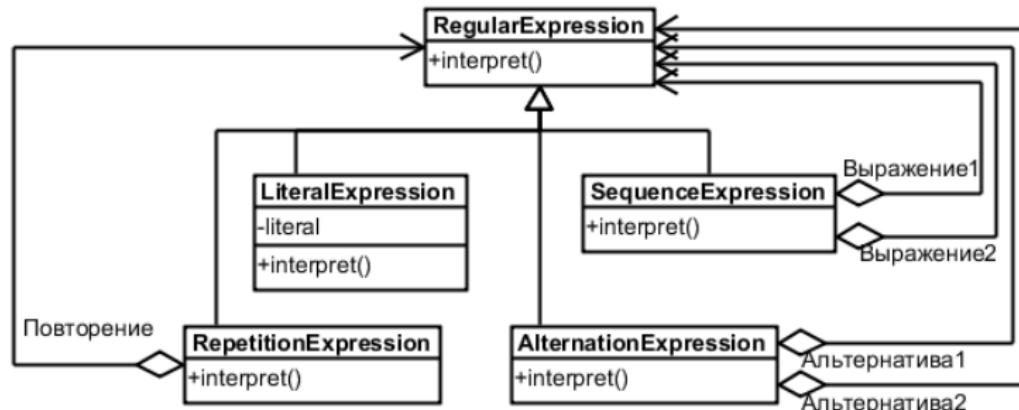
“Интерпретатор” (Interpreter)

Определяет представление грамматики и интерпретатор для заданного языка.



- ▶ Грамматика должна быть проста (иначе лучше “Visitor”)
- ▶ Эффективность не критична

“Интерпретатор”, пример



“Интерпретатор”, детали реализации

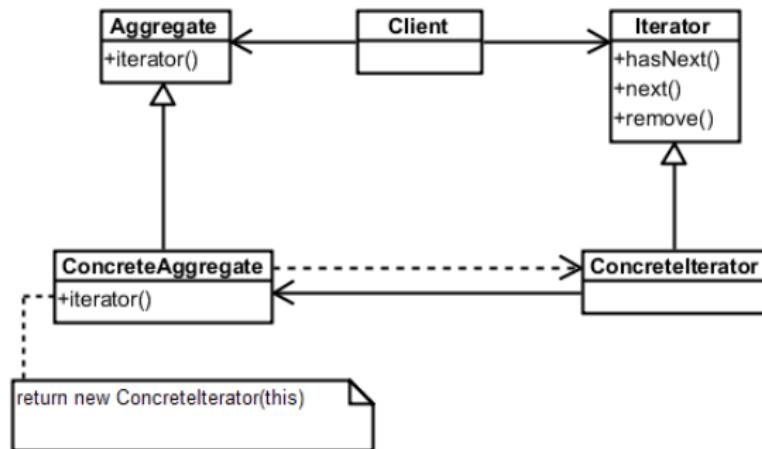
10-е правило Гринспена:

Любая достаточно сложная программа на Си или Фортране содержит заново написанную, неспецифицированную, глючную и медленную реализацию половины языка Common Lisp

- ▶ Построение дерева — отдельная задача
- ▶ Несколько разных операций над деревом — лучше “Visitor”
- ▶ Можно использовать “Приспособленец” для разделения терминальных символов

“Итератор” (Iterator)

Инкапсулирует способ обхода коллекции.



- ▶ Разные итераторы для разных способов обхода
- ▶ Можно обходить не только коллекции

“Итератор”, примеры

- ▶ Java-стиль:

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

- ▶ .NET-стиль:

```
public interface IEnumerator<T>  
{  
    bool MoveNext();  
    T Current { get; }  
    void Reset();  
}
```

“Итератор”, детали реализации (1)

- ▶ Внешние итераторы

foreach (Thing t **in** collection)

```
{  
    Console.WriteLine(t);  
}
```

- ▶ Внутренние итераторы

```
collection.ToList().ForEach(t => Console.WriteLine(t));
```

“Итератор”, детали реализации (2)

- ▶ Итераторы и курсоры
- ▶ Устойчивые и неустойчивые итераторы
 - ▶ Паттерн “Наблюдатель”
 - ▶ Даже обнаружение модификации коллекции может быть непросто
- ▶ Дополнительные операции

Лекция 9: Антипаттерны

То, как делать не надо

Юрий Литвинов
yurii.litvinov@gmail.com

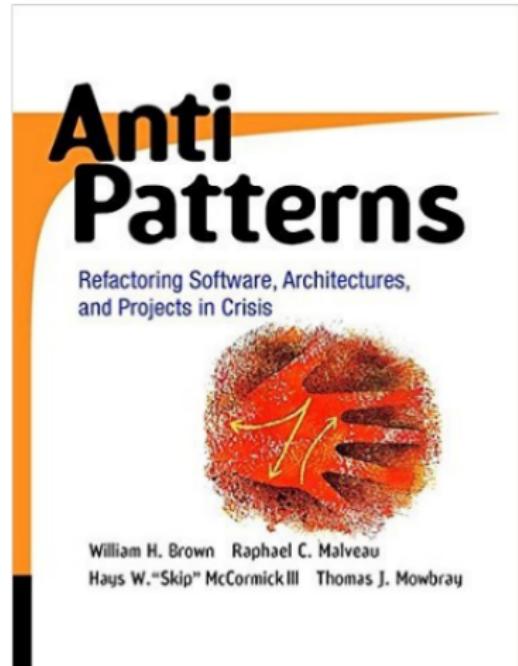
29.10.2020г

Антипаттерны — что это и зачем?

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

Книжка

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



Семь причин провала проектов

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

Circular dependency

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

Sequential coupling

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

Call super

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

Yo-yo problem

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

Busy waiting

- ▶ Ожидание наступления некоторого события в бесконечном цикле с неблокирующими вызовами проверки наступления события
 - ▶ Ещё использование циклов для задержек
- ▶ Не всегда плохо, может быть валидным решением на встроенных устройствах
 - ▶ 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

Error hiding

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

Magic numbers, Magic strings

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

God Object (The Blob)

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

- ▶ Один класс управляет всем процессом вычислений, остальные в основном предоставляют ему данные
 - ▶ Привычка к структурному программированию, разделение данных и кода
 - ▶ Постепенная эволюция proof-of-concept без рефакторинга (лень, спешка)
 - ▶ Архитектурная ошибка, неправильное разделение обязанностей
- ▶ Хаотичное объединение различных 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 А и В лежат внутри объекта G, может так случиться, что А может быть в В, а В — 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

Lava Flow

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

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

Lava Flow, что делать

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

Functional Decomposition

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

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

Functional Decomposition, что делать

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

Poltergeists

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

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

Golden Hammer

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

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

Golden Hammer, как бороться

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

Design smells

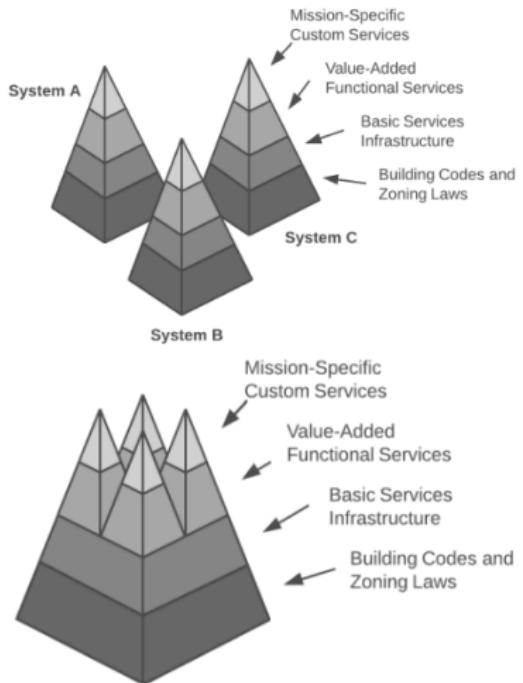
- ▶ **Missing abstraction** — использование примитивных или слишком общих типов данных вместо своих классов
- ▶ **Multifaceted abstraction** — нарушение единственности ответственности
- ▶ **Duplicate abstraction** — две абстракции имеют одно предназначение
- ▶ **Deficient encapsulation** — слишком широкая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

Design smells (2)

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

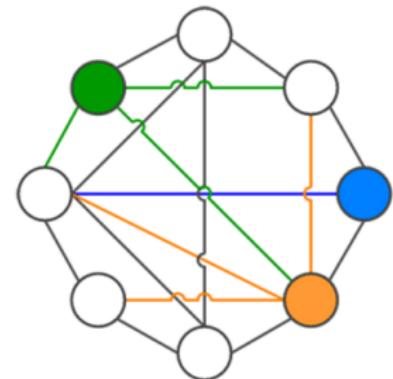
Stovepipe Enterprise (Island of Automation)

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



Stovepipe System

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



Vendor Lock-In

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

Architecture By Implication

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

Design By Committee

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

Лекция 10: Архитектурные стили

Юрий Литвинов

yurii.litvinov@gmail.com

05.11.2020г

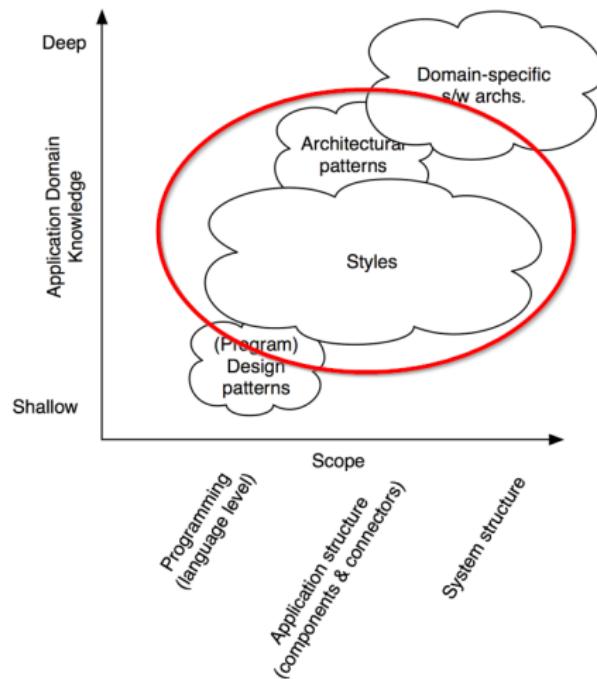
Архитектурные шаблоны и стили

Архитектурный стиль — набор решений, которые

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

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

Архитектурные шаблоны и стили, классификация



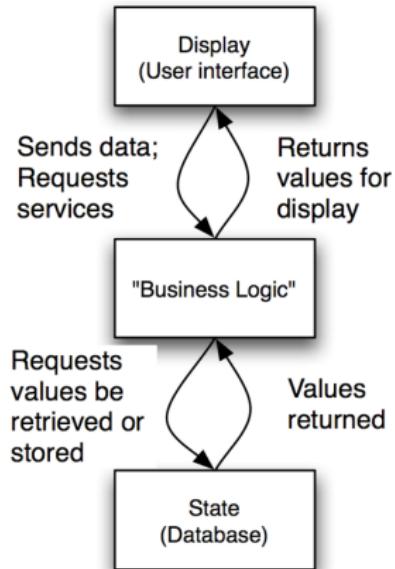
© N. Medvidovic

Пример: трёхзвенная архитектура

State-Logic-Display

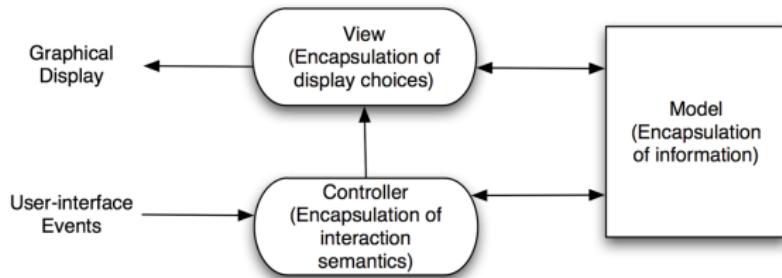
Примеры применения

- ▶ Бизнес-приложения
- ▶ Многопользовательские игры
- ▶ Веб-приложения



© N. Medvidovic

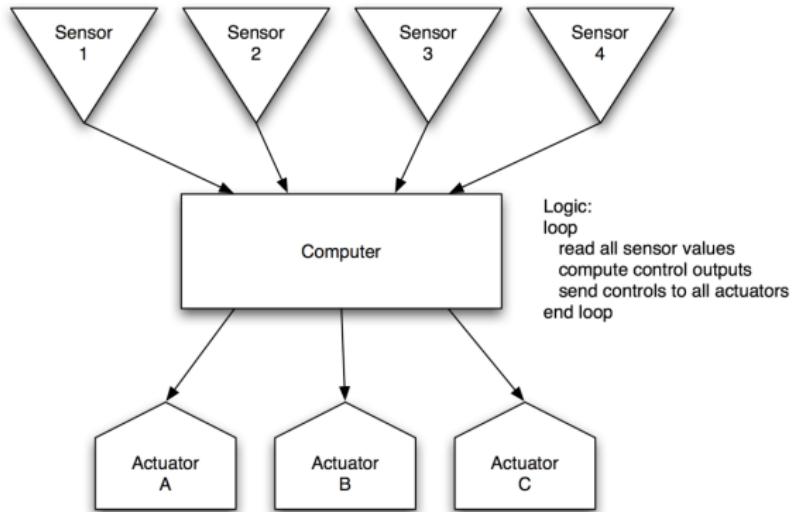
Пример: Model-View-Controller



© N. Medvidovic

- ▶ Разделяет данные, представление и взаимодействие с пользователем
- ▶ Если в модели что-то меняется, она оповещает представление (представления)
- ▶ Через контроллер проходит всё взаимодействие с пользователем
 - ▶ Естественное место для паттерна “Команда” и Undo/Redo

Пример: Sense-Compute-Control



© N. Medvidovic

- ▶ Применяется во встроенных системах и робототехнике

Архитектурные стили

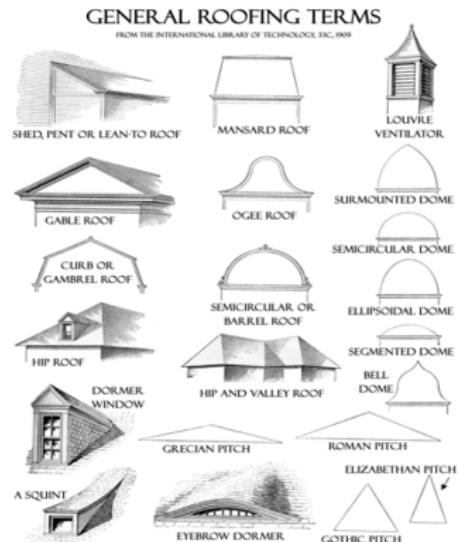
- ▶ Именованная коллекция архитектурных решений
- ▶ Менее узкоспециализированные, чем архитектурные паттерны



© N. Medvidovic

Архитектурные стили

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



© N. Medvidovic

Преимущества использования стилей

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

Основные характеристики стилей

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

Игра “Посадка на луну”

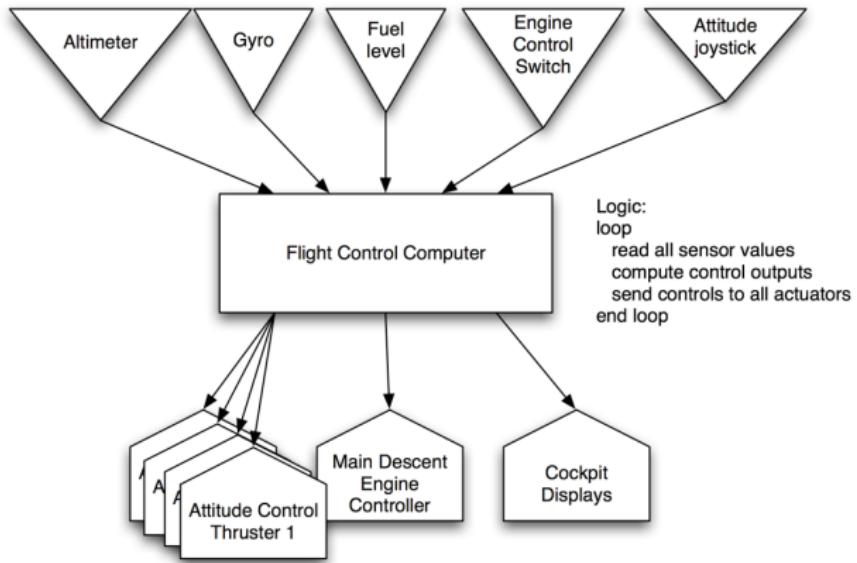
Lunar Lander

- ▶ Игрок управляет двигателем спускаемого аппарата
- ▶ Топливо ограничено
- ▶ Заданы начальная высота и скорость
- ▶ Победа засчитывается, если скорость при касании грунта меньше заданной
- ▶ Продвинутая версия позволяет управлять горизонтальным движением



© N. Medvidovic

Sense-Compute-Control-реализация

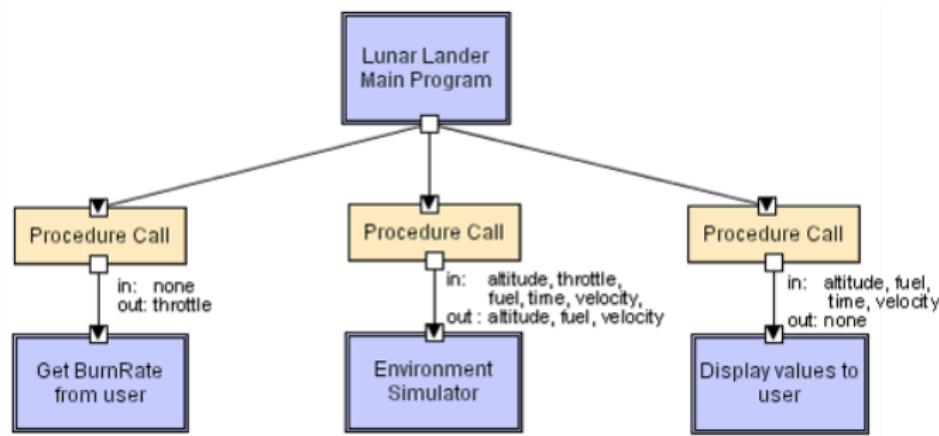


© N. Medvidovic

Некоторые известные стили

- ▶ “Традиционные”, связанные с языком
 - ▶ Главная программа/подпрограммы
 - ▶ Объектно-ориентированный
- ▶ Уровневый стиль
 - ▶ Виртуальные машины
 - ▶ Клиент-сервер
- ▶ Стили, ориентированные на поток данных
 - ▶ Пакетное исполнение
 - ▶ Каналы и фильтры
- ▶ Peer-to-peer
- ▶ Общая память
 - ▶ Blackboard
 - ▶ Ориентированные на правила
- ▶ Интерпретаторы
 - ▶ Интерпретатор
 - ▶ Мобильный код
- ▶ Неявный вызов
 - ▶ Событийно-ориентированный
 - ▶ Издатель-подписчик
- ▶ “Производные” стили
 - ▶ Распределённые объекты
 - ▶ REST
 - ▶ C2

Главная программа/подпрограммы

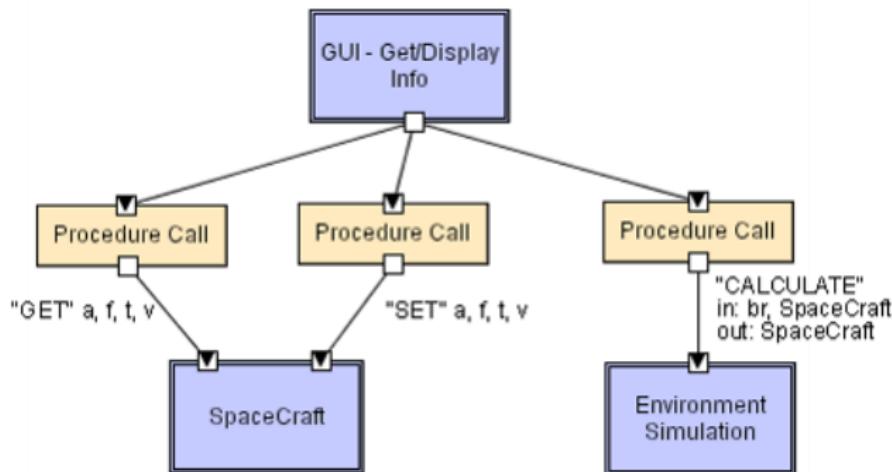


© N. Medvidovic

Объектно-ориентированный стиль

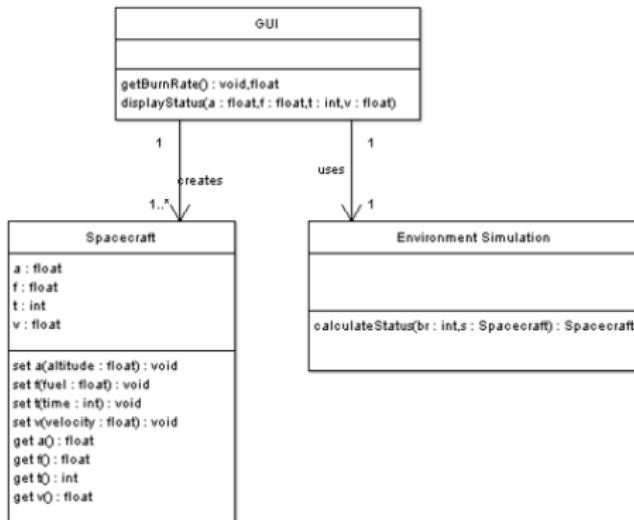
- ▶ Компоненты — объекты
- ▶ Соединители — сообщения и вызовы методов
- ▶ Инварианты:
 - ▶ Объекты отвечают за своё внутреннее состояние
 - ▶ Реализация скрыта от других объектов
- ▶ Преимущества:
 - ▶ Декомпозиция системы в набор взаимодействующих агентов
 - ▶ Внутреннее представление объектов можно менять независимо
 - ▶ Близко к предметной области
- ▶ Недостатки:
 - ▶ Побочные эффекты при вызове методов
 - ▶ Объекты вынуждены знать обо всех, от кого зависят

Объектно-ориентированный стиль, Lunar Lander



© N. Medvidovic

Или то же на UML

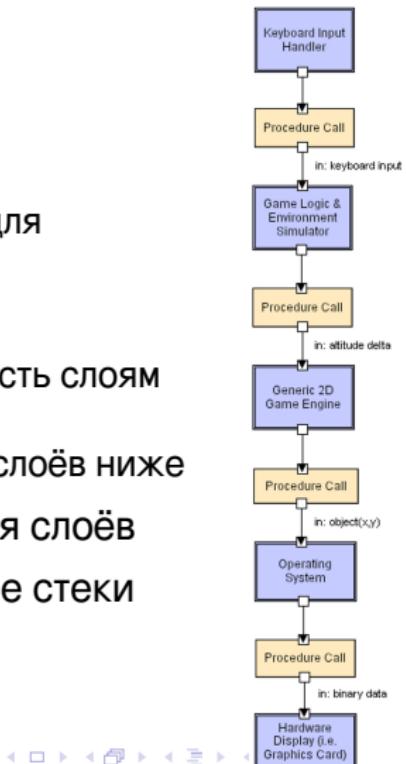


© N. Medvidovic

Слоистый стиль

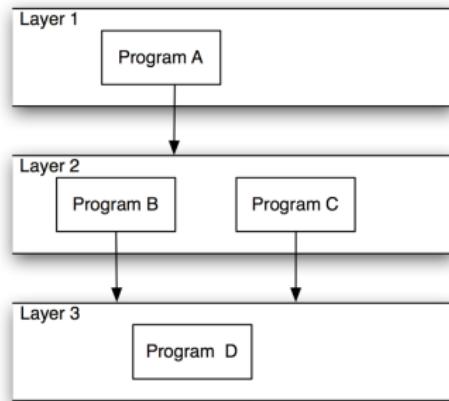
Layered style

- ▶ Иерархическая организация системы
 - ▶ “Многоуровневый клиент-сервер”
 - ▶ Каждый слой предоставляет интерфейс для использования слоями выше
- ▶ Каждый слой работает как:
 - ▶ Сервер — предоставляет функциональность слоям выше
 - ▶ Клиент — использует функциональность слоёв ниже
- ▶ Соединители — протоколы взаимодействия слоёв
- ▶ Пример — операционные системы, сетевые стеки протоколов



Слоистый стиль, подробности

- ▶ Преимущества:
 - ▶ Повышение уровня абстракции
 - ▶ Лёгкость в расширении
 - ▶ Изменения в каждом уровне затрагивают максимум два соседних
 - ▶ Возможны разные реализации уровня, если они удовлетворяют интерфейсу
- ▶ Недостатки:
 - ▶ Не всегда применим
 - ▶ Проблемы с производительностью

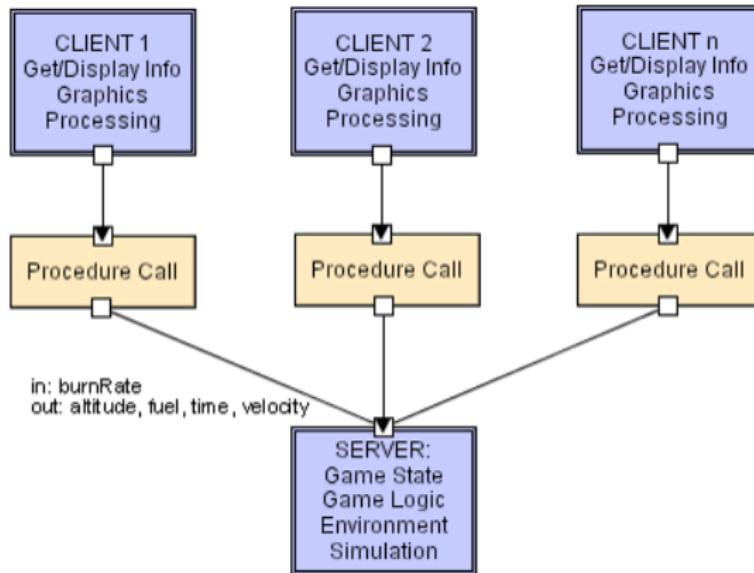


© N. Medvidovic

“Клиент-сервер”

- ▶ Компоненты — клиенты и серверы
- ▶ Серверы не знают ничего о клиентах, даже их количество
- ▶ Клиенты знают только про сервера и не могут общаться друг с другом
- ▶ Соединители — сетевые протоколы

“Клиент-сервер”, Lunar Lander

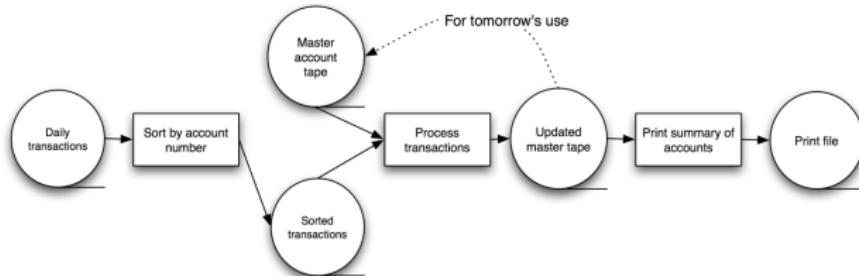


© N. Medvidovic

Пакетная обработка

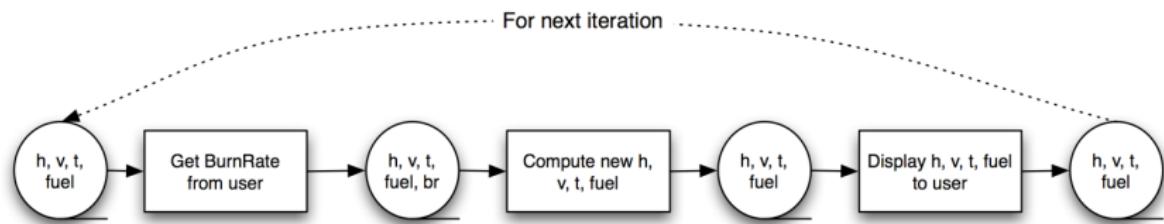
- ▶ Система строится как набор отдельных программ, выполняющихся последовательно
- ▶ Данные стандартным для ОС способом передаются от программы к программе
 - ▶ Pipes, named pipes, файлы
- ▶ Данные — в явном виде всё, необходимое для работы

Типичен для финансовых систем глубокой древности (“Прадедушка стилей”)



© N. Medvidovic

Пакетная обработка, Lunar Lander



© N. Medvidovic

Play-by-email?

Каналы и фильтры

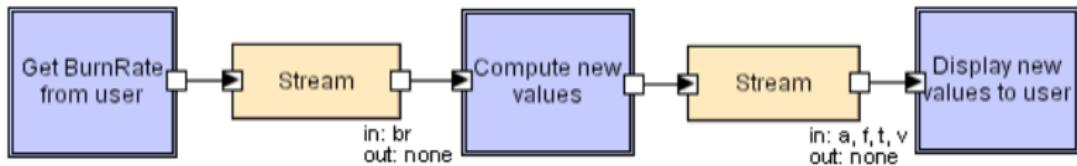
Pipes and filters

- ▶ Компоненты — это фильтры, преобразующие данные из входных каналов в данные в выходных каналах
- ▶ Соединители — каналы
- ▶ Инварианты:
 - ▶ Фильтры независимы (не имеют разделяемого состояния)
 - ▶ Фильтры не знают о фильтрах до или после них
- ▶ Вариации:
 - ▶ Конвейеры — линейные последовательности фильтров
 - ▶ Ограниченные каналы — где канал это очередь с ограниченным количеством элементов
 - ▶ Типизированные каналы — где каналы отличаются по типу передаваемых данных

Каналы и фильтры, подробности

- ▶ Преимущества:
 - ▶ Поведение системы — это просто последовательное применение поведений компонентов
 - ▶ Легко добавлять, заменять и переиспользовать фильтры
 - ▶ Любые два фильтра можно использовать вместе
 - ▶ Широкие возможности для анализа
 - ▶ Пропускная способность, задержки, deadlock-и
 - ▶ Широкие возможности для параллелизма
- ▶ Недостатки:
 - ▶ Последовательное исполнение
 - ▶ Проблемы с интерактивными приложениями
 - ▶ Пропускная способность определяется самым “узким” элементом

Каналы и фильтры, Lunar Lander

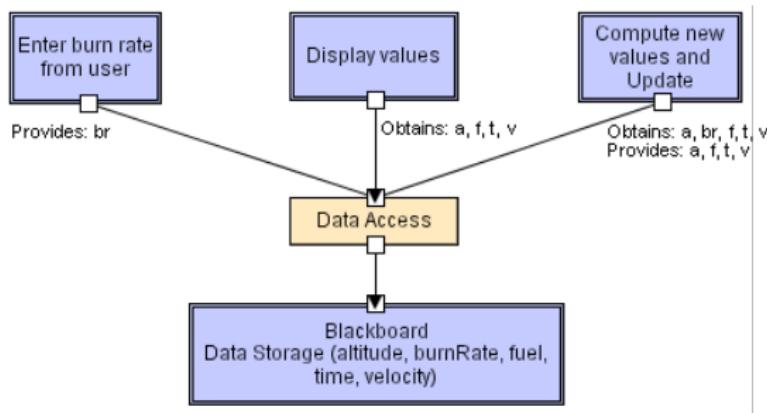


© N. Medvidovic

Blackboard

- ▶ Два типа компонентов:
 - ▶ Центральная структура данных — та самая “Blackboard”
 - ▶ Компоненты, работающие с blackboard
- ▶ Инварианты:
 - ▶ Управление системой осуществляется только через состояние доски
 - ▶ Компоненты не знают друг о друге и не имеют своего состояния
- ▶ Часто применяется в системах искусственного интеллекта

Blackboard, Lunar Lander



© N. Medvidovic

Стили с неявным вызовом

- ▶ Оповещение о событии вместо явного вызова метода
 - ▶ “Слушатели” могут подписаться на событие
 - ▶ Система при наступлении события сама вызывает все зарегистрированные методы слушателей
- ▶ Компоненты имеют два вида интерфейсов — методы и события
- ▶ Два типа соединителей:
 - ▶ Явный вызов метода
 - ▶ Неявный вызов по наступлению события
- ▶ Инварианты:
 - ▶ Те, кто производит события, не знают, кто и как на них отреагирует
 - ▶ Не делается никаких предположений о том, как событие будет обработано и будет ли вообще

Стили с неявным вызовом, преимущества и недостатки

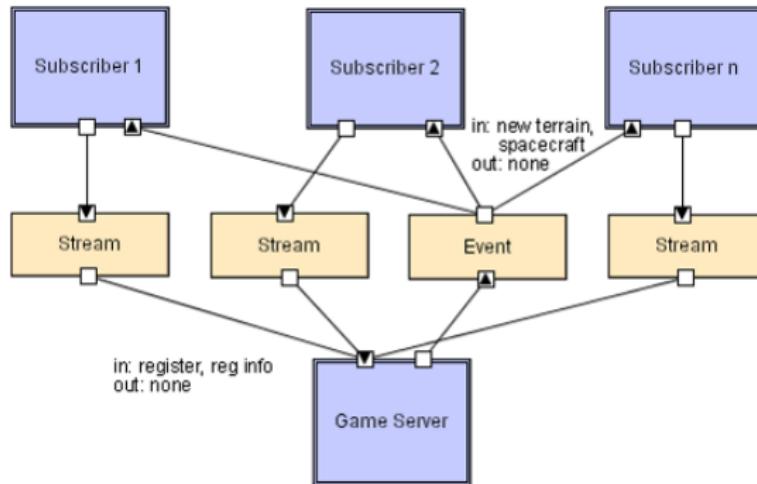
- ▶ Преимущества:
 - ▶ Переиспользование компонентов
 - ▶ Очень низкая связность между компонентами
 - ▶ Лёгкость в конфигурировании системы
 - ▶ Как во время компиляции, так и во время выполнения
- ▶ Недостатки:
 - ▶ Зачастую неинтуитивная структура системы
 - ▶ Компоненты не управляют последовательностью вычислений
 - ▶ Непонятно, кто отреагирует на запрос и в каком порядке придут ответы
 - ▶ Тяжело отлаживаться
 - ▶ Гонки даже в однопоточном приложении

Издатель-подписчик

Publish-subscribe

- ▶ Подписчики регистрируются, чтобы получать нужные им сообщения или данные. Издатели публикуют сообщения, синхронно или асинхронно.
- ▶ Компоненты: издатели, подписчики, “маршрутизаторы”
- ▶ Соединители: как правило, сетевые протоколы, часто механизм наподобие паттерна “Наблюдатель”
- ▶ Данные: подписки, нотификации, публикуемая информация
- ▶ Топология: подписчики подключаются к издателям напрямую, либо через посредников
- ▶ Преимущества: очень низкая связность между компонентами, при этом высокая эффективность распределения информации

Издатель-подписчик, Lunar Lander

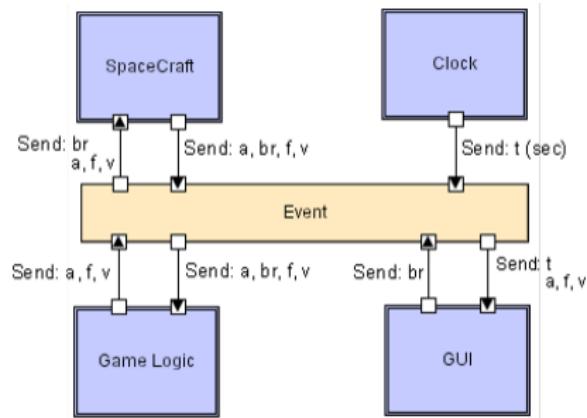


© N. Medvidovic

Событийно-ориентированный стиль

- ▶ Независимые компоненты посылают и принимают события, передаваемые по шинам
- ▶ Компоненты: независимые генераторы или потребители событий
- ▶ Соединители: шины событий (хотя бы одна)
- ▶ Данные: события и связанные с ними данные, посылаемые пошине
- ▶ Топология: компоненты общаются только с шинами событий, не друг с другом
- ▶ Варианты: push- и pull-режимы работы с шиной
- ▶ Преимущества: лёгкость масштабирования и добавления новой функциональности, эффективно для распределённых приложений

Событийно-ориентированный Lunar Lander

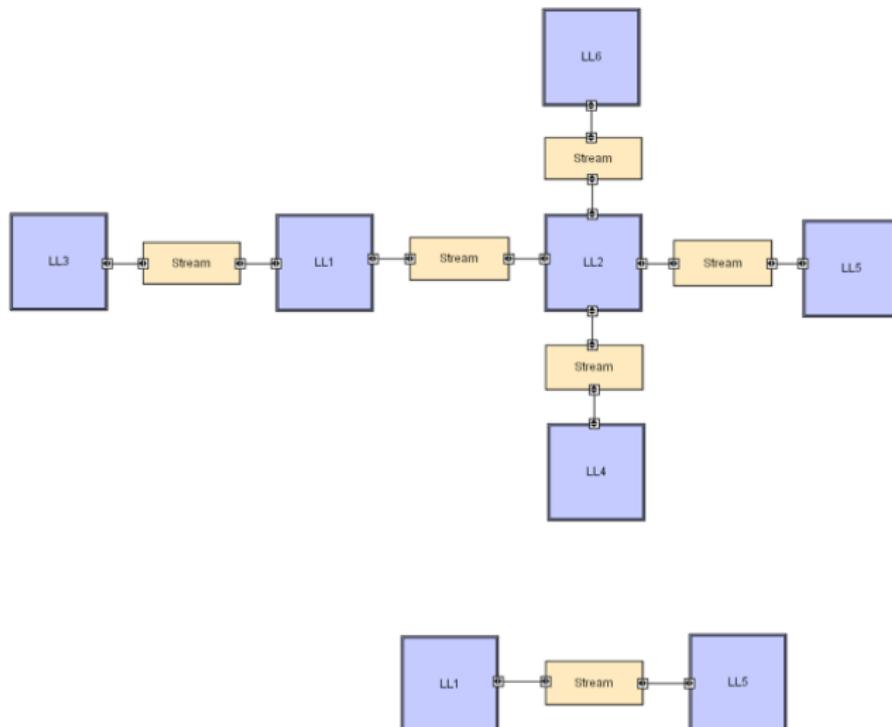


© N. Medvidovic

Peer-to-peer

- ▶ Состояние и поведение распределены между компонентами, которые могут выступать как клиенты и как серверы
- ▶ Компоненты: имеют своё состояние и свой поток управления
- ▶ Соединители: как правило, сетевые протоколы
- ▶ Элементы данных: сетевые сообщения
- ▶ Топология: сеть (возможно, с избыточными связями между компонентами), может динамически меняться
- ▶ Преимущества:
 - ▶ Хорош для распределённых вычислений
 - ▶ Устойчив к отказам
 - ▶ Если протокол взаимодействия позволяет, легко масштабируется

Peer-to-peer Lunar Lander



© N. Medvidovic

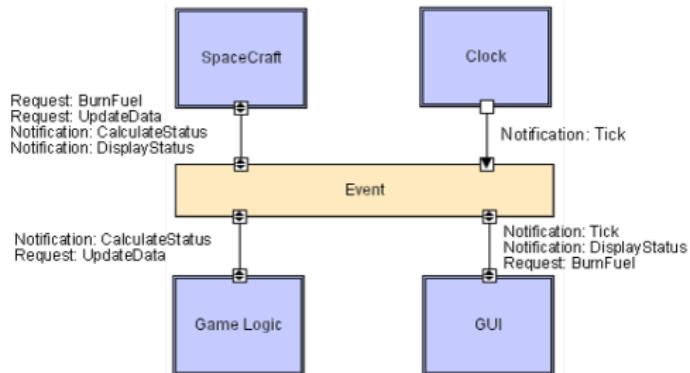
Гетерогенные стили

- ▶ Более сложные стили, полученные соединением простых стилей
 - ▶ REST
 - ▶ C2
 - ▶ Распределённые объекты
 - ▶ ОО + клиент-сервер
 - ▶ CORBA

С2

- ▶ Стиль с неявным вызовом, где компоненты общаются только через коннекторы, маршрутизирующие сообщения
- ▶ Компоненты: независимые, потенциально параллельные производители или потребители
- ▶ Соединители: маршрутизаторы сообщений, которые могут фильтровать, преобразовывать и рассылать сообщения двух видов: нотификации и запросы
- ▶ Элементы данных: сообщения, содержащие данные
 - ▶ Нотификации анонсируют изменения в состоянии
 - ▶ Запросы — запрашивают выполнение действия
- ▶ Топология: слои компонентов и соединителей с определённым “верхом” и “низом”, где нотификации шлются “вниз”, а запросы — “вверх”

C2 Lunar Lander

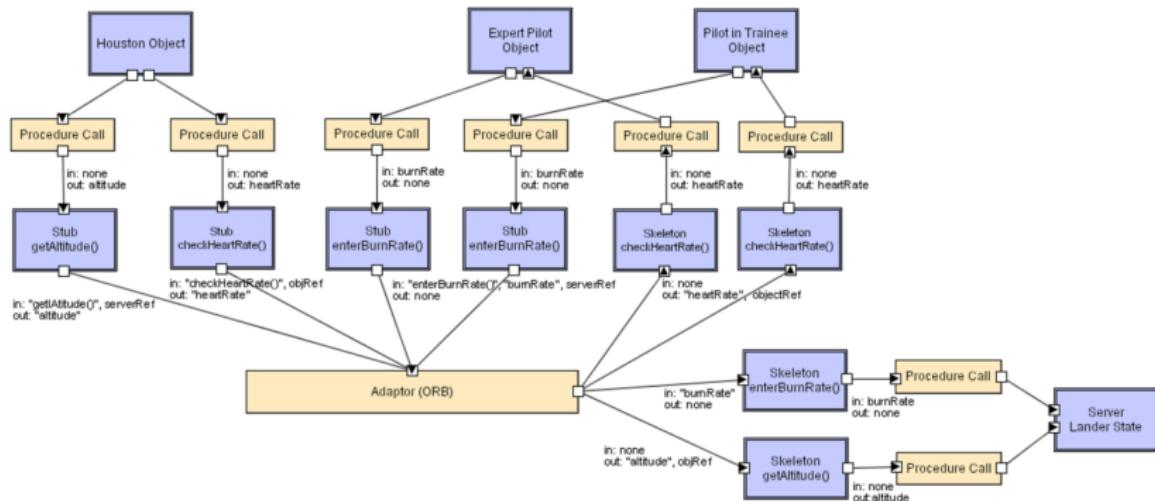


© N. Medvidovic

CORBA

- ▶ “Объекты” работают на гетерогенных хостах, реализованные на разных языках программирования
- ▶ Объекты предоставляют сервисы через чётко определённые интерфейсы и вызывают методы через RPC-протоколы
- ▶ Топология: граф объектов в самом общем смысле
- ▶ Дополнительные ограничения:
 - ▶ Передаваемые при вызове метода данные должны быть сериализуемы
 - ▶ Вызывающие должны обрабатывать ошибки, связанные с работой сети
- ▶ Преимущества: независимость от платформы, языка и местоположения сервиса

CORBA Lunar Lander



© N. Medvidovic

Лекция 11: Domain-Driven Design

Юрий Литвинов

yurii.litvinov@gmail.com

12.11.2020г

Domain-Driven Design

Domain-Driven Design — модная нынче методология проектирования, использующая предметную область как основу архитектуры системы

- ▶ Архитектура приложения строится вокруг **Модели предметной области**
- ▶ Модель определяет **Единый язык**, на котором общаются и разработчики, и эксперты, описывая естественными фразами то, что происходит и в программе, и в реальности
- ▶ Модель — это не только диаграммы, это ещё (и прежде всего) код, и устное общение

DDD даёт ответ на вопрос “откуда брать эти все классы” и позволяет целенаправленно уточнять и улучшать архитектуру системы. Особенно полезно, когда предметная область не очень знакома.

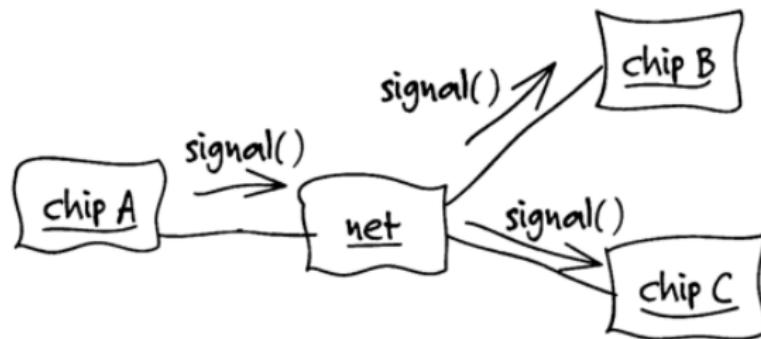
Книжка

Эрик Эванс, “Предметно-ориентированное проектирование. Структуризация сложных программных систем”. М., “Вильямс”, 2010, 448 стр.

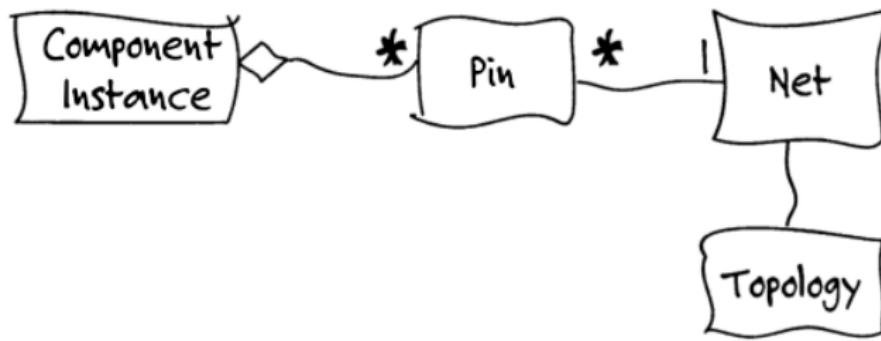


Domain-Driven Design, анализ

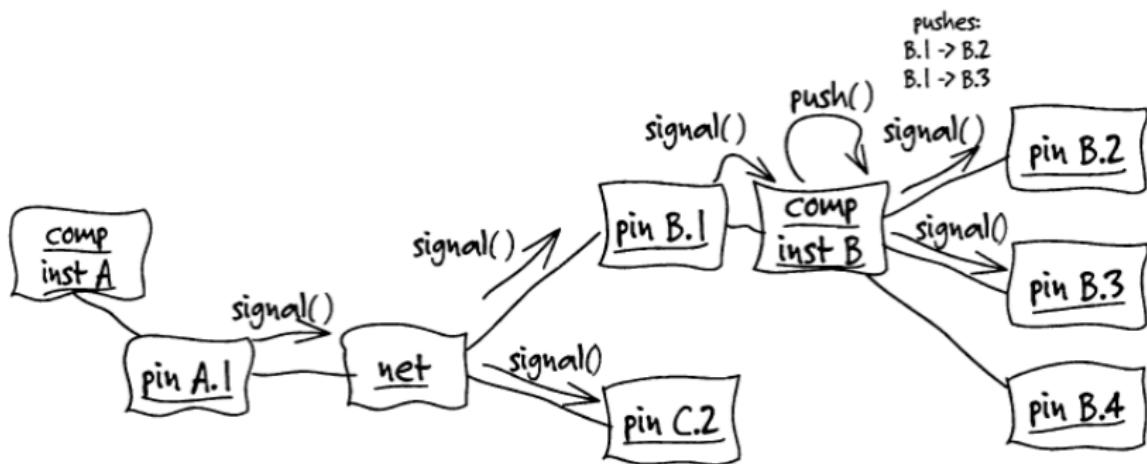
Пример: печатные платы



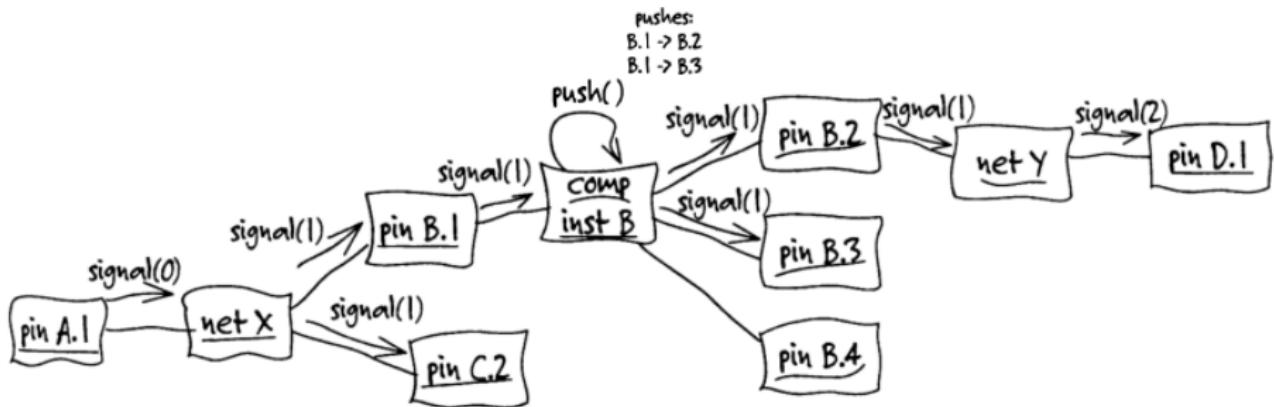
Печатные платы, топология



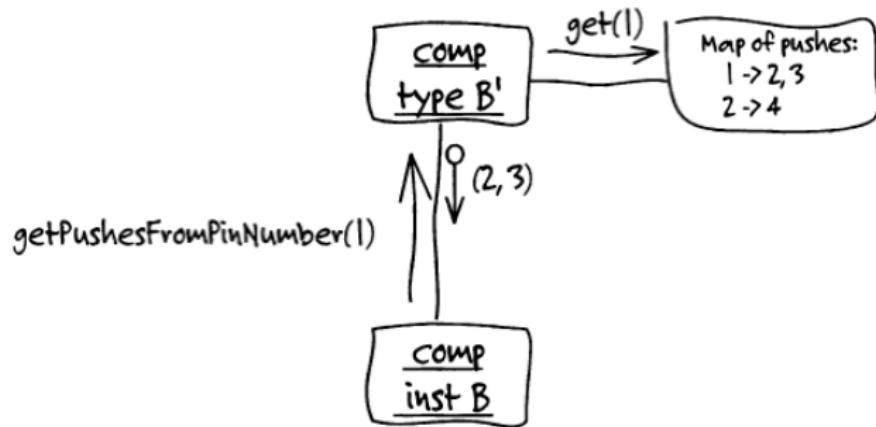
Печатные платы, сигналы



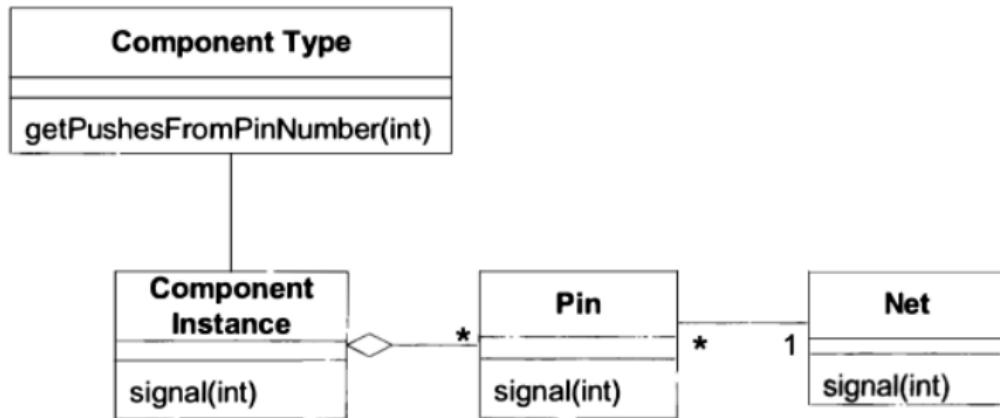
Печатные платы, прозванивание



Печатные платы, типы



Печатные платы, модель



Выводы: правила игры

- ▶ Детали реализации не участвуют в модели
 - ▶ “База данных? Какая база данных?”
- ▶ Должно быть можно общаться, пользуясь только именами классов и методов
- ▶ Не нужные для текущей задачи сущности предметной области не должны быть в модели
- ▶ Могут быть скрытые сущности, которые следует выделить явно
 - ▶ при этом объяснив экспертам их роль в реальной жизни и послушав их мнение
 - ▶ например, различные ограничения могут стать отдельными классами
- ▶ Диаграммы объектов могут быть очень полезны

Единый язык

- ▶ У программистов и специалистов предметной области свой профессиональный жаргон
- ▶ Свои жаргоны появляются даже среди групп разработчиков в одном проекте
- ▶ Необходимость перевода размывает смысл понятий
- ▶ “Еретики” используют понятия в разных смыслах
- ▶ Единый язык — понятия из модели (классы, методы), паттерны, элементы “высокоуровневой” структуры системы (которая не отражается в коде)
- ▶ Изменения в языке — рефакторинг кода
- ▶ Языков в проекте может быть много

Без единого языка

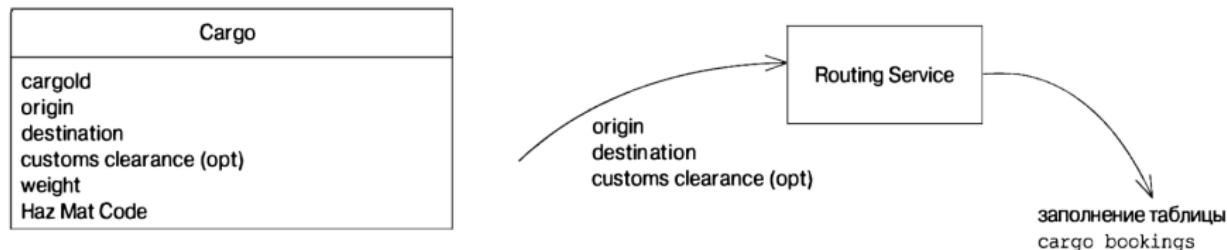
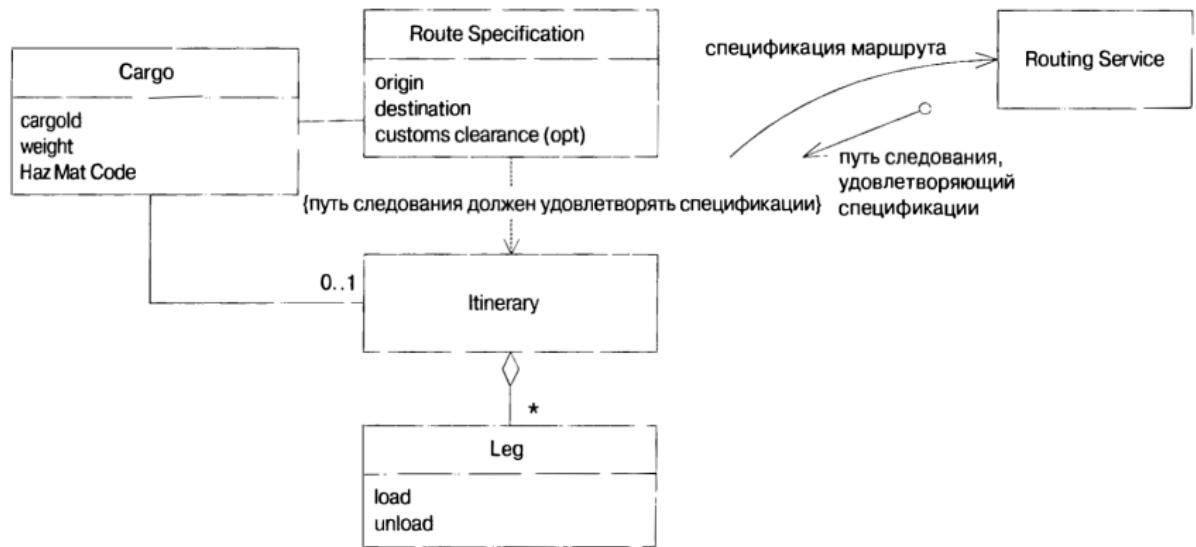


Таблица БД: **cargo_bookings**

Cargo_ID	Transport	Load	Unload

С единым языком



“Моделирование вслух”

*Если передать в **Маршрутизатор** пункт отправки, пункт назначения, время прибытия, то он найдет нужные остановки в пути следования груза, а потом, ну... запишет их в базу данных.*

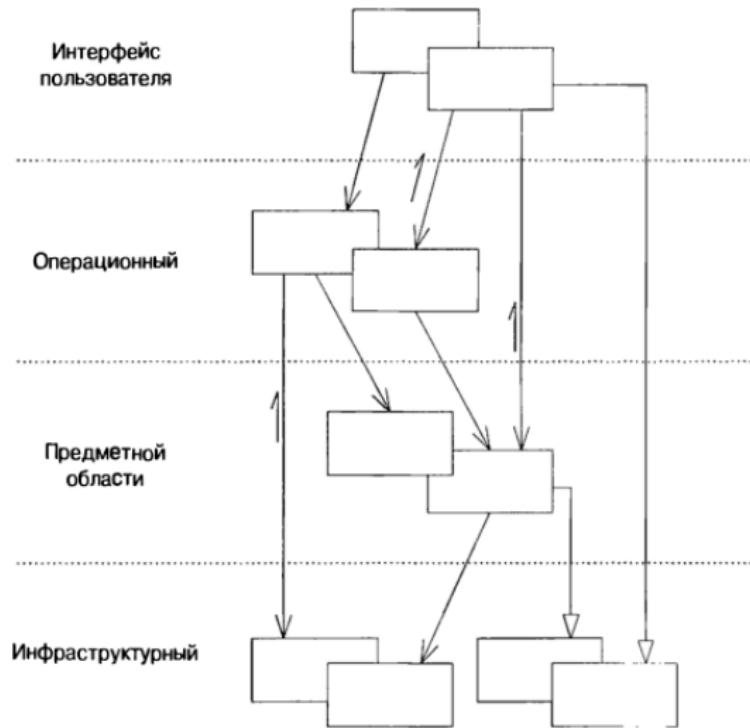
*Пункт отправки, пункт назначения и все такое... все это идет в **Маршрутизатор**, а оттуда получаем **Маршрут**, в котором записано все, что нужно.*

***Маршрутизатор** находит **Маршрут**, удовлетворяющий **Спецификации маршрута**.*

Модель и реализация

- ▶ Модель, не соответствующая коду, бесполезна
- ▶ Код, созданный без модели, скорее всего, работает неправильно
 - ▶ “Разрушительный рефакторинг”
 - ▶ Нельзя разделять моделюровщиков и программистов
- ▶ Модель в DDD выполняет роль и модели анализа, и модели проектирования одновременно
 - ▶ Это требует баланса между техническими деталями и адекватностью выражения предметной области
 - ▶ Часто требуется несколько итераций рефакторинга
- ▶ Язык программирования должен поддерживать парадигму модели
- ▶ Модель, привязанная к реализации, хороша и для пользователя

Изоляция предметной области



Изоляция предметной области, соображения

- ▶ Модель предметной области должна быть отделена от остальной программы
- ▶ Классы модели умеют делать только “суть”
- ▶ Сборка всего воедино и общее управление процессом — на операционный уровень
 - ▶ Бизнес-регламенты — на уровне модели предметной области
- ▶ Все технические вещи — на инфраструктурный уровень
 - ▶ Работа с БД
 - ▶ Middleware, сетевые коммуникации
 - ▶ Утилиты
 - ▶ Абстрактные базовые классы
- ▶ Observer или вариации MVC для связей “снизу вверх”

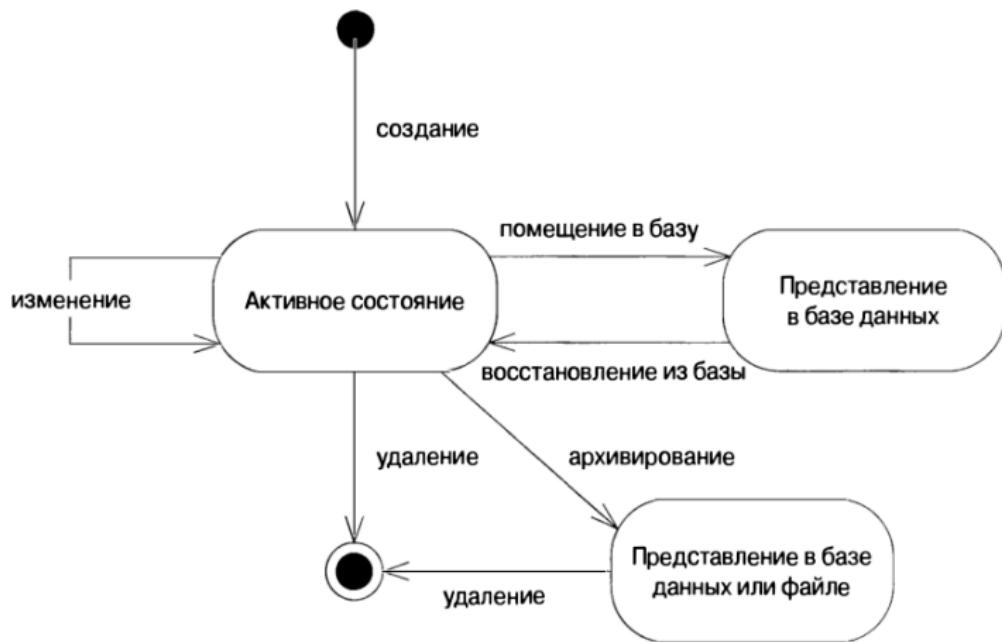
Антипаттерн “Умный GUI”

- ▶ А давайте всю бизнес-логику писать прямо в обработчиках на форме
- ▶ Код GUI напрямую работает с БД
- ▶ Делает невозможным проектирование по модели
- ▶ Не всегда плохо
 - ▶ Применимы средства быстрой разработки приложений
 - ▶ Прирост производительности на начальных этапах
 - ▶ Легко приделывать новые фичи и переписывать старые
- ▶ Не всегда хорошо
 - ▶ Очень сложно переиспользование
 - ▶ Сложно реализовать сложное поведение (зато легко простое)
 - ▶ Сложно интегрироваться

Основные структурные элементы модели

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

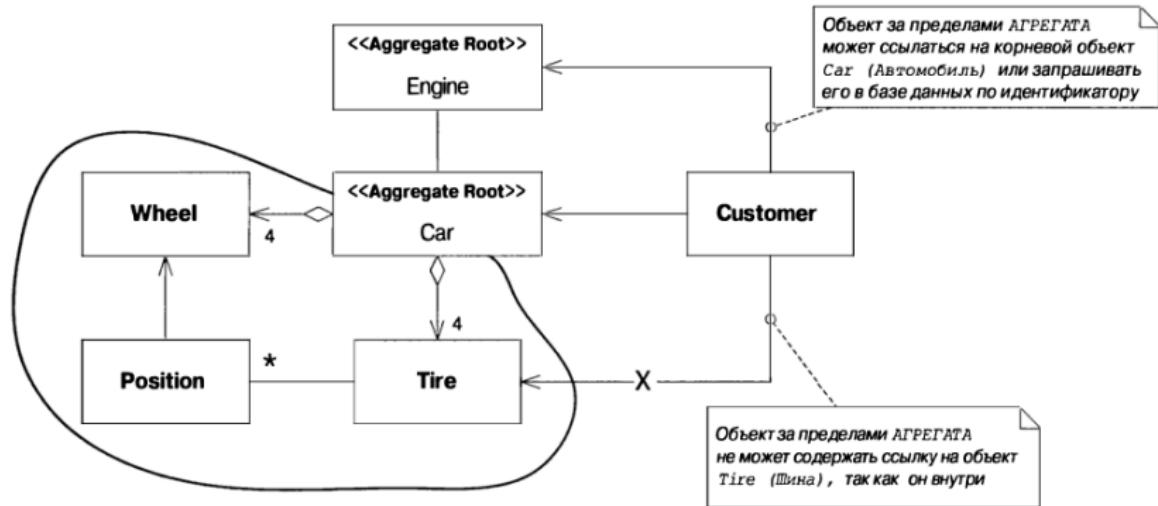
Жизненный цикл объекта



Агрегаты

- ▶ **Агрегат** — изолированный кусок модели, имеющий **корень** и **границу**
- ▶ Корень — глобально идентичный объект-сущность
- ▶ Остальные объекты в агрегате идентичны локально
- ▶ Извне агрегата можно хранить ссылку только на корень
 - ▶ Отдавать временную ссылку можно
- ▶ Корень отвечает за поддержание инвариантов всего агрегата

Агрегат, пример



Фабрика

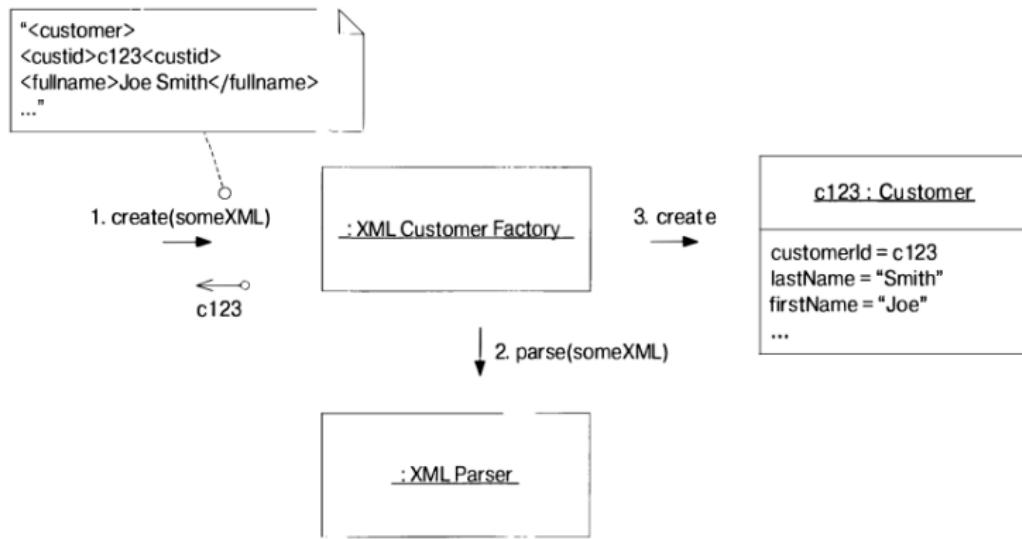


Фабрика служит для создания объектов или агрегатов

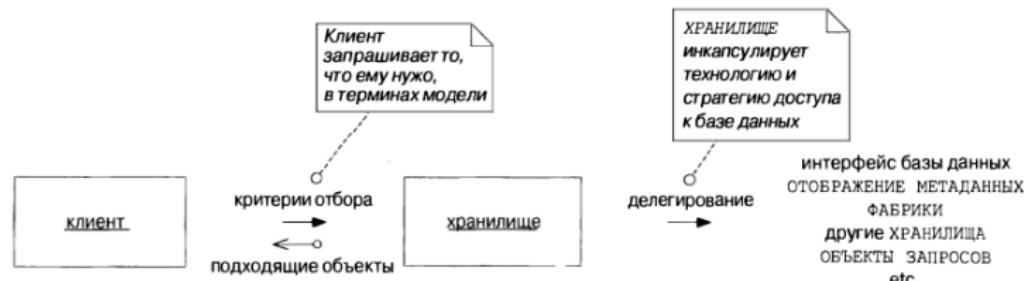
- ▶ Скрывает внутреннее устройство конструируемого объекта
 - ▶ Операция создания “атомарна” и обеспечивает инварианты
- ▶ Изолирует сложную операцию создания
- ▶ Как правило, не имеет бизнес-смысла, но является частью модели
- ▶ Реализуется аж несколькими разными паттернами

Пример

Фабрика, использующаяся для восстановления объекта



Хранилище (Repository)



Репозиторий хранит объекты и предоставляет к ним доступ

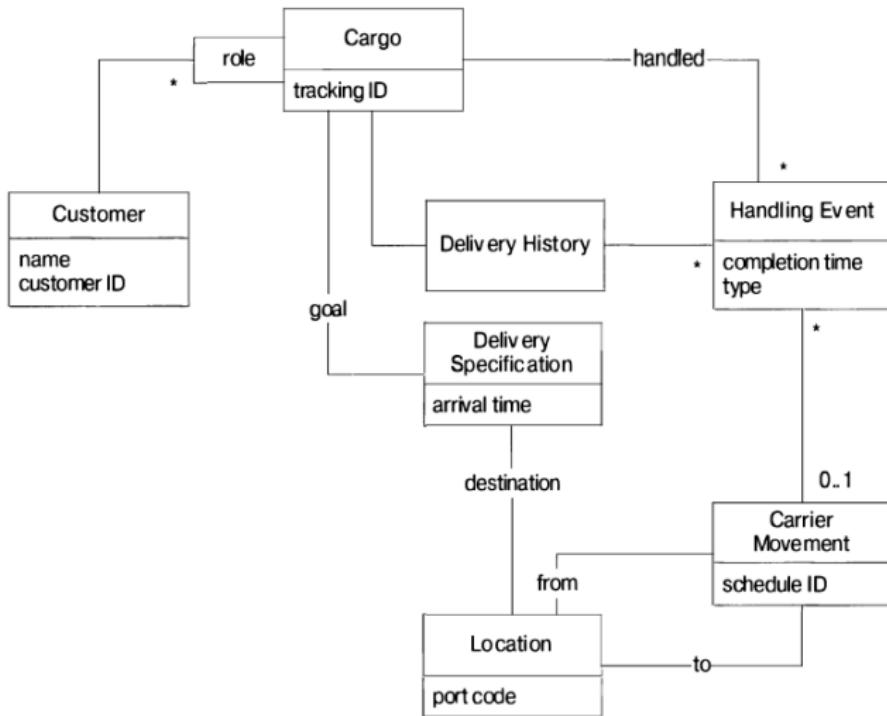
- ▶ Может инкапсулировать запросы к БД
- ▶ Может использовать фабрики
- ▶ Может обладать развитым интерфейсом запросов

Пример, система грузоперевозок

Требования:

1. Отслеживать ключевые манипуляции с грузом клиента
 2. Оформлять заказ заранее
 3. Автоматически высылать клиенту счет-фактуру по достижении грузом некоторого операционного пункта маршрута
- ▶ В работе с Грузом (Cargo) участвует несколько Клиентов (Customers), каждый из которых играет свою роль (Role)
 - ▶ Должна задаваться (be specified) цель (goal) доставки груза
 - ▶ Цель (goal) доставки груза достигается в результате последовательности Переездов (Carrier Movement), которые удовлетворяют Заданию (Specification)

Модель



Уровень приложения

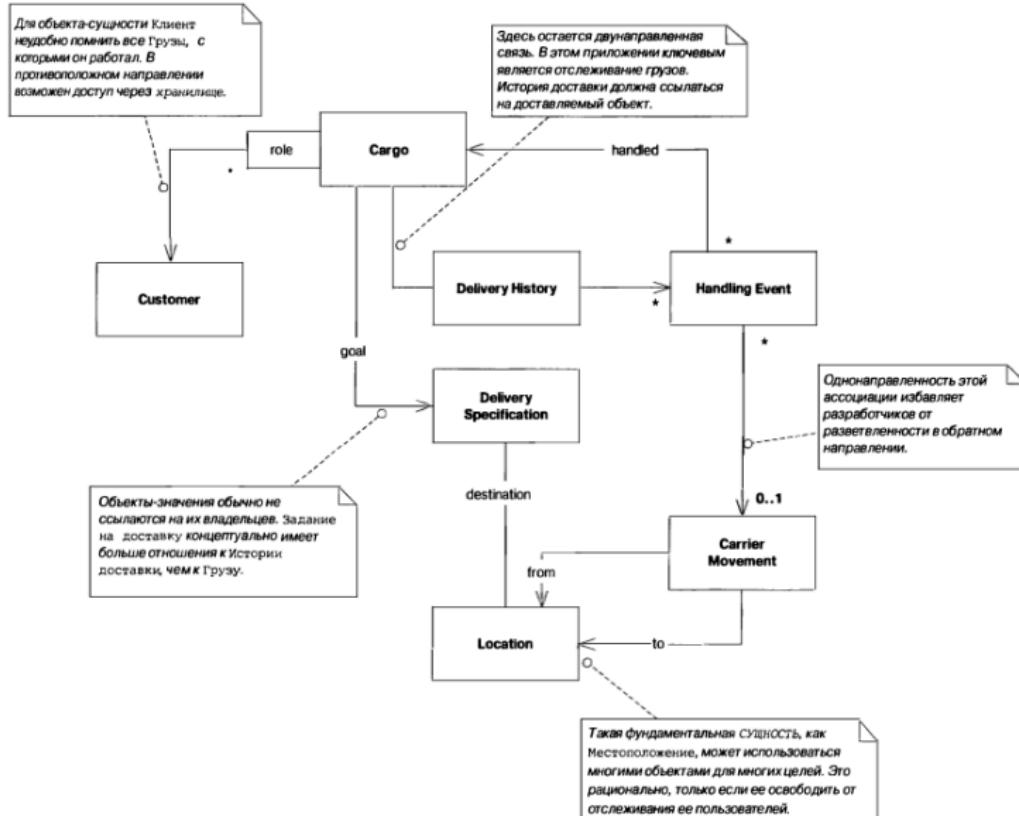
Применим уровневую архитектуру и выделим операции уровня приложения:

- ▶ Маршрутный запрос (Tracking Query) — манипуляции с конкретным грузом
- ▶ Служба резервирования (Booking Application) — позволяет заказать доставку нового груза
- ▶ Служба регистрации событий (Incident Logging Application) — регистрирует действия с грузом (связана с маршрутным запросом)

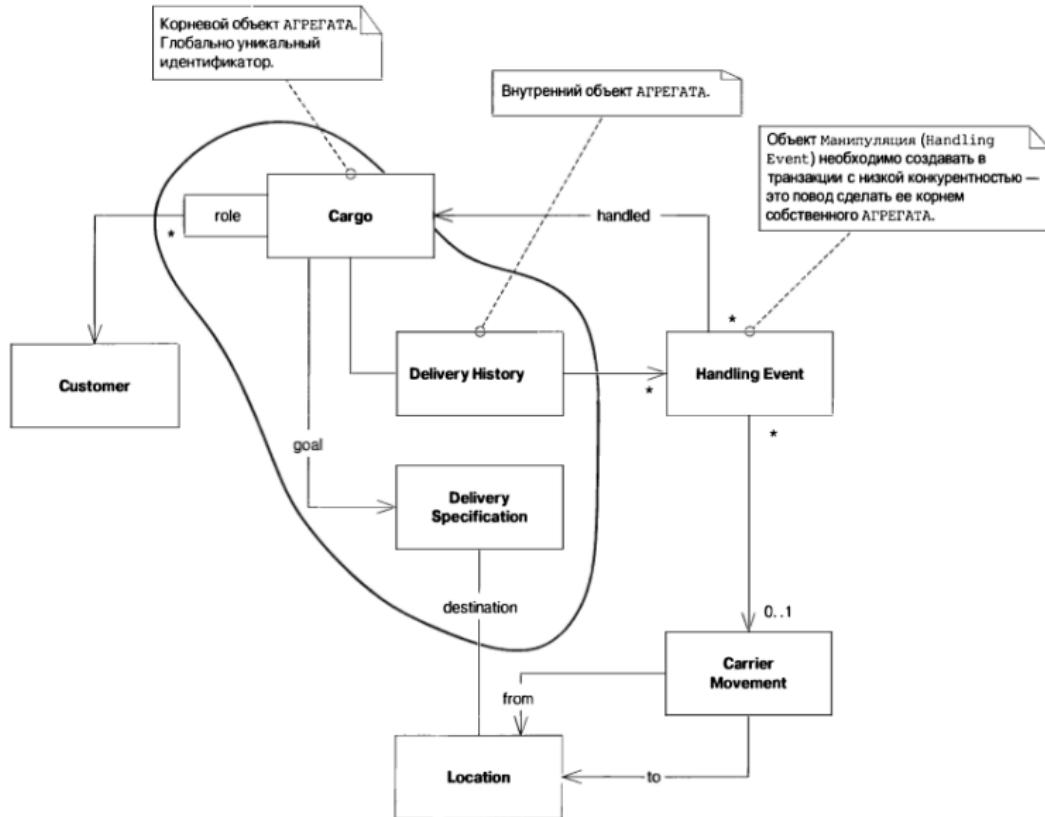
Сущности или значения?

- ▶ **Клиент (Customer)** — сущность
- ▶ **Груз (Cargo)** — сущность
- ▶ **Манипуляция (Handling Event) и Переезд (Carrier Movement)** — сущности
- ▶ **Местоположение (Location)** — сущность
- ▶ **История доставки (Delivery History)** — сущность, локально идентична в пределах агрегата “Груз”
- ▶ **Задание на доставку (Delivery Specification)** — значение
- ▶ Всё остальное — значения

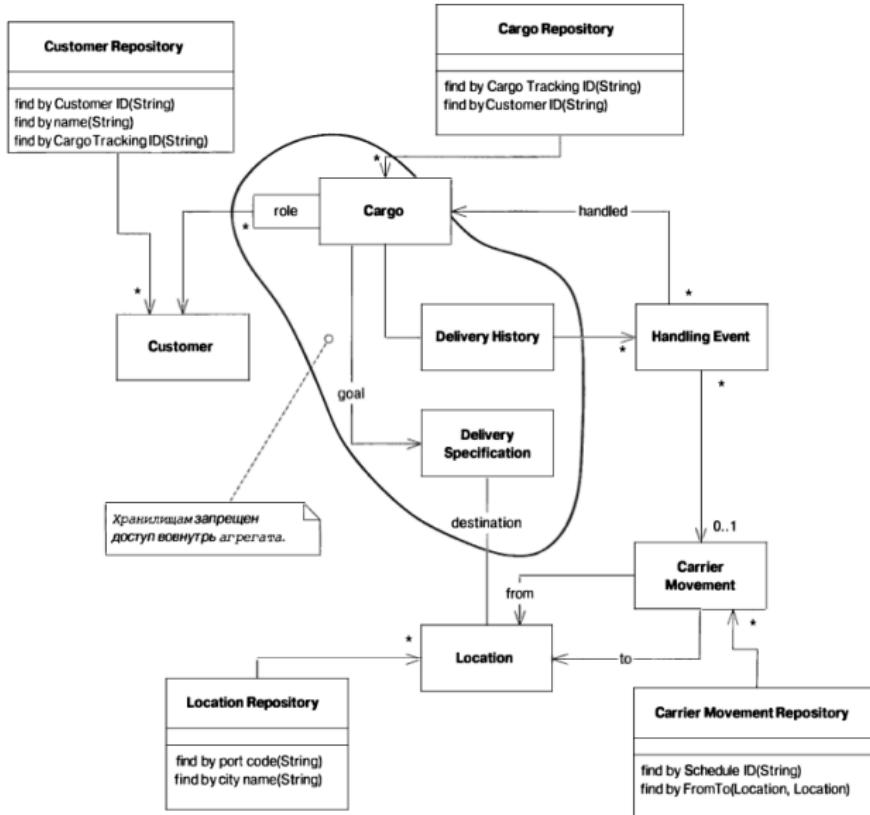
Направленность ассоциаций



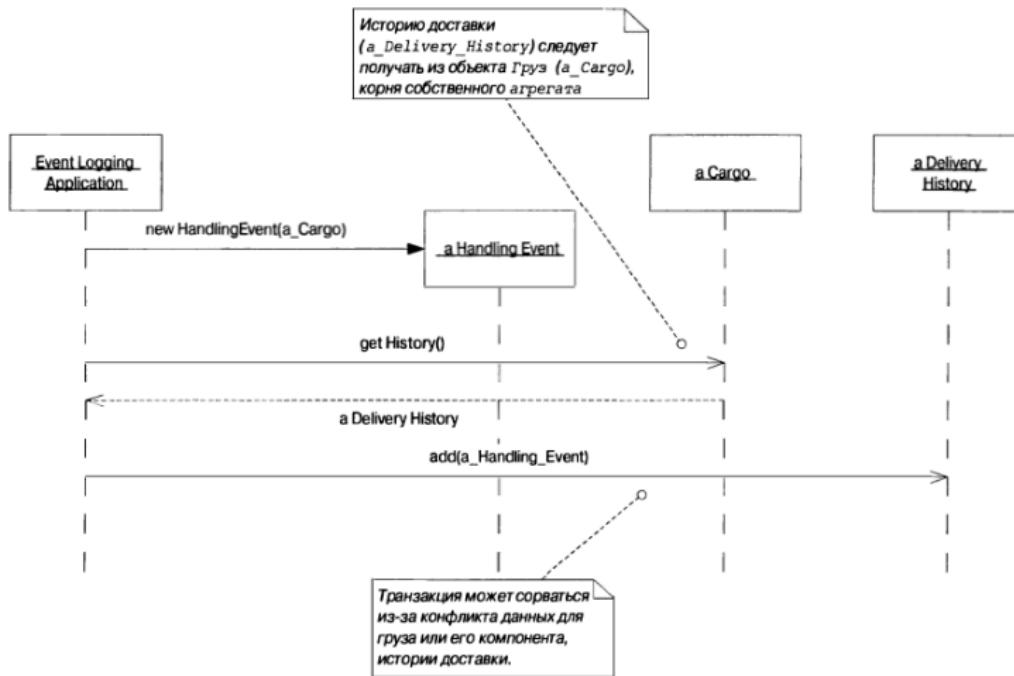
Границы агрегатов



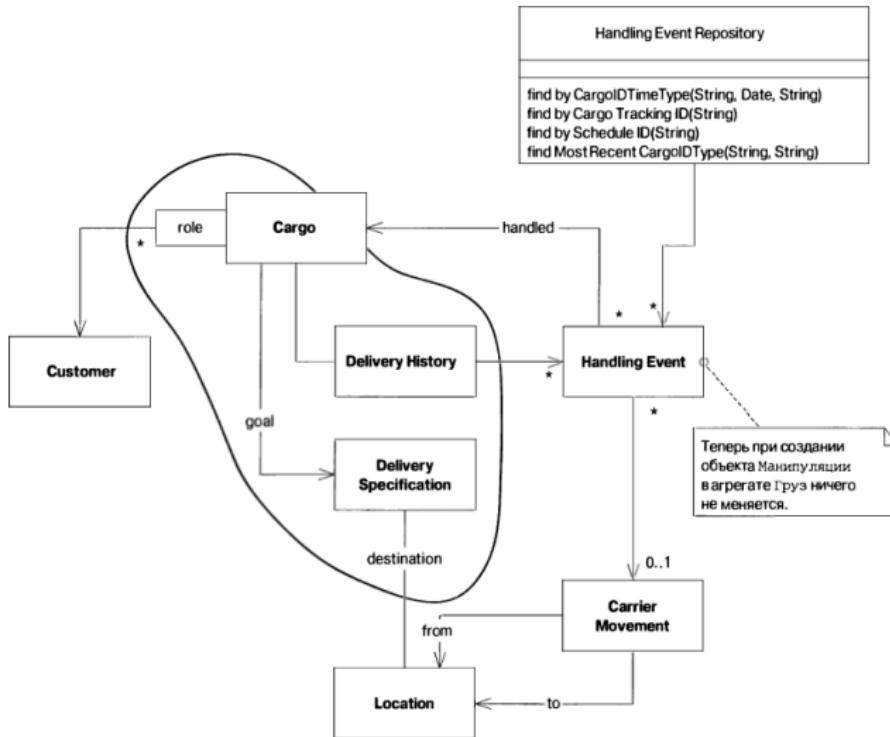
Хранилища



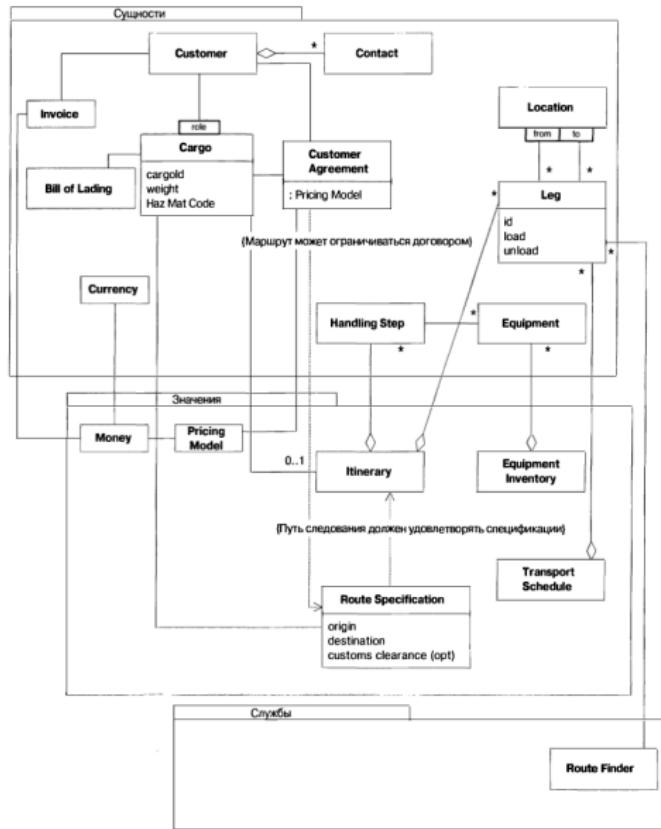
Тестовый сценарий, добавление события



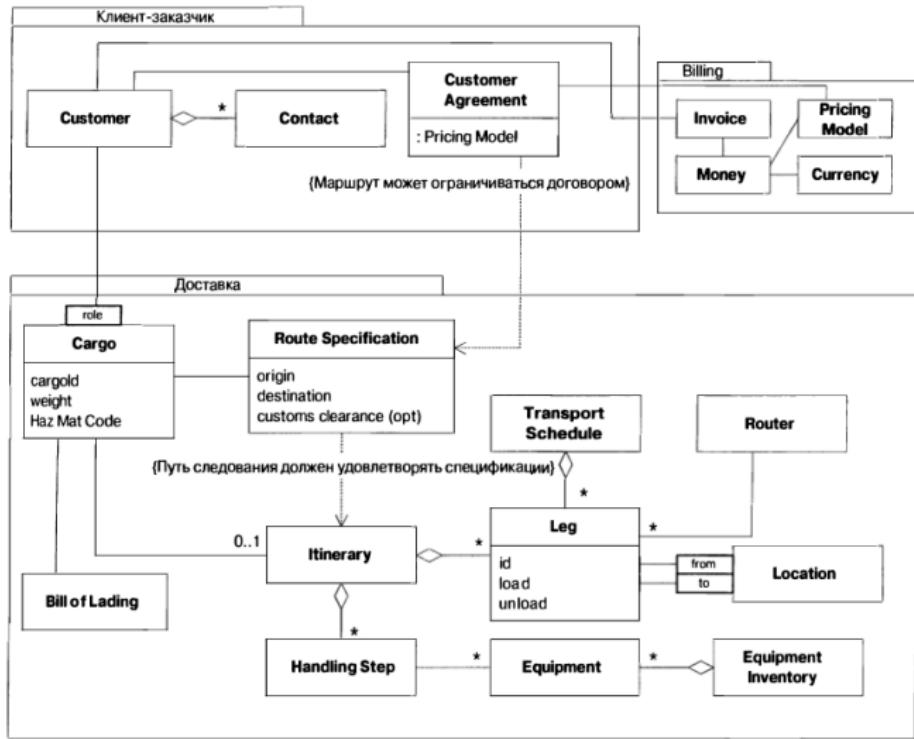
Рефакторинг, не хранить события явно



Разбиение по модулям, плохо

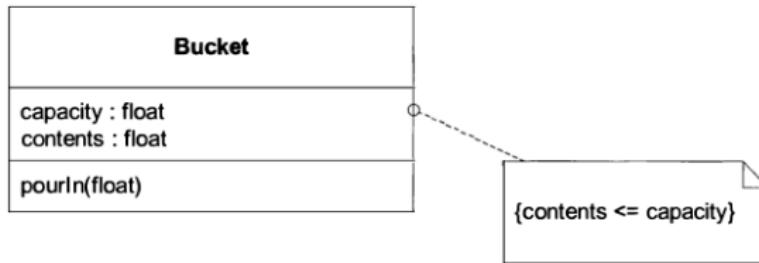


Разбиение по модулям, хорошо



Моделирование ограничений

Простой пример



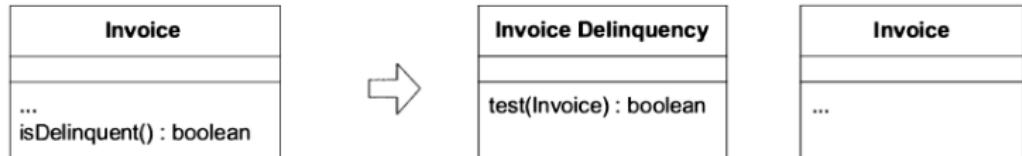
Код, до

```
class Bucket {  
    private float capacity;  
    private float contents;  
  
    public void pourIn(float addedVolume) {  
        if (contents + addedVolume > capacity) {  
            contents = capacity;  
        } else {  
            contents = contents + addedVolume;  
        }  
    }  
}
```

Код, после

```
class Bucket {  
    private float capacity;  
    private float contents;  
  
    public void pourIn(float addedVolume) {  
        float volumePresent = contents + addedVolume;  
        contents = constrainedToCapacity(volumePresent);  
    }  
  
    private float constrainedToCapacity(float volumePlacedIn) {  
        if (volumePlacedIn > capacity) return capacity;  
        return volumePlacedIn;  
    }  
}
```

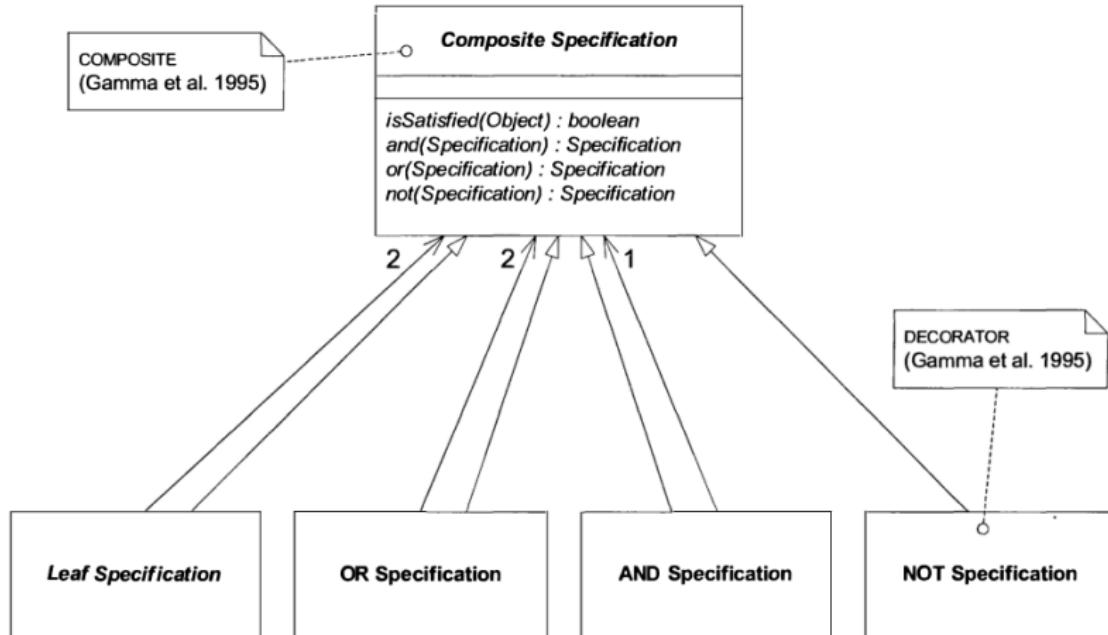
Паттерн “Спецификация”



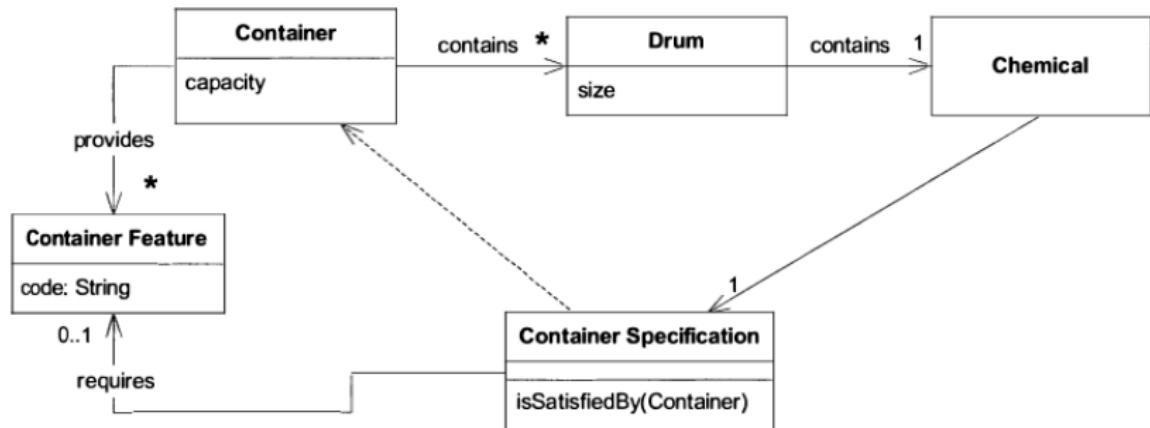
Спецификация инкапсулирует ограничение в отдельном объекте

- ▶ Предикат
- ▶ Может быть использована для выборки или конструирования объектов

Композитные спецификации



Пример: склад химикатов



Код, спецификация

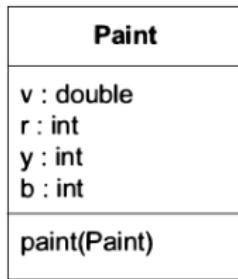
```
public class ContainerSpecification {  
    private ContainerFeature requiredFeature;  
  
    public ContainerSpecification(ContainerFeature required) {  
        requiredFeature = required;  
    }  
  
    boolean isSatisfiedBy(Container aContainer) {  
        return aContainer.getFeatures().contains(requiredFeature);  
    }  
}
```

Код, контейнер

```
boolean isSafelyPacked() {  
    Iterator it = contents.iterator();  
    while (it.hasNext()) {  
        Drum drum = (Drum) it.next() ;  
        if (!drum.containerSpecification().isSatisfiedBy(this))  
            return false ;  
    }  
    return true;  
}
```

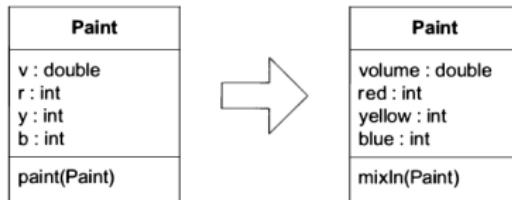
Пример рефакторинга, смешивание красок

Начальное состояние



```
public void paint (Paint paint) {  
    v = v + paint.getV(); // После смешивания объем суммируется  
    // Опущено много строк сложного расчета смешивания цветов,  
    // который заканчивается присваиванием новых значений  
    // компонентов r (красного), b (синего) и у (желтого).  
}
```

Шаг 1: говорящий интерфейс



```

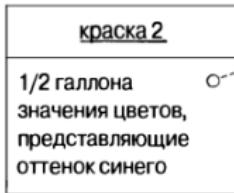
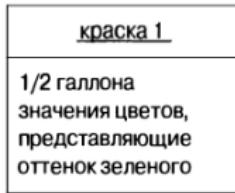
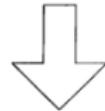
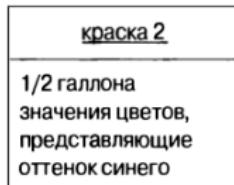
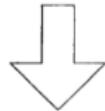
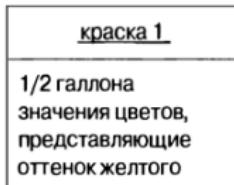
public void testPaint() {
    // Начинаем с чистой желтой краски объемом = 100
    Paint ourPaint = new Paint(100.0, 0, 50, 0);
    // Берем чистую синюю краску объемом = 100
    Paint blue = new Paint(100.0, 0, 0, 50);
    // Примешиваем синюю краску к желтой
    ourPaint.mixIn(blue);
    // Должно получиться 200.0 единиц зеленой краски
    assertEquals(200.0, ourPaint.getVolume(), 0.01);
    assertEquals(25, ourPaint.getBlue());
    assertEquals(25, ourPaint.getYellow());
    assertEquals(0, ourPaint.getRed());
}

```

Шаг 2: функции без побочных эффектов (1)

Проблема

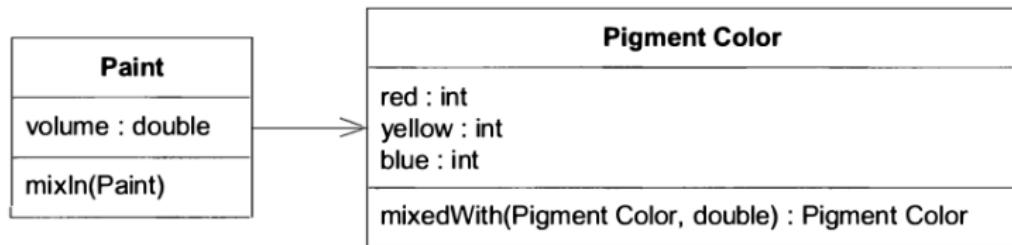
`mixIn(paint2)`
примешиваем краску 2



Что должно быть
здесь? Исходные
разработчики
ничего не указали,
т.к. это их, похоже,
не интересовало.

Шаг 2: функции без побочных эффектов (2)

Идея рефакторинга



Шаг 2: функции без побочных эффектов (3)

Рефакторинг

```
public class PigmentColor {  
    public PigmentColor mixedWith(PigmentColor other, double ratio) {  
        // Много строк сложного расчета смешивания цветов.  
        // в результате создается новый объект PigmentColor  
        // с новыми пропорциями красного, синего и желтого.  
    }  
}  
  
public class Paint {  
    public void mixIn(Paint other) {  
        volume = volume + other.getVolume();  
        double ratio = other.getVolume() / volume;  
        pigmentColor = pigmentColor.mixedWith(other.pigmentColor(), ratio);  
    }  
}
```

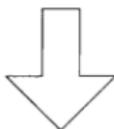
Шаг 2: функции без побочных эффектов (4)

Результат

mixedWith(color2)
смешиваем с цветом 2
→
←○
color 3

цвет 1
значения цветов, представляющие оттенок желтого

цвет 2
значения цветов, представляющие оттенок синего



цвет 3
значения цветов, представляющие оттенок зеленого

Создан новый
ОБЪЕКТ-ЗНАЧЕНИЕ.
Прежние объекты
не изменяются.

Шаг 3: assertions (1)

Инварианты, как они есть

Постуловие для mixIn():

После pl.mixIn(p2):

pl.volume увеличивается на объем p2.volume
p2.volume не изменяется

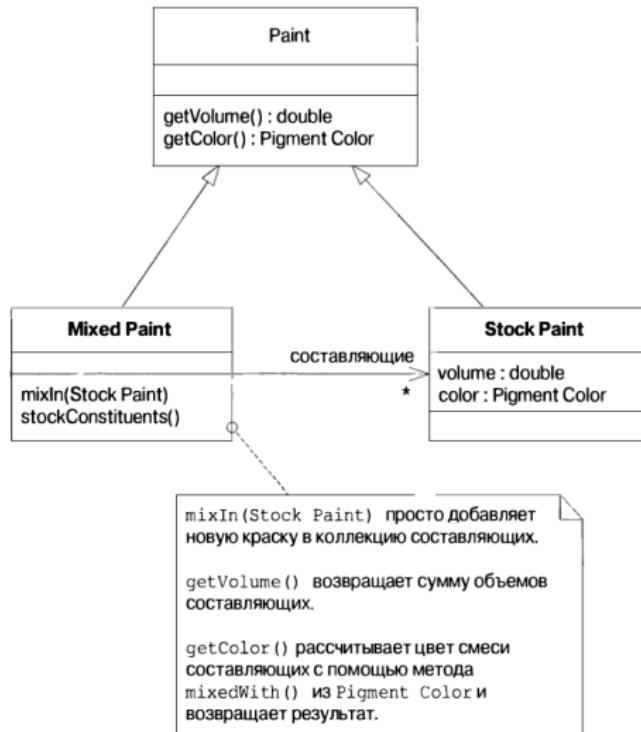
И инвариант:

Общий объем краски не должен измениться от смешивания

???

Шаг 3: assertions (2)

Рефакторинг



Лекция 12: Domain-Driven Design, стратегические аспекты

Юрий Литвинов
yurii.litvinov@gmail.com

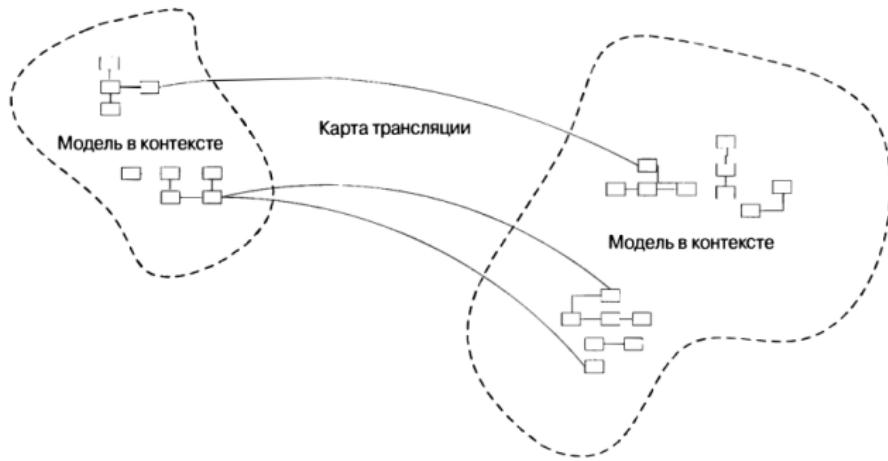
19.11.2020г

Проблемы DDD в больших системах

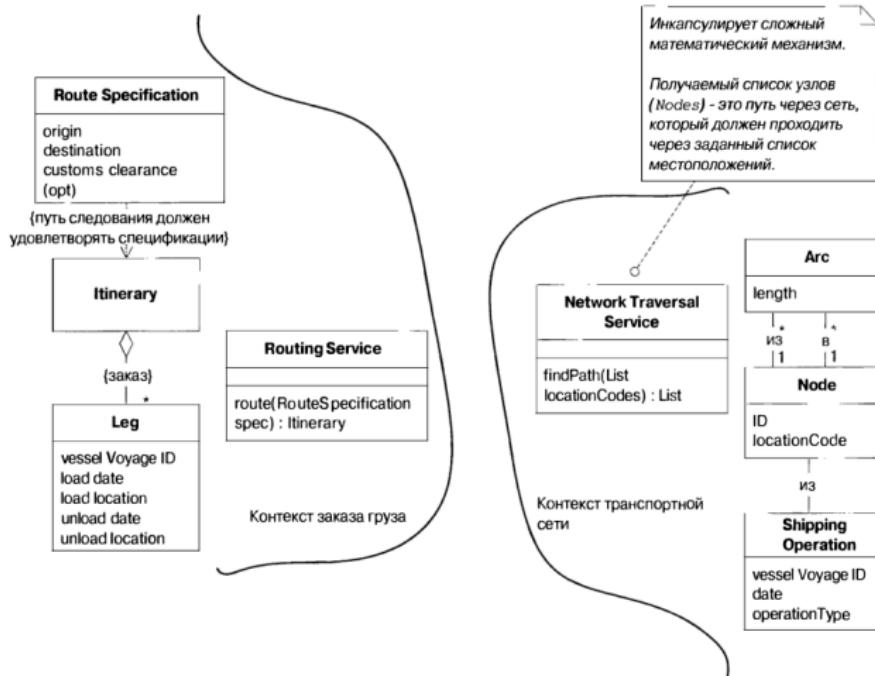
- ▶ Несколько команд => несколько видений продукта
- ▶ Модель предметной области
 - ▶ Интегрированная — слишком большие затраты на поддержание целостности, слишком общая модель, чтобы быть полезной
 - ▶ Фрагментированная — затрудняет переиспользование и интеграцию системы
- ▶ Опасность ошибок при интеграции и переиспользовании
 - ▶ Класс “Платёж” — платёж поставщику или платёж клиента

Принципы поддержания целостности модели

- ▶ Ограниченный контекст (Bounded context)
- ▶ Непрерывная интеграция (Continuous Integration)
- ▶ Карта контекстов (Context map)



Пример, границы контекстов

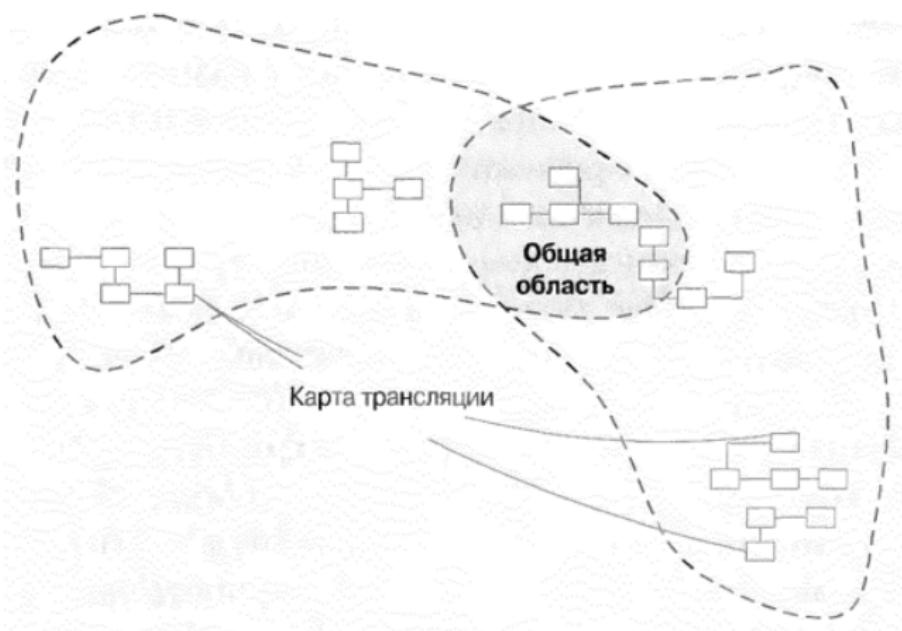


Типовые ситуации интеграции контекстов

- ▶ Общее ядро (Shared Kernel)
- ▶ Заказчик-поставщик (Customer-Supplier)
- ▶ Конформист (Conformist)
- ▶ Предохранительный уровень (Anticorruption Layer)
- ▶ Отдельное существование (Separate ways)
- ▶ Служба с открытым протоколом (Open Host Service)
- ▶ Общедоступный язык (Published Language)

Общее ядро

Shared Kernel



Заказчик-поставщик

Customer-Supplier

- ▶ Имеет смысл, когда одна компонента целиком зависит от другой
- ▶ Может привести к блокированию действий одной или другой команды
- ▶ Следует явно зафиксировать отношения между команды
 - ▶ Одна выступает в роли заказчика (одного из заказчиков) — участвует в планировании, поставляет задачи
 - ▶ Автоматизированные приёмочные тесты
- ▶ Желательно, чтобы команды находились в одной иерархии управления

Конформист

Conformist

- ▶ Имеет смысл, когда нет способа повлиять на компоненту, от которой полностью зависим
 - ▶ Legacy-приложение, навязанная сверху технология и т.п.
- ▶ Просто принимаем модель и миропонимание “основной” компоненты
- ▶ Не всегда плохо: чужой код может на самом деле выражать большее понимание предметной области

Предохранительный уровень

Anticorruption Layer

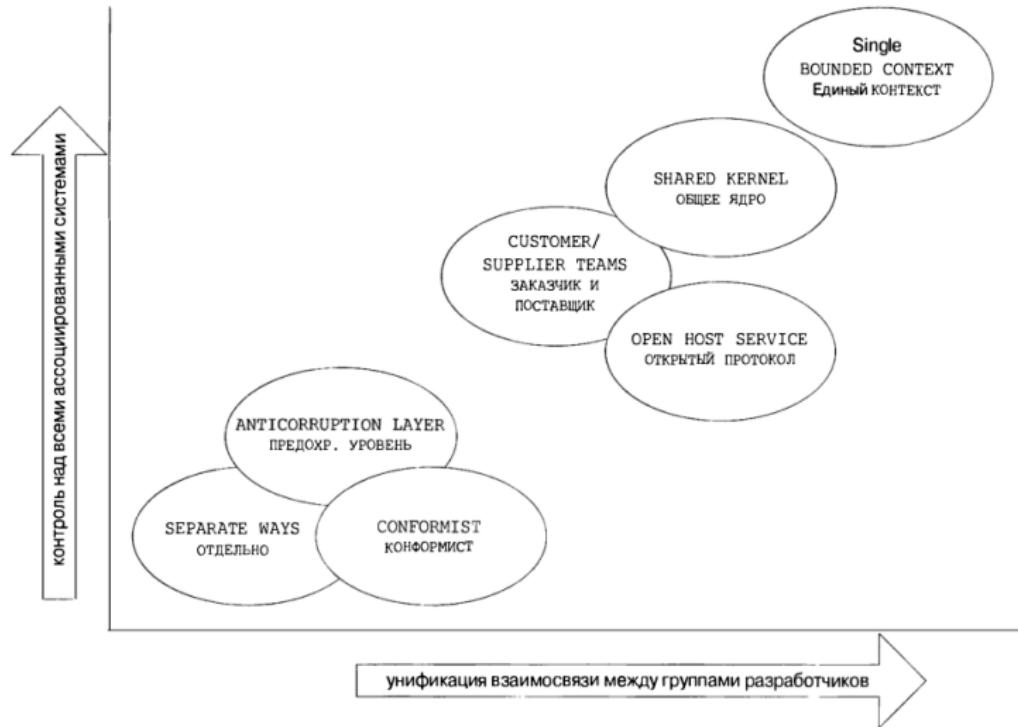
- ▶ Имеет смысл, когда “Конформист” не подходит
- ▶ Кусок кода (возможно, большой и страшный), отвечающий за трансляцию из одной модели в другую
 - ▶ Паттерны “Фасад” и “Адаптер”



Ещё приёмы

- ▶ **Отдельное существование (Separate ways)** — когда преимущества от интеграции меньше затрат на неё
- ▶ **Служба с открытым протоколом (Open Host Service)** — когда клиентов много
- ▶ **Общедоступный язык (Published Language)** — когда клиентов очень много, общая среда для общения

Итого, шаблоны интеграции



Пример: унификация слона

Шесть седовласых мудрецов
Сошли из разных стран.
К несчастью, каждый был незряч,
Зато умом блестал.
Они исследовать слона
Явились в Индостан.

Один погладил бок слона.
Довольный тем сполна,
Сказал он: "Истина теперь
Как божий день видна:
Предмет, что мы зовем слоном,
Отвесная стена!"

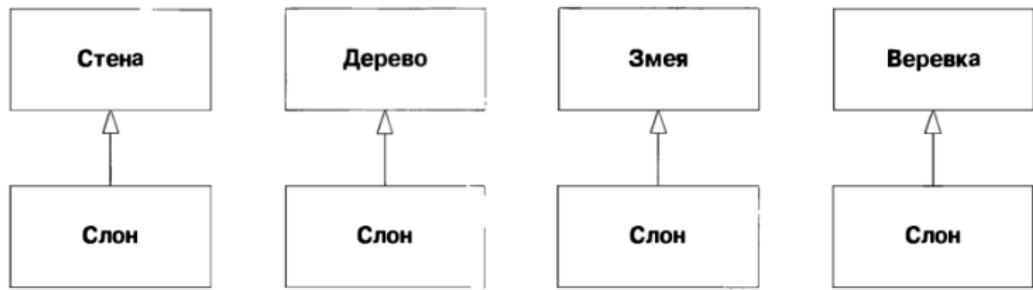
А третий хобот в руки взял
И закричал: "Друзья!
Гораздо проще наш вопрос,
Уверен в этом я!
Сей слон — живое существо,
А именно змея!"

Мудрец четвертый обхватил
Одну из ног слона
И важно молвил: "Это ствол,
Картина мне ясна!
Слон — дерево, что зацветет,
Когда придет весна!"

Тем временем шестой из них
Добрался до хвоста.
И рассмеялся от того,
Как истина проста.
"Ваш слон — веревка. Если ж нет
Зашейте мне уста!"

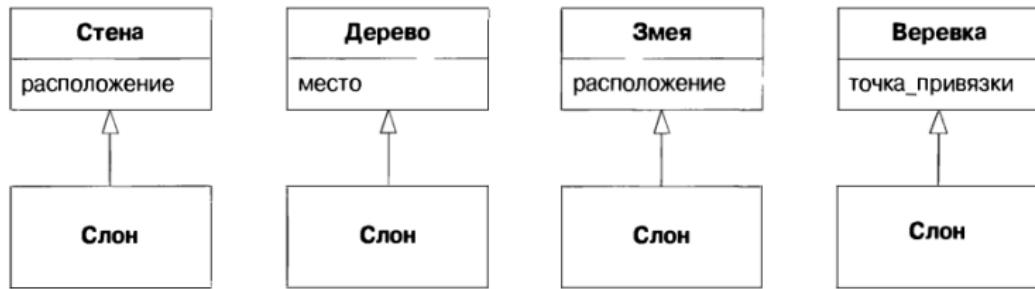
А как известно, мудрецам
Присущ упрямый нрав.
Спор развязав, они дошли
Едва ль не до расправ.
Но правды ни один не знал,
Хотя был в чем-то прав.

Унификация слона, Separate ways



Слон, минимальная интеграция

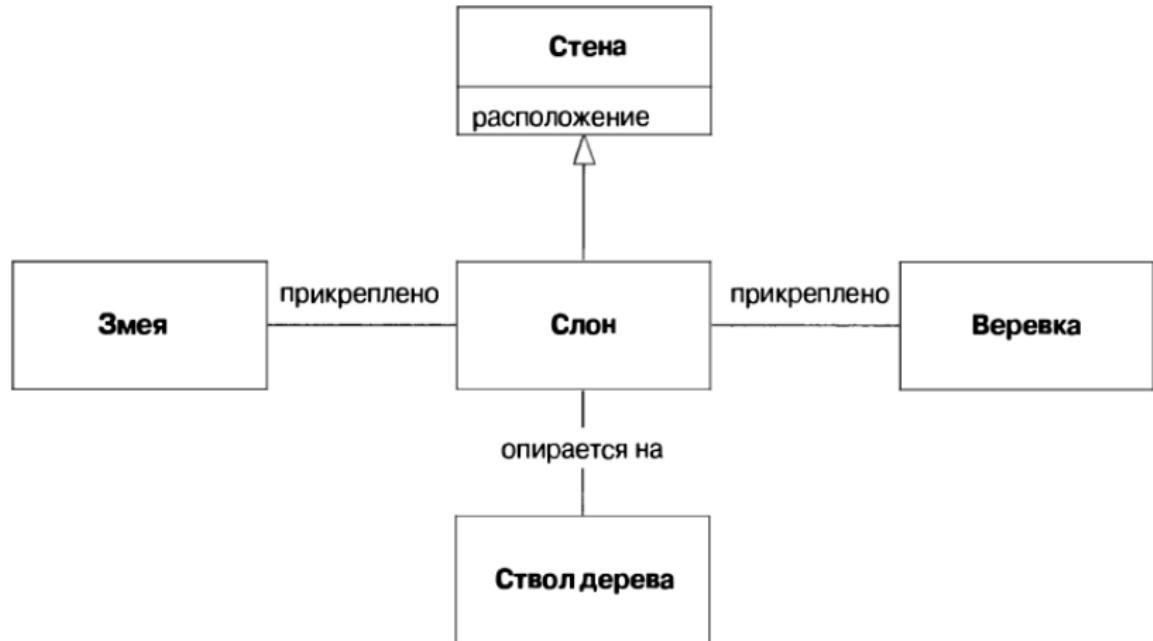
Anticorruption Layer



Трансляция: {Стена . расположение \leftrightarrow Дерево . место \leftrightarrow Змея . расположение \leftrightarrow Веревка . точка_привязки}

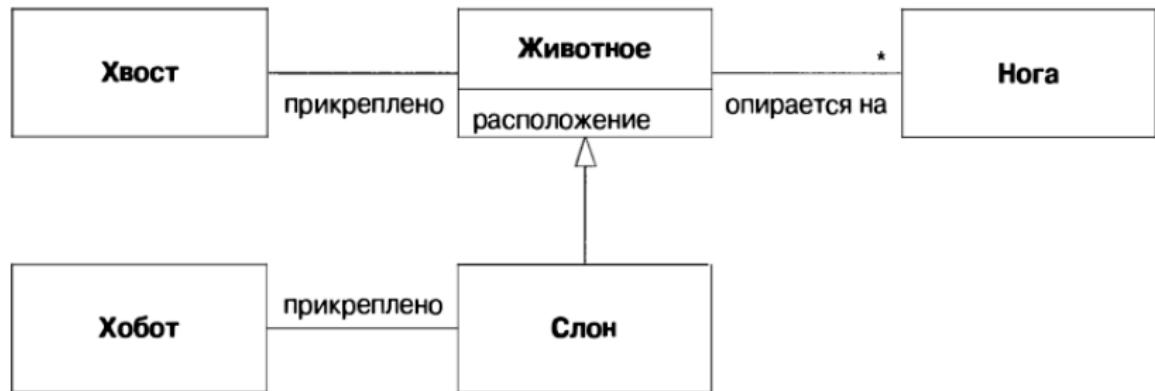
Слон, слабая интеграция

Shared Kernel



Слон, сильная интеграция

Bounded Context



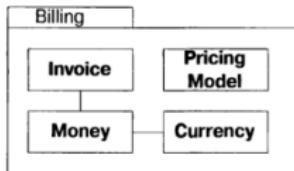
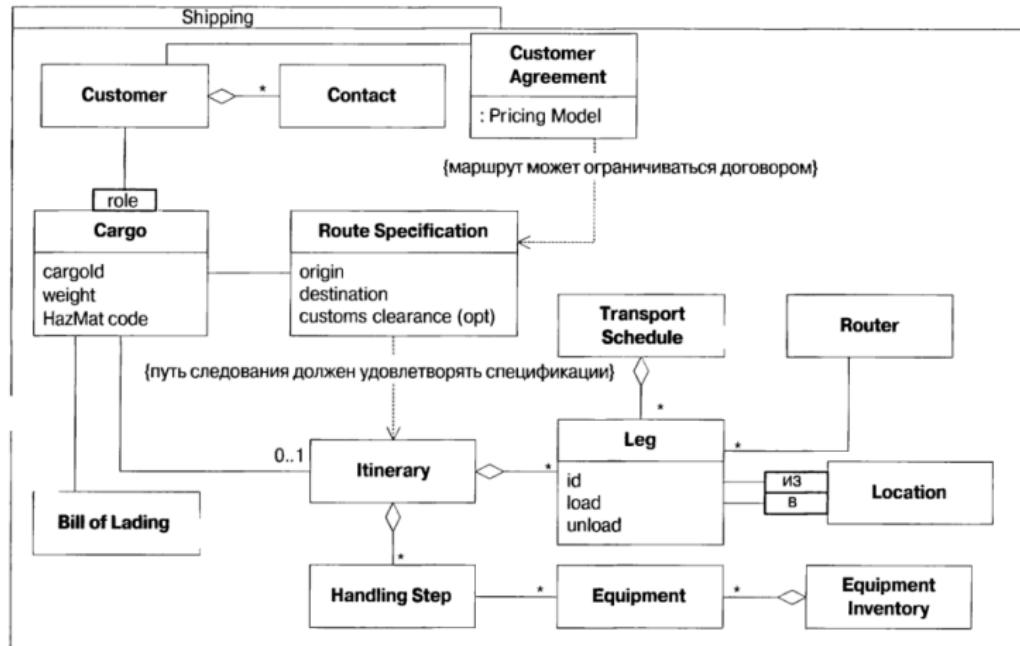
Дистилляция

- ▶ **Дистилляция** — процесс выделения самого существенного в системе и отделения его от вспомогательного кода
- ▶ **Смыслоное ядро (Core Domain)** — то, что, собственно, делает систему ценной
 - ▶ Должно быть минимальным и чётко отделённым от остальных компонент системы
 - ▶ Опытные программисты не любят им заниматься, с этим надо бороться
 - ▶ Только Core Domain, фактически, составляет know-how

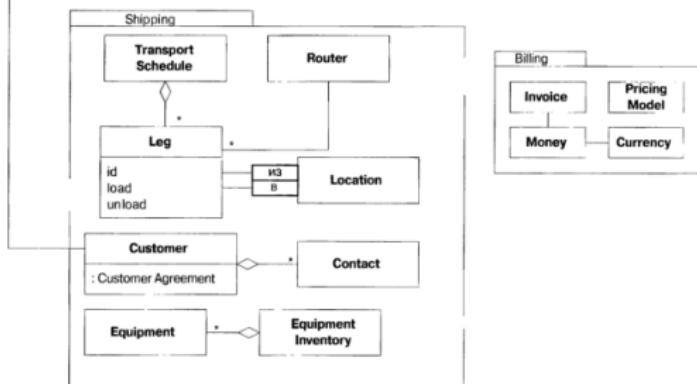
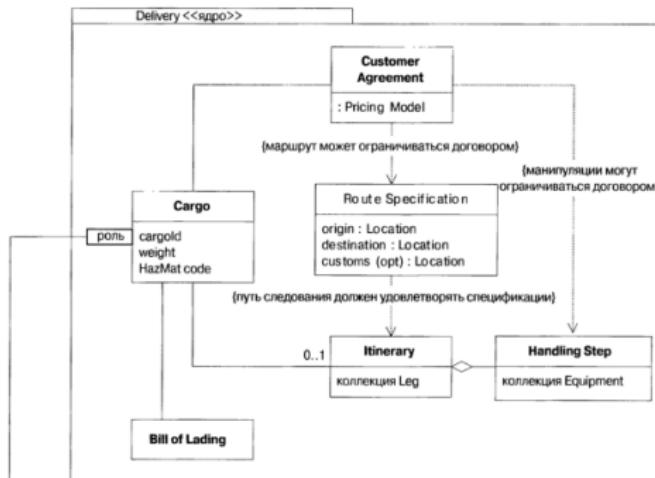
Приёмы дистилляции

- ▶ **Неспециализированные подобласти (Generic Subdomains)** — куски кода, неспецифичные для системы
- ▶ **Domain Vision Statement** — документ (на одну страницу), описывающий смысловое ядро и его полезность
- ▶ **Выделенное ядро (Highlighted Core)**
 - ▶ Дистилляционный документ — 3-7 страниц текста про то, что составляет смысловое ядро и как его элементы взаимодействуют друг с другом
 - ▶ Flagged Core — элементы ядра выделены на существующей модели
- ▶ **Связный механизм (Cohesive Mechanism)** — куски кода, неспецифичные для предметной области вообще
 - ▶ Технические вещи, типа графов

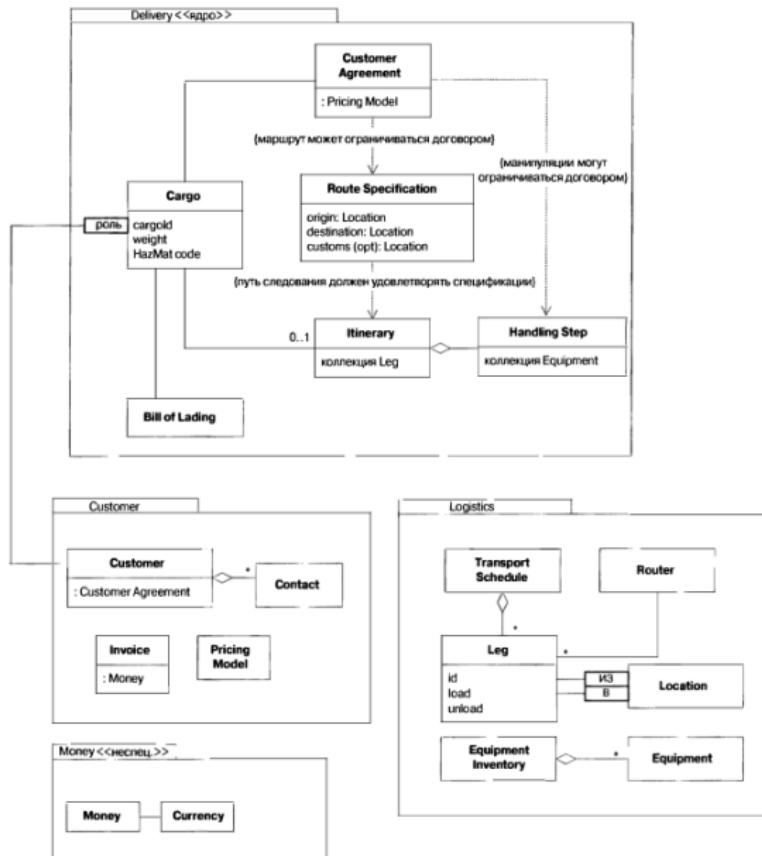
Пример, грузоперевозки



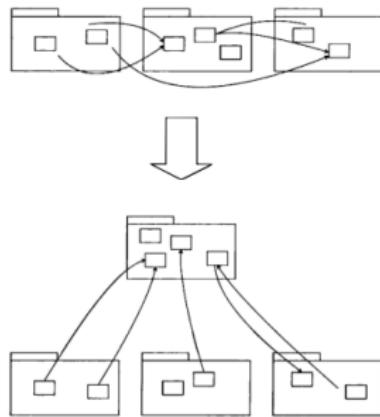
Смыслоное ядро



Смыслоное ядро



Абстрактное ядро



- ▶ Применяется, когда даже ядро оказывается слишком большим
- ▶ Состоит из абстрактных классов, которые потом реализуют отдельные модули

Крупномасштабная структура

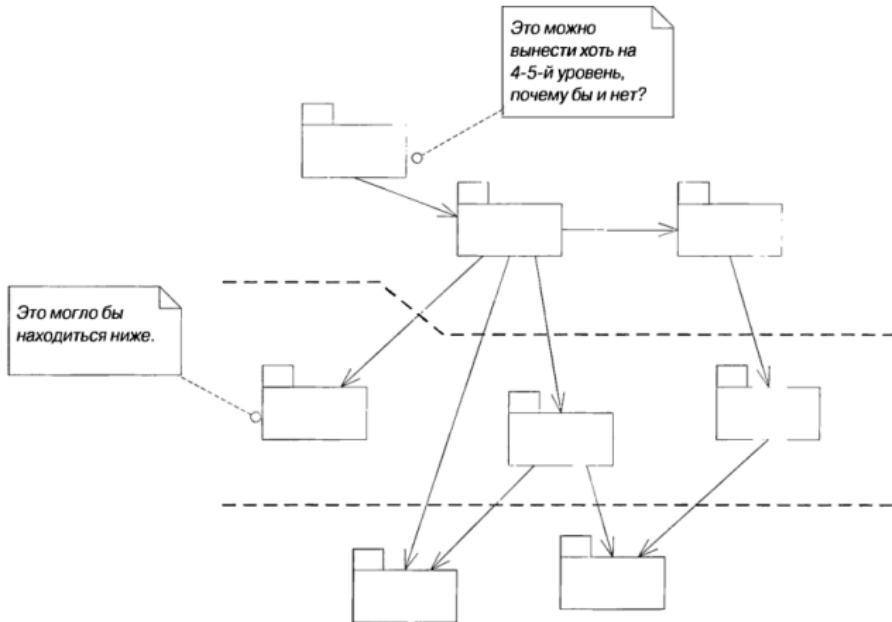
- ▶ **Крупномасштабная структура** — набор общих правил, по которым строится система или группа систем
- ▶ Должна эволюционировать вместе с моделью и кодом
- ▶ Не должна быть слишком жёсткой
 - ▶ Модель “Архитектор в башне из слоновой кости” не работает
- ▶ Лучше какая-то, чем никакой
- ▶ Небольшие проекты могут прекрасно жить и без всего этого
- ▶ Самая полезная структура — общий язык

Метафора системы

- ▶ **Метафора** определяет то, как в целом понимать систему
 - ▶ Множества примеров: рабочий стол, firewall и т.д.
- ▶ Метафора не всегда есть
 - ▶ Иногда используется термин “наивная метафора”, обозначающий метафору, в точности соответствующую модели, но термин сам по себе плох
- ▶ Метафора может быть опасной
 - ▶ Метафора тащит за собой лишний смысл

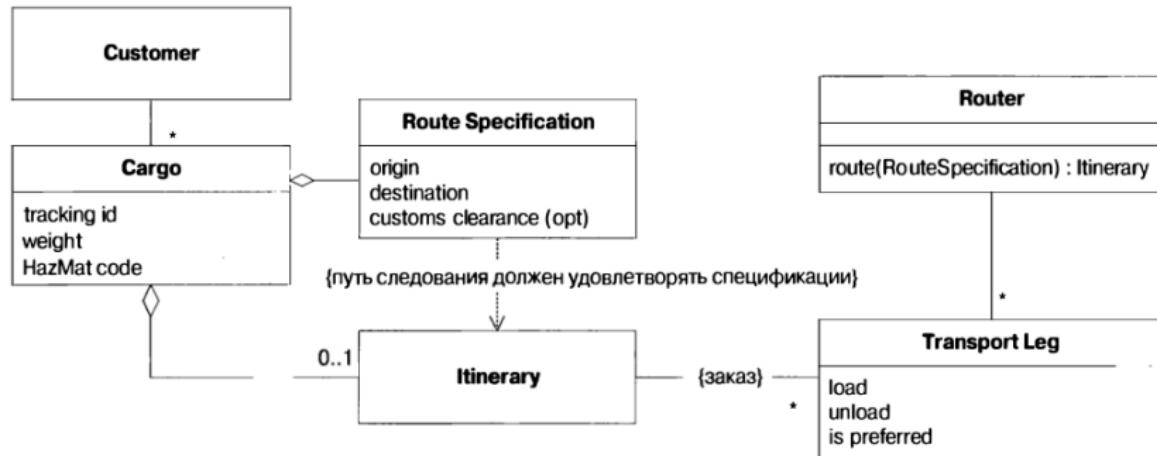
Уровневая структура

Не должна быть механической

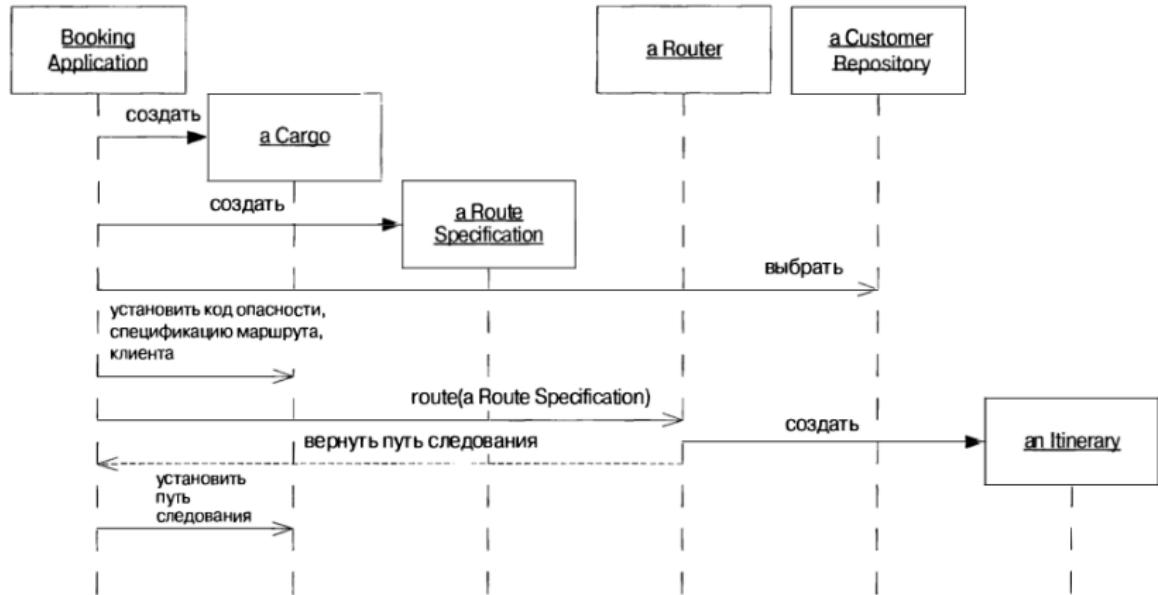


Пример, перевозка грузов

Исходная модель



Установка пути следования

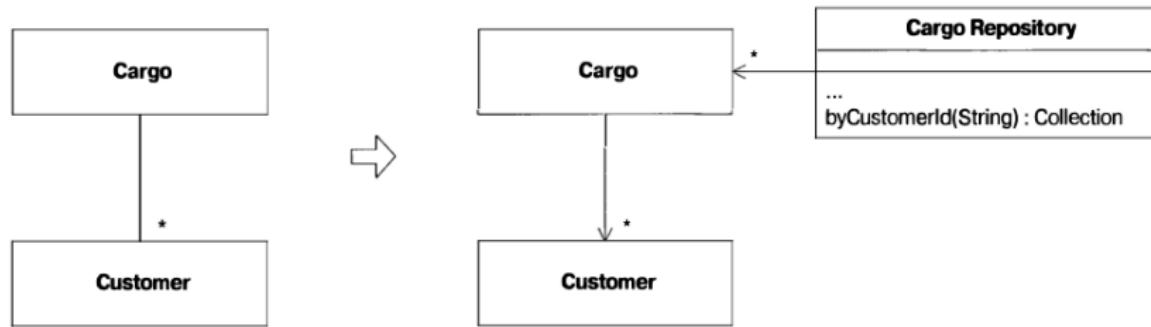


Рефакторинг

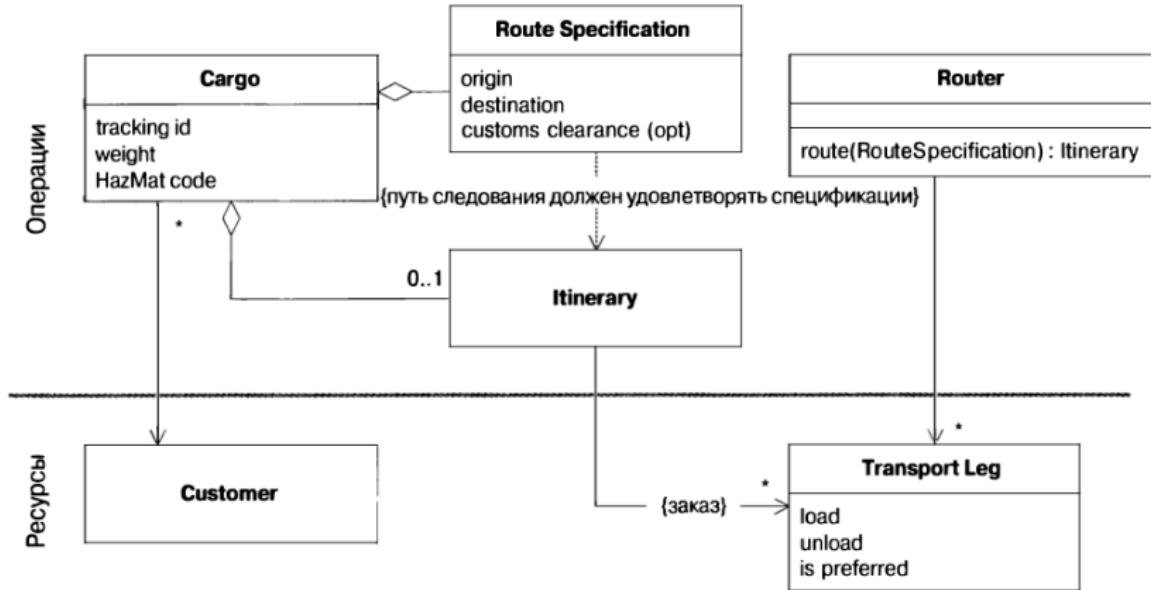
Два уровня:

- ▶ ресурсный — то, что обеспечивает наши возможности
- ▶ операционный — то, как мы пользуемся нашими возможностями

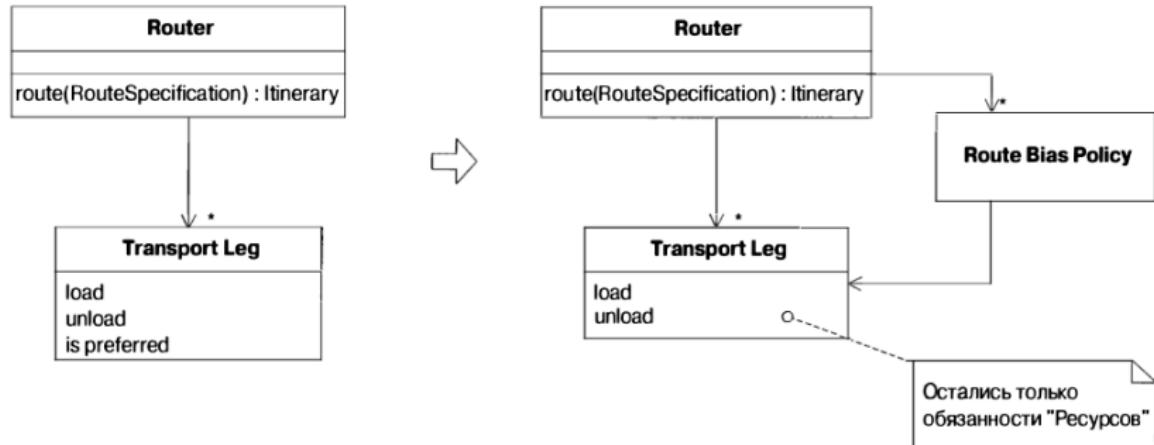
Двунаправленная связь между *Customer* и *Cargo* мешает



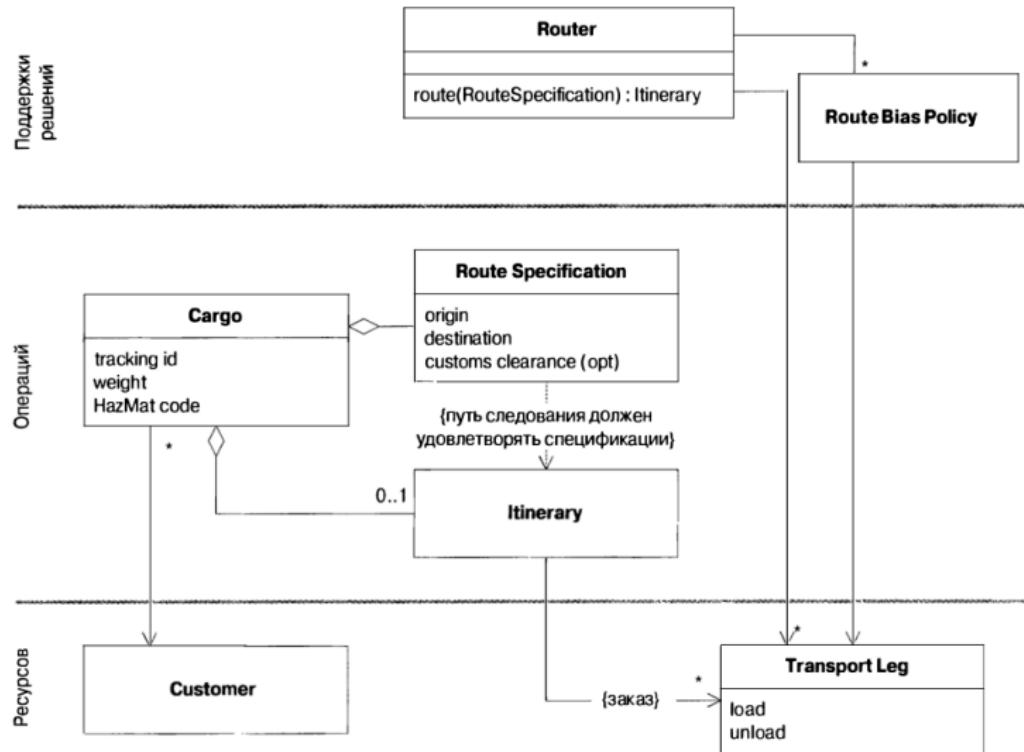
Два уровня



Рефакторинг, выделение уровня принятия решений



Три уровня



Работа с опасными грузами, первая версия

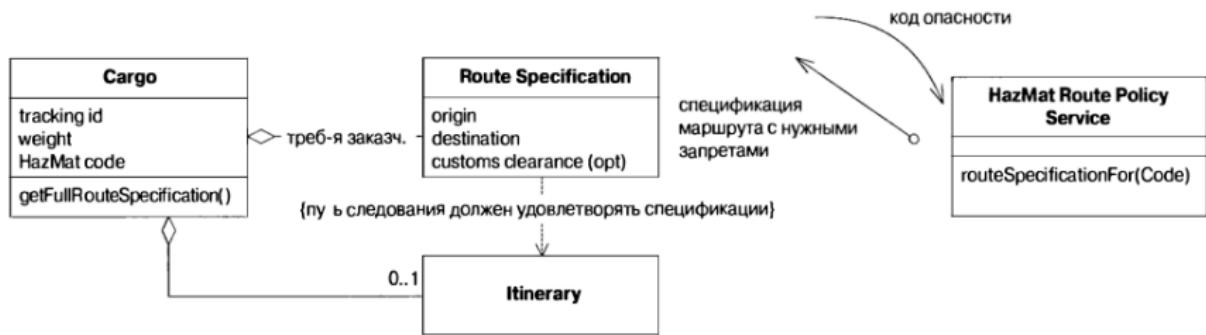
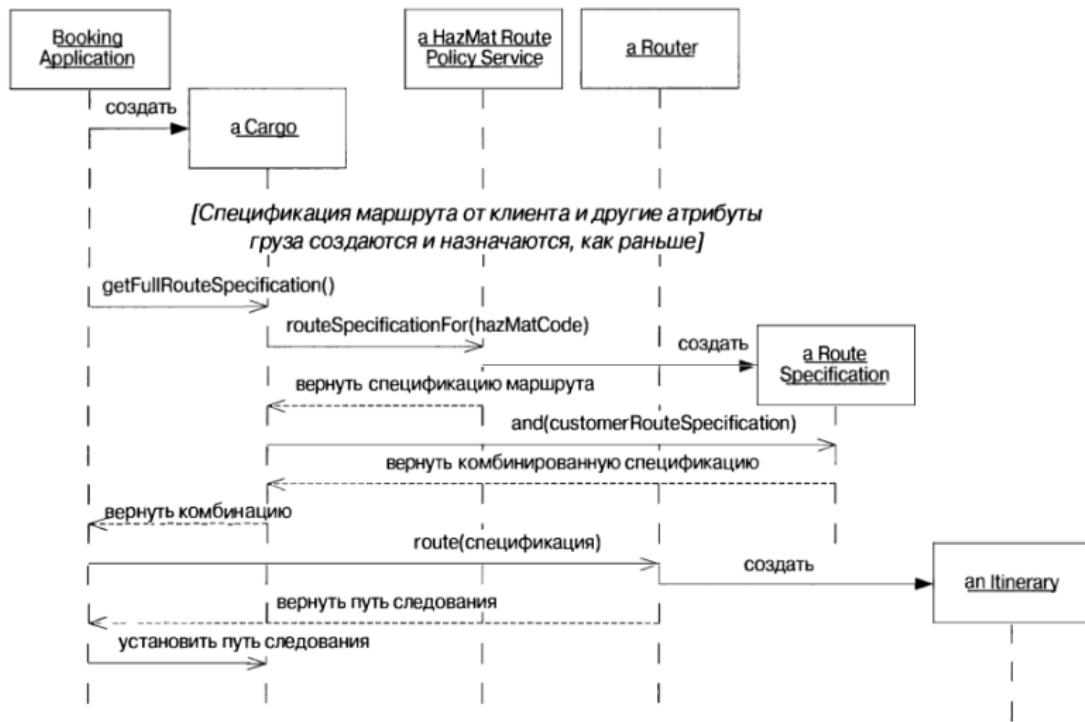


Диаграмма последовательностей

Работа с опасными грузами, первая версия



Работа с опасными грузами, вторая версия

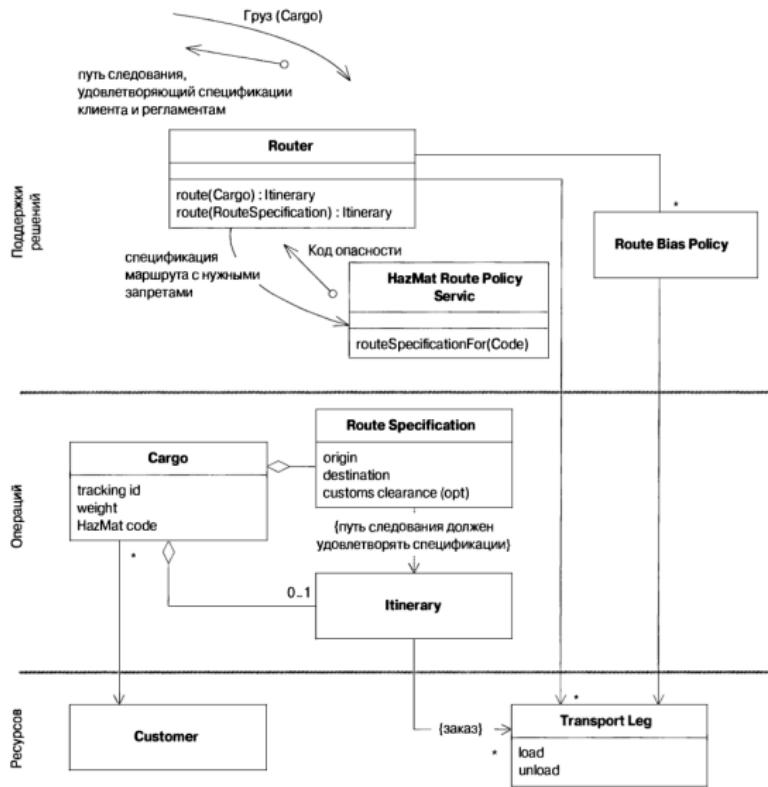
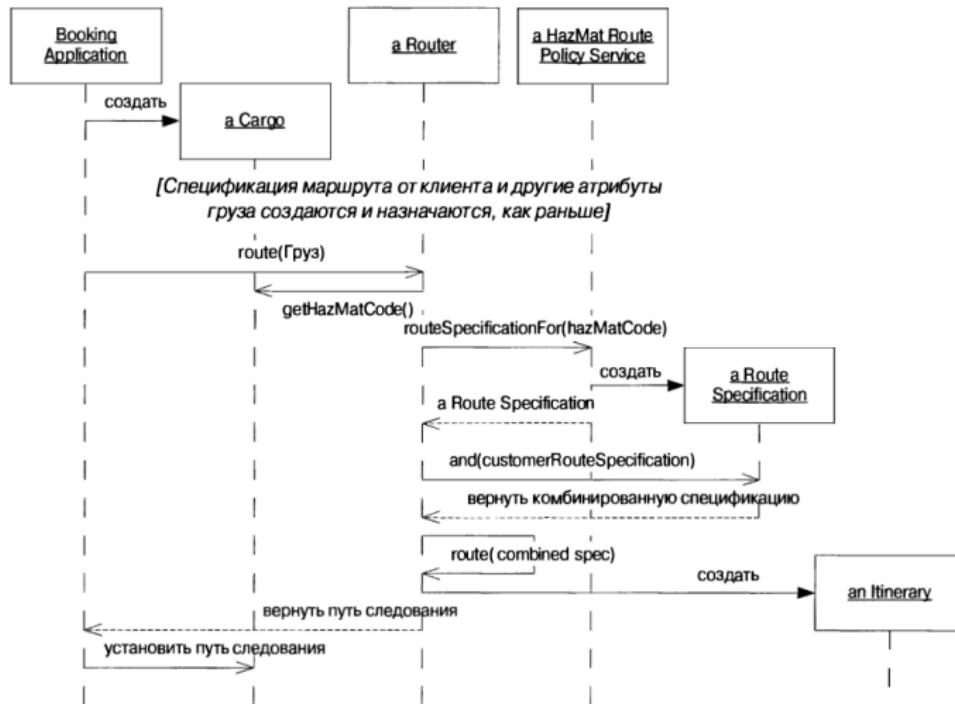


Диаграмма последовательностей

Работа с опасными грузами, вторая версия



Типичные уровни в системах автоматизации производства

Принятия решений	Аналитические механизмы	Практически отсутствует как состояние, так и его изменение	Анализ управления Оптимизация использования Сокращение рабочего цикла
Регламентный	Стратегии Связи-ограничения (на основании целей или закономерностей данной отрасли)	Медленное изменение состояния	Приоритет изделий Предписанные регламенты изготовления деталей
Операционный	Состояние, отражающее реальное положение дел (деятельности и планов)	Быстрое изменение состояния	Инвентарная опись Учет состояния незаконченных деталей
Потенциальный	Состояние, отражающее реальное положение дел (ресурсов)	Изменение состояния в среднем темпе	Возможности оборудования Наличие оборудования Перемещение по территории

зависимость

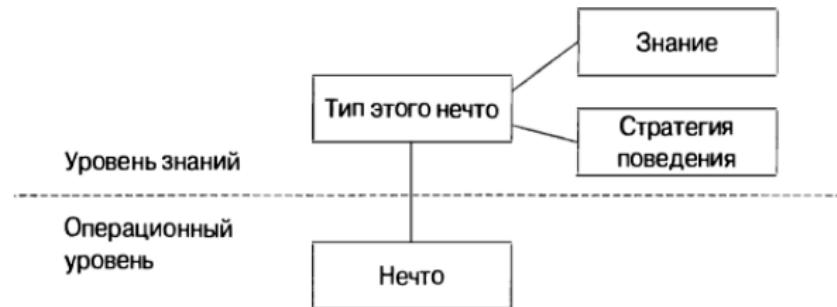
Типичные уровни в финансовых системах

Приятия решений	Аналитические механизмы	Практически отсутствует как состояние, так и его изменение	Анализ рисков Анализ портфелей Средства ведения переговоров
Регламентный	Стратегии Связи-ограничения (на основании целей или закономерностей данной отрасли)	Медленное изменение состояния	Пределы резервов Цели размещения активов
Обязательств	Состояние, отражающее сделки и договоры с клиентами	Изменение состояния в среднем темпе	Соглашения с клиентами Соглашения по синдикации
Операционный	Состояние, отражающее реальное положение дел (деятельности и планов)	Быстрое изменение состояния	Состояние кредитов Начисления Выплаты и распределения

↓
зависимость

Другие высокоуровневые структуры

- ▶ **Уровень знаний (Knowledge level)** использует информацию о типах сущностей, позволяя гибко переконфигурировать систему



- ▶ **Подключаемые компоненты (Pluggable Component Framework)** — стиль, описывающий общее ядро и набор взаимозаменяемых плагинов, которыми оно управляет
- ▶ Разные стили не исключают друг друга!

Лекция 13: Проектирование распределённых приложений

Часть первая: транспортные вопросы

Юрий Литвинов
yurii.litvinov@gmail.com

03.12.2020г

Распределённые системы

- ▶ Компоненты приложения находятся в компьютерной сети
- ▶ Взаимодействуют через обмен сообщениями
- ▶ Основное назначение — работа с общими ресурсами
- ▶ Особенности
 - ▶ Параллельная работа
 - ▶ Независимые отказы
 - ▶ Отсутствие единого времени

Частые заблуждения при проектировании распределённых систем

- ▶ Сеть надёжна
- ▶ Задержка (latency) равна нулю
- ▶ Пропускная способность бесконечна
- ▶ Сеть безопасна
- ▶ Топология сети неизменна
- ▶ Администрирование сети централизовано
- ▶ Передача данных “бесплатна”
- ▶ Сеть однородна

Архитектура распределённых систем

- ▶ Какие сущности взаимодействуют между собой в распределённой системе?
- ▶ Как они взаимодействуют?
- ▶ Какие (возможно изменяющиеся) роли и ответственности имеют эти сущности в рамках всей архитектуры?
- ▶ Как они размещаются на физическую инфраструктуру?

Виды сущностей

- ▶ Узлы-процессы-потоки — сущности уровня ОС (или сами вычислительные узлы, если ОС не поддерживает даже процессы)
- ▶ Объекты — обычные объекты из ООП, с интерфейсами, описанными на IDL, вызывающие друг друга по сети
- ▶ Компоненты — более высокоуровневые сущности, как правило, предполагают middleware
- ▶ Веб-сервисы — ещё более высокоуровневые сущности, независимые приложения с чётко определённым способом их использовать

Виды взаимодействия

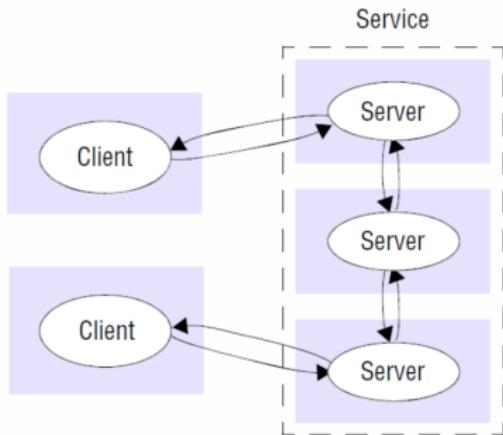
- ▶ Межпроцессное взаимодействие
- ▶ Удалённые вызовы
 - ▶ Протоколы вида “запрос-ответ”
 - ▶ Удалённые вызовы процедур (remote procedure calls, RPC)
 - ▶ Удалённые вызовы методов (remote method invocation, RMI)
- ▶ Неявное взаимодействие
 - ▶ Групповое взаимодействие
 - ▶ Модель “издатель-подписчик”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

Роли и обязанности

- ▶ Клиент-сервер
- ▶ Peer-to-peer

Варианты размещения

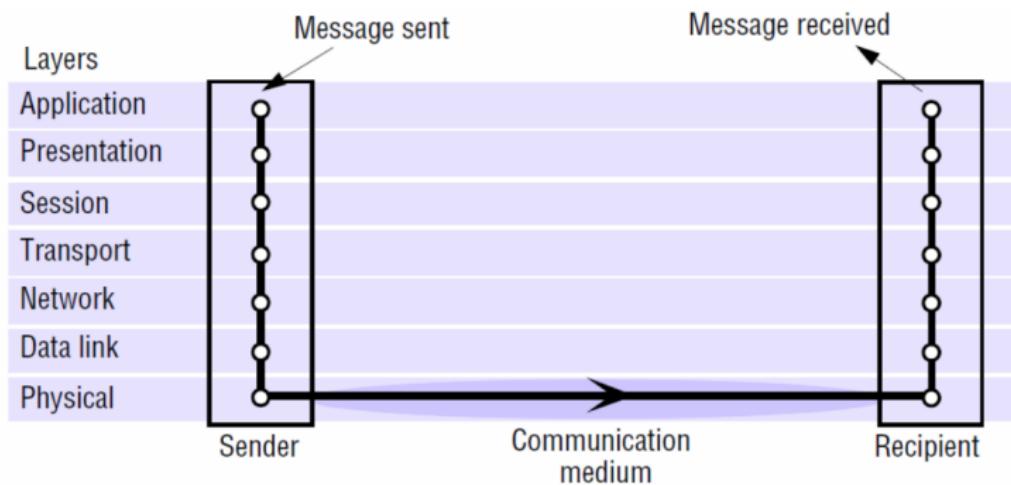
- ▶ Разбиение сервисов по нескольким серверам
- ▶ Кэширование
- ▶ Мобильный код
- ▶ Мобильный агент



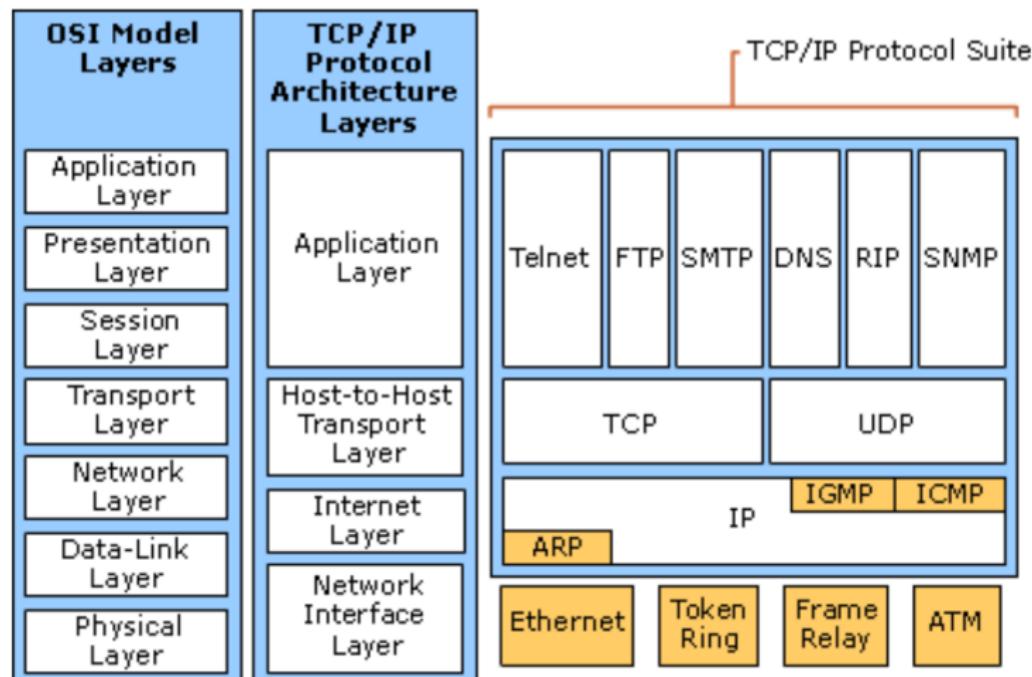
Типичные архитектурные стили

- ▶ Уровневая архитектура
 - ▶ ОС
 - ▶ Коммуникационная инфраструктура (Middleware)
 - ▶ Приложения и сервисы
- ▶ Клиент-сервер
 - ▶ Тонкий клиент
 - ▶ Бизнес-логика и данные — на сервере
- ▶ Трёхзвенная и N-уровневая архитектуры
 - ▶ Бизнес-логику и работу с данными часто разделяют

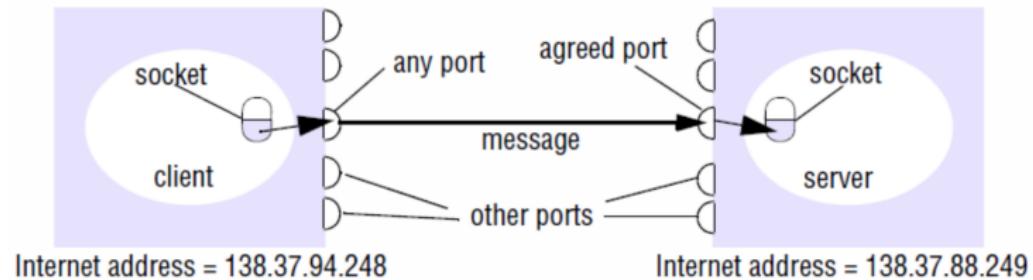
Модель OSI



Стек протоколов TCP/IP

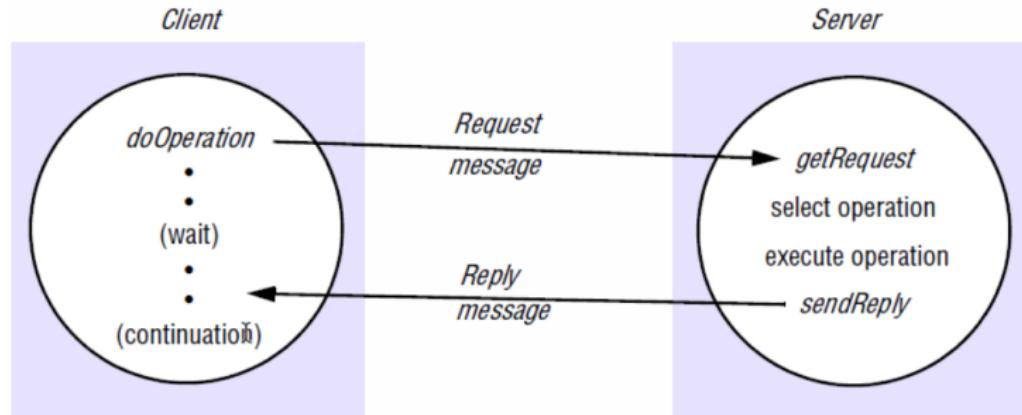


Абстракция сокета



Протоколы “запрос-ответ”

- ▶ Запрос, действие, ответ
- ▶ Преимущественно синхронные вызовы



“Запрос-ответ” поверх UDP

- + Уведомления не нужны
- + Установление соединения — в два раза больше сообщений
- + Управление потоком не имеет смысла
- Потери пакетов
 - ▶ Таймаут + повторный запрос на уровне бизнес-логики
 - ▶ Защита от повторного выполнения операции (хранение “истории”)
 - ▶ Новый запрос как подтверждение получения прошлого
- Неопределённый порядок пакетов

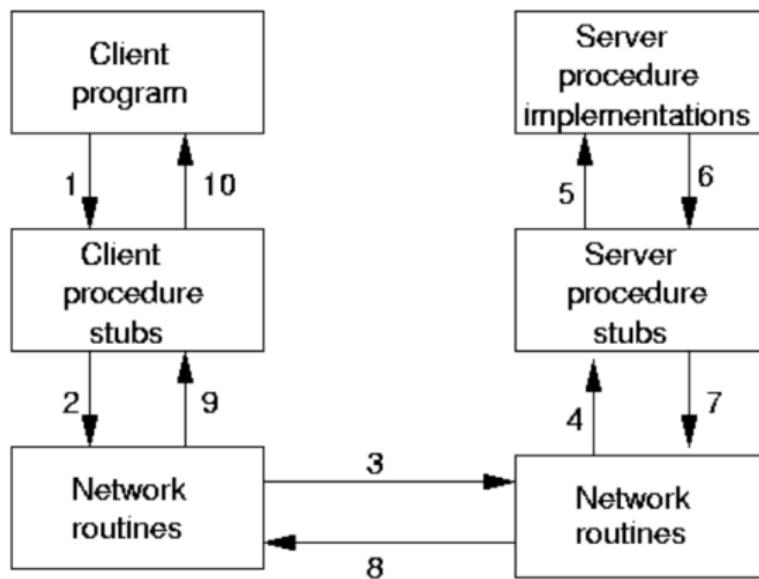
“Запрос-ответ” поверх TCP

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

HTTP

- ▶ Пример протокола “запрос-ответ”
- ▶ Реализован поверх TCP
- ▶ Соединение на всё время взаимодействия
- ▶ Маршалинг данных в ASCII
 - ▶ MIME
- ▶ HTTP 2.0
 - ▶ Бинарный протокол
 - ▶ Обязательное шифрование
 - ▶ Мультиплексирование запросов в одном TCP соединении
 - ▶ “Предсказывающая посылка данных”

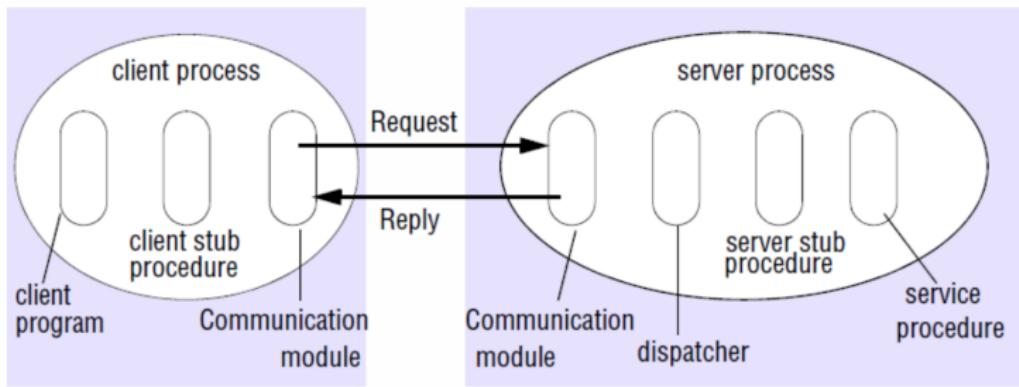
RPC



Прозрачность RPC-вызовов

- ▶ Изначальная цель — максимальная похожесть на обычные вызовы
 - ▶ Location and access transparency
- ▶ Удалённые вызовы более уязвимы к отказам
 - ▶ Нужно понимать разницу между отказом сети и отказом сервиса
 - ▶ Exponential backoff
 - ▶ Клиенты должны знать о задержках при передаче данных
 - ▶ Возможность прервать вызов
- ▶ Явная маркировка удалённых вызовов?
 - ▶ Прозрачность синтаксиса
 - ▶ Явное отличие в интерфейсах
 - ▶ Указание сематики вызова

Структура RPC middleware

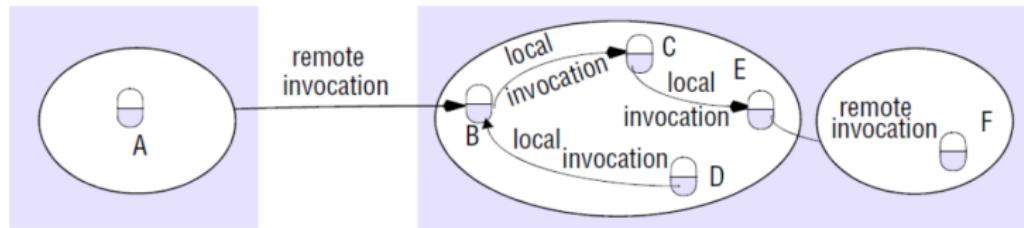


Удалённые вызовы методов (RMI)

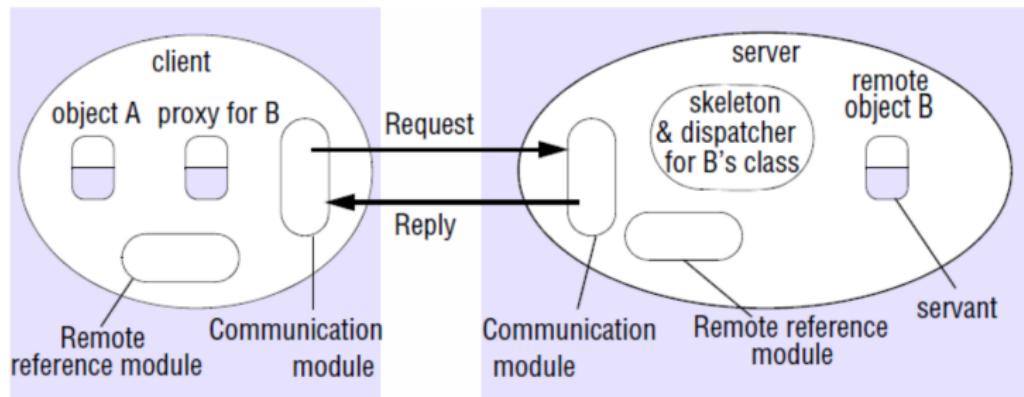
- ▶ Продолжение идей RPC
 - ▶ Программирование через интерфейсы
 - ▶ Работа поверх протоколов “запрос-ответ”
 - ▶ At-least-once или at-most-once семантика вызовов
 - ▶ Прозрачность синтаксиса вызовов
- ▶ Особенности ОО-программ
 - ▶ Наследование, полиморфизм
 - ▶ Передача параметров по ссылкам
 - ▶ Исключения
 - ▶ Распределённая сборка мусора

Локальные и удалённые вызовы

- ▶ Локальные и удалённые объекты
- ▶ Интерфейсы удалённых объектов
- ▶ Ссылки на удалённые объекты
 - ▶ Как параметры или результаты удалённых вызовов



Структура RMI middleware



Protocol buffers

protobuf

- ▶ Механизм сериализации-десериализации данных
- ▶ Компактное бинарное представление
- ▶ Декларативное описание формата данных, генерация кода для языка программирования
 - ▶ Поддерживается Java, Python, Objective-C, C++, Go, JavaNano, Ruby, C#
- ▶ Бывает v2 и v3, с некоторыми синтаксическими различиями
- ▶ Хитрый протокол передачи,
<https://developers.google.com/protocol-buffers/docs/encoding>
 - ▶ До 10 раз компактнее XML

Пример

Файл .proto:

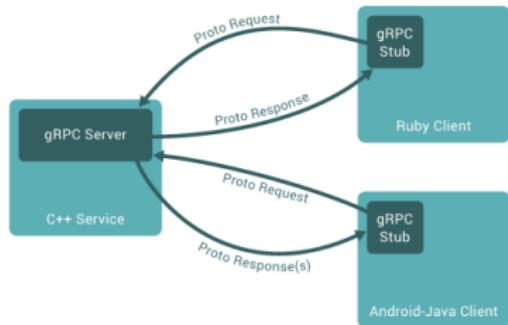
```
message Person {  
    required string name = 1;  
    required int32 id = 2;  
    optional string email = 3;  
}
```

Файл .java:

```
Person john = Person.newBuilder()  
    .setId(1234)  
    .setName("John Doe")  
    .setEmail("jdoe@example.com")  
    .build();  
  
output = new FileOutputStream(args[0]);  
john.writeTo(output);
```

gRPC

- ▶ средство для удалённого вызова (RPC)
- ▶ Работает поверх protobuf
- ▶ Разрабатывается Google
- ▶ Поддерживает C++, Java, Objective-C, Python, Ruby, Go, C#, Node.js



Технические подробности

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

```
service RouteGuide {  
    rpc GetFeature(Point) returns (Feature) {}  
    rpc ListFeatures(Rectangle) returns (stream Feature) {}  
    rpc RecordRoute(stream Point) returns (RouteSummary) {}  
    rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}  
}
```

- ▶ Сборка — плагином grpc к protoc

Реализация сервиса на Java

```
private static class RouteGuideService extends RouteGuideGrpc.RouteGuideImplBase {  
    ...  
    @Override  
    public void getFeature(Point request, StreamObserver<Feature> responseObserver) {  
        responseObserver.onNext(checkFeature(request));  
        responseObserver.onCompleted();  
    }  
  
    @Override  
    public void listFeatures(Rectangle request, StreamObserver<Feature> responseObserver) {  
        for (Feature feature : features) {  
            ...  
            int lat = feature.getLocation().getLatitude();  
            int lon = feature.getLocation().getLongitude();  
            if (lon >= left && lon <= right && lat >= bottom && lat <= top) {  
                responseObserver.onNext(feature);  
            }  
        }  
        responseObserver.onCompleted();  
    }  
}
```

Реализация сервиса на Java (2)

```
@Override
public StreamObserver<RouteNote> routeChat(
    final StreamObserver<RouteNote> responseObserver) {
    return new StreamObserver<RouteNote>() {
        @Override
        public void onNext(RouteNote note) {
            List<RouteNote> notes = getOrCreateNotes(note.getLocation());
            for (RouteNote prevNote : notes.toArray(new RouteNote[0])) {
                responseObserver.onNext(prevNote);
            }
            notes.add(note);
        }
        @Override
        public void onError(Throwable t) {
            logger.log(Level.WARNING, "routeChat cancelled");
        }
        @Override
        public void onCompleted() {
            responseObserver.onCompleted();
        }
    };
}
```

Реализация клиента на Java (1)

```
public RouteGuideClient(String host, int port) {  
    this(ManagedChannelBuilder.forAddress(host, port).usePlaintext(true));  
}  
  
public RouteGuideClient(ManagedChannelBuilder<?> channelBuilder) {  
    channel = channelBuilder.build();  
    blockingStub = RouteGuideGrpc.newBlockingStub(channel);  
    asyncStub = RouteGuideGrpc.newStub(channel);  
}
```

Реализация клиента на Java (2)

```
public void getFeature(int lat, int lon) {
    Point request = Point.newBuilder().setLatitude(lat).setLongitude(lon).build();
    Feature feature;
    try {
        feature = blockingStub.getFeature(request);
    } catch (StatusRuntimeException e) {
        warning("RPC failed: {0}", e.getStatus());
        return;
    }
    if (RouteGuideUtil.exists(feature)) {
        info("Found feature called \"{0}\" at {1}, {2}",
            feature.getName(),
            RouteGuideUtil.getLatitude(feature.getLocation()),
            RouteGuideUtil.getLongitude(feature.getLocation()));
    } else {
        info("Found no feature at {0}, {1}",
            RouteGuideUtil.getLatitude(feature.getLocation()),
            RouteGuideUtil.getLongitude(feature.getLocation()));
    }
}
```

Веб-сервисы

- ▶ Перенос специализации клиент-сервера в web
- ▶ Сложные приложения как интеграция веб-сервисов
- ▶ HTTP-запрос для выполнения команды
 - ▶ Асинхронное взаимодействие
 - ▶ Ответ-запрос
 - ▶ Событийные схемы
- ▶ XML или JSON как основной формат сообщений
 - ▶ SOAP/WSDL/UDDI
 - ▶ XML-RPC
 - ▶ REST

Лекция 14: Проектирование распределённых приложений

Часть вторая: высокоуровневые вещи

Юрий Литвинов
yurii.litvinov@gmail.com

10.12.2020г

Docker

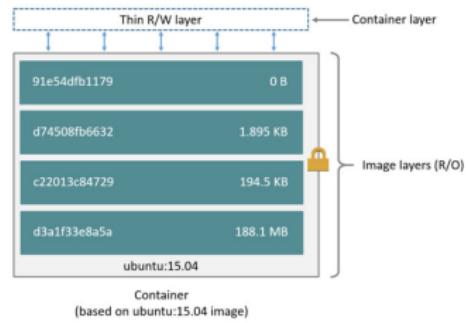
- ▶ Средство для “упаковки” приложений в изолированные контейнеры
- ▶ Что-то вроде легковесной виртуальной машины
- ▶
- ▶ Широкий инструментарий: DSL для описания образов, публичный репозиторий, поддержка оркестраторами



© <https://www.docker.com>

Docker Image

- ▶ Окружение и приложение
- ▶ Состоит из слоёв
 - ▶ Все слои read-only
 - ▶ Образы делят слои между собой как процессы делят динамические библиотеки
- ▶ На основе одного образа можно создать другой



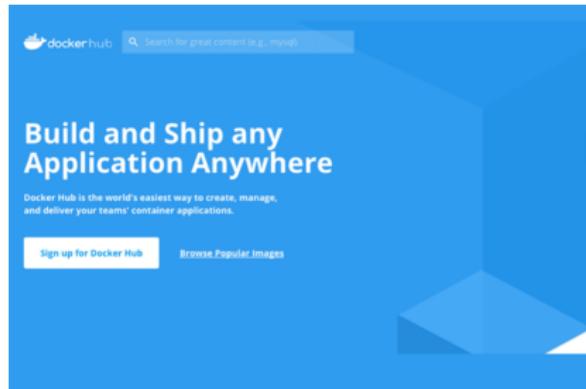
Docker Container

- ▶ Образ с дополнительным write слоем
- ▶ Содержит один запущенный процесс
- ▶ Может быть сохранен как новый образ



DockerHub

- ▶ Внешний репозиторий образов
 - ▶ Официальные образы
 - ▶ Пользовательские образы
 - ▶ Приватные репозитории
- ▶ Простой CI/CD
- ▶ Высокая доступность



Базовые команды

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

Dockerfile

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

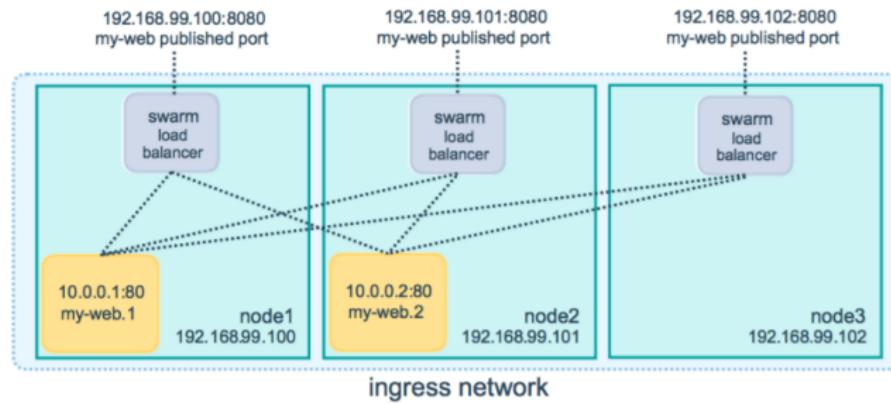
Балансировка нагрузки

docker-compose.yml

```
version: "3"
services:
  web:
    # replace username/repo:tag with your name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "80:80"
  networks:
    - webnet
networks:
  webnet:
```

Swarm-ы

- ▶ Машина, на которой запускается контейнер, становится главной
- ▶ Другие машины могут присоединяться к swarm-у и получать копию контейнера
- ▶ Docker балансирует нагрузку по машинам



© <https://www.docker.com>

SOAP-ориентированные сервисы

- ▶ Simple Object Access Protocol
- ▶ Web Services Description Language
- ▶ Universal Discovery, Description and Integration



SOAP-сообщение

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://example.org/alert">
      <m:msg>Get up at 6:30 AM</m:msg>
    </m:alert>
  </env:Body>
</env:Envelope>
```

WSDL-описание

```
<message name="getTermRequest">
    <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
    <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
    <operation name="getTerm">
        <input message="getTermRequest"/>
        <output message="getTermResponse"/>
    </operation>
</portType>
```

Достоинства SOAP-based сервисов

- ▶ Автоматический режим описания сервисов
- ▶ Автоматическая поддержка описаний SOAP-клиентом
- ▶ Автоматическая валидация сообщений
 - ▶ Валидность xml
 - ▶ Проверка по схеме
 - ▶ Проверка SOAP-сервером
- ▶ Работа через HTTP
 - ▶ Хоть через обычный GET

Недостатки SOAP-based сервисов

- ▶ Огромный размер сообщений
- ▶ Сложность описаний на клиенте и сервере
- ▶ Один запрос — один ответ
 - ▶ Поддержка транзакций на уровне бизнес-логики
- ▶ Сложности миграции при изменении описания

Пример: WCF

- ▶ Платформа для создания веб-сервисов
- ▶ Часть .NET Framework, начиная с 3.0
- ▶ Умеет WSDL, SOAP и т.д., очень конфигурируема
- ▶ Автоматическая генерация заглушек на стороне клиента
- ▶ ABCs of WCF:
 - ▶ Address
 - ▶ Binding
 - ▶ Contract



© <http://www.c-sharpcorner.com>



Пример, описание контракта

```
[ServiceContract(Namespace = "http://Microsoft.ServiceModel.Samples")]
public interface ICalculator
{
    [OperationContract]
    double Add(double n1, double n2);

    [OperationContract]
    double Subtract(double n1, double n2);

    [OperationContract]
    double Multiply(double n1, double n2);

    [OperationContract]
    double Divide(double n1, double n2);
}
```

Пример, реализация контракта

```
public class CalculatorService : ICalculator
{
    public double Add(double n1, double n2)
        => n1 + n2;

    public double Subtract(double n1, double n2)
        => n1 - n2

    public double Multiply(double n1, double n2)
        => n1 * n2;

    public double Divide(double n1, double n2)
        => n1 / n2;
}
```

Пример, self-hosted service

```
static void Main(string[] args)
{
    Uri baseAddress = new Uri("http://localhost:8000/ServiceModelSamples/Service");
    ServiceHost selfHost = new ServiceHost(typeof(CalculatorService), baseAddress);

    try {
        selfHost.AddServiceEndpoint(typeof(ICalculator), new WSHttpBinding(), "CalculatorService");

        ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
        smb.HttpGetEnabled = true;
        selfHost.Description.Behaviors.Add(smb);

        selfHost.Open();
        Console.WriteLine("The service is ready. Press <ENTER> to terminate service.");
        Console.ReadLine();

        selfHost.Close();
    } catch (CommunicationException ce) {
        Console.WriteLine($"An exception occurred: {ce.Message}");
        selfHost.Abort();
    }
}
```

Пример, клиент

- ▶ Генерация заглушки:

```
svchost.exe /language:cs /out:generatedProxy.cs /config:app.config^
http://localhost:8000/ServiceModelSamples/service
```

- ▶ Клиент:

```
static void Main(string[] args)
{
    CalculatorClient client = new CalculatorClient();

    double value1 = 100.00D;
    double value2 = 15.99D;
    double result = client.Add(value1, value2);
    Console.WriteLine($"Add({value1},{value2}) = {result}");

    client.Close();
}
```

Пример, конфигурация клиента

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <!-- specifies the version of WCF to use-->
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5,Profile=Client" />
  </startup>
  <system.serviceModel>
    <bindings>
      <!-- Uses wsHttpBinding-->
      <wsHttpBinding>
        <binding name="WSHttpBinding_ICalculator" />
      </wsHttpBinding>
    </bindings>
    <client>
      <!-- specifies the endpoint to use when calling the service -->
      <endpoint address="http://localhost:8000/ServiceModelSamples/Service/CalculatorService"
        binding="wsHttpBinding" bindingConfiguration="WSHttpBinding_ICalculator"
        contract="ServiceReference1.ICalculator" name="WSHttpBinding_ICalculator">
        <identity>
          <userPrincipalName value="migree@redmond.corp.microsoft.com" />
        </identity>
      </endpoint>
    </client>
  </system.serviceModel>
</configuration>
```

Очереди сообщений

- ▶ Используются для гарантированной доставки сообщений
 - ▶ Даже если отправитель и получатель доступны в разное время
 - ▶ Локальное хранилище сообщений на каждом устройстве
- ▶ Реализуют модель “издатель-подписчик”, но могут работать и в режиме “точка-точка”
- ▶ Как правило, имеют развитые возможности маршрутизации, фильтрации и преобразования сообщений
 - ▶ Разветвители, агрегаторы, преобразователи порядка

RabbitMQ

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



Пример, отправитель

```
using System;
using RabbitMQ.Client;
using System.Text;

class Send
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using (var connection = factory.CreateConnection())
        {
            using (var channel = connection.CreateModel())
            {
                channel.QueueDeclare(queue: "hello", durable: false, exclusive: false,
                                      autoDelete: false, arguments: null);

                string message = "Hello World!";
                var body = Encoding.UTF8.GetBytes(message);

                channel.BasicPublish(exchange: "", routingKey: "hello",
                                      basicProperties: null, body: body);
            }
        }
    }
}
```

Пример, получатель

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Text;

class Receive
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using (var connection = factory.CreateConnection())
        using (var channel = connection.CreateModel())
        {
            channel.QueueDeclare(queue: "hello", durable: false, exclusive: false, autoDelete: false, arguments: null);

            var consumer = new EventingBasicConsumer(channel);
            consumer.Received += (model, ea) =>
            {
                var body = ea.Body;
                var message = Encoding.UTF8.GetString(body);
                Console.WriteLine(" [x] Received {0}", message);
            };
            channel.BasicConsume(queue: "hello", autoAck: true, consumer: consumer);
        }
    }
}
```

Representational State Transfer (REST)

- ▶ Модель клиент-сервер
- ▶ Отсутствие состояния
- ▶ Кэширование
- ▶ Единообразие интерфейса
- ▶ Слои

Интерфейс сервиса

- ▶ Коллекции
 - ▶ `http://api.example.com/resources/`
- ▶ Элементы
 - ▶ `http://api.example.com/resources/item/17`
- ▶ HTTP-методы
 - ▶ GET
 - ▶ PUT
 - ▶ POST
 - ▶ DELETE
- ▶ Передача параметров прямо в URL
 - ▶ `http://api.example.com/resources?user=me&access_token=ASFQF`

Пример, Google Drive REST API

- ▶ GET <https://www.googleapis.com/drive/v2/files> — список всех файлов
- ▶ GET <https://www.googleapis.com/drive/v2/files/fileId> — метаданные файла по его Id
- ▶ POST <https://www.googleapis.com/upload/drive/v2/files> — загрузить новый файл
- ▶ PUT <https://www.googleapis.com/upload/drive/v2/files/fileId> — обновить файл
- ▶ DELETE <https://www.googleapis.com/drive/v2/files/fileId> — удалить файл

Достоинства

- ▶ Надёжность
- ▶ Производительность
- ▶ Масштабируемость
- ▶ Прозрачность системы взаимодействия
- ▶ Простота интерфейсов
- ▶ Портативность компонентов
- ▶ Лёгкость внесения изменений

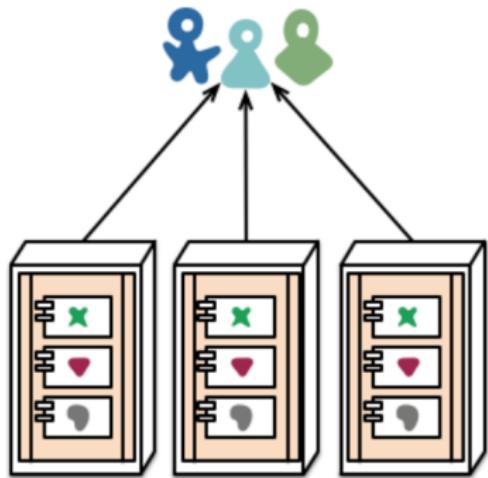
Микросервисы

- ▶ Набор небольших сервисов
 - ▶ Разные языки и технологии
- ▶ Каждый в собственном процессе
 - ▶ Независимое развёртывание
 - ▶ Децентрализованное управление
- ▶ Легковесные коммуникации

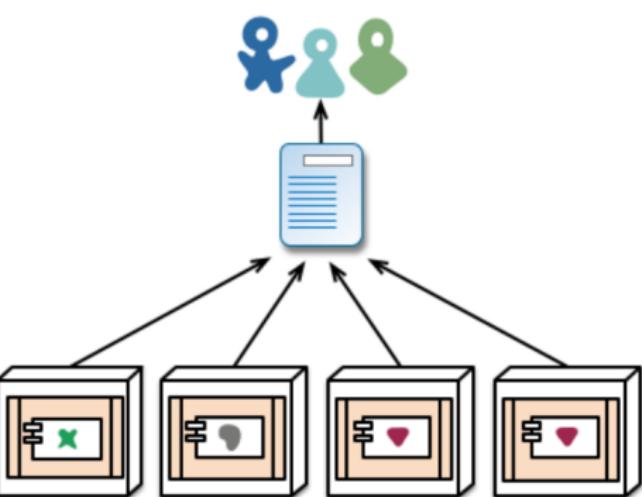
Монолитные приложения

- ▶ Большой и сложный MVC
- ▶ Единый процесс разработки и стек технологий
- ▶ Сложная архитектура
- ▶ Сложно масштабировать
- ▶ Сложно вносить изменения

Разбиение на сервисы



monolith - multiple modules in the same process



microservices - modules running in different processes

Основные особенности

- ▶ Микросервисы и SOA
- ▶ Smart endpoints and dumb pipes
- ▶ Проектирование под отказ
- ▶ Асинхронные вызовы
- ▶ Децентрализованное управление данными
- ▶ Автоматизация инфраструктуры
- ▶ Эволюционный дизайн

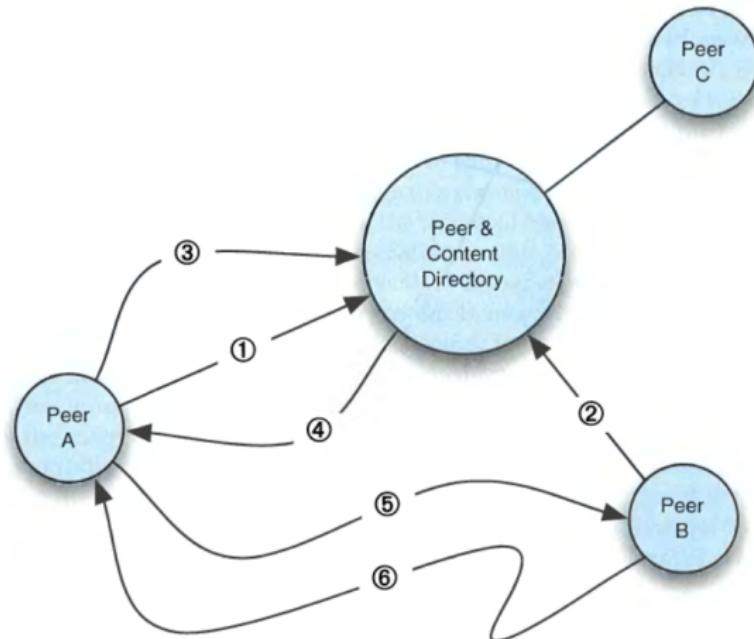
Основные проблемы

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

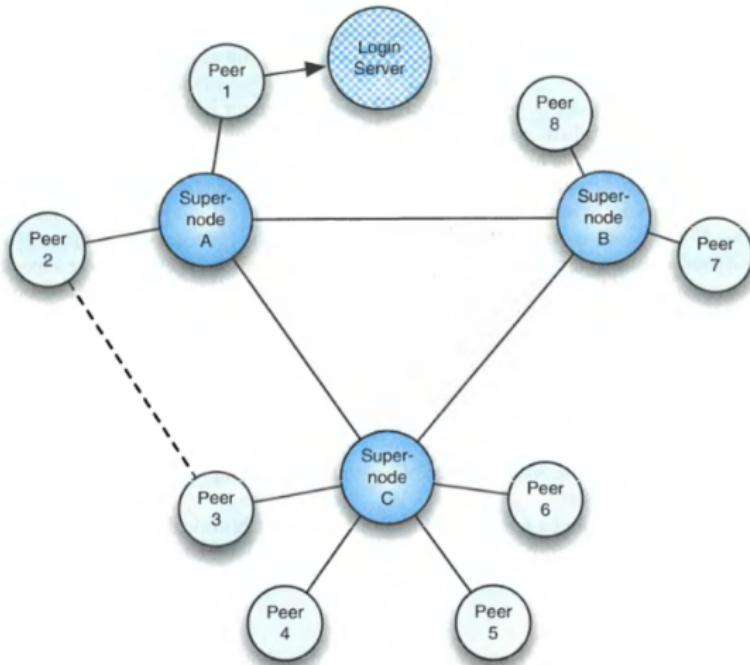
Архитектура Peer-to-Peer

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

Napster: hybrid client-server/P2P



Skype: Overlayed P2P



BitTorrent : Resource Trading P2P

- ▶ Обмен сегментами
- ▶ Поиск не входит в протокол
- ▶ Трекеры
- ▶ Метаданные
- ▶ Управление приоритетами
- ▶ Бестрекерная реализация

Лекция 15: Примеры архитектур

Юрий Литвинов

yurii.litvinov@gmail.com

17.12.2020г

Enterprise Fizz-Buzz

Задача:

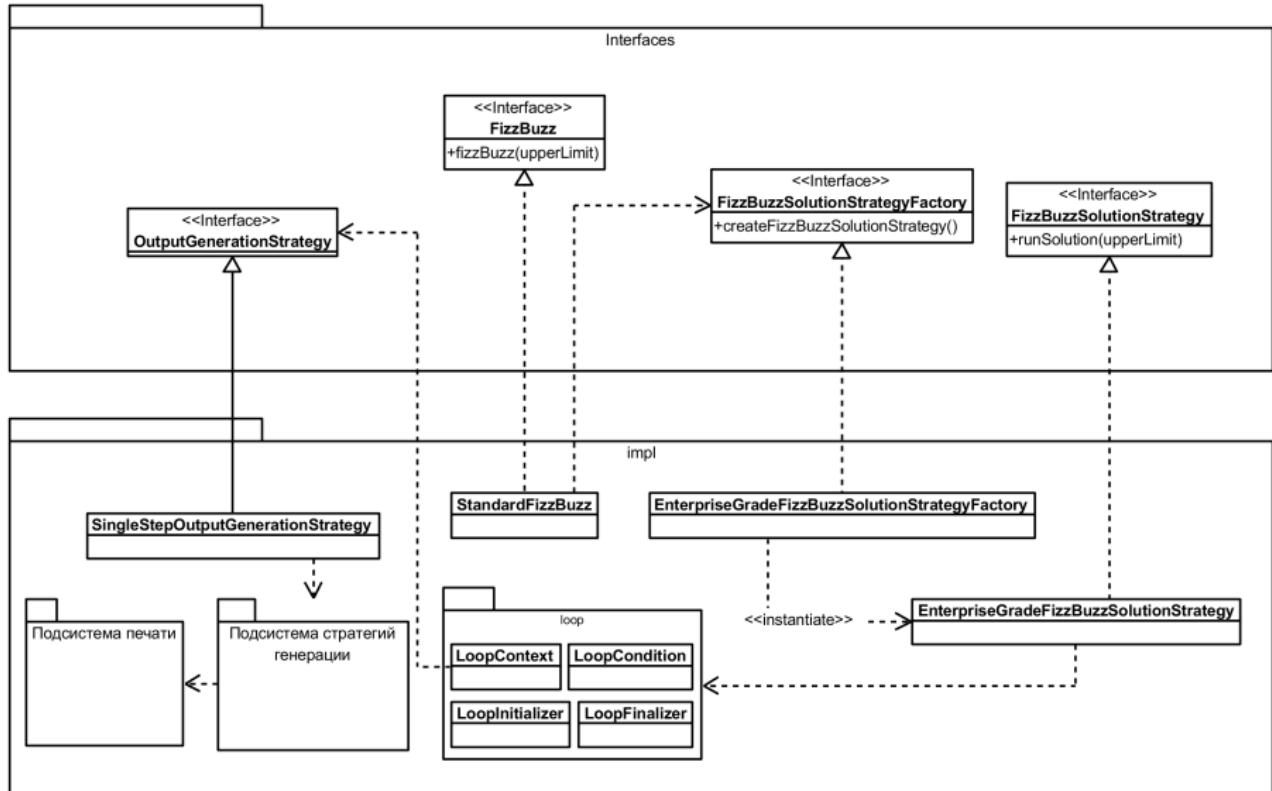
Для чисел от 1 до 100:

- ▶ если число делится на 3, вывести “Fizz”
- ▶ если число делится на 5, вывести “Buzz”
- ▶ если число делится и на 3, и на 5, вывести “FizzBuzz”
- ▶ во всех остальных случаях вывести само число

Решение:

<https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition>

Структура системы



Хорошие идеи

- ▶ Separation of Concerns
- ▶ Dependency Inversion
- ▶ Dependency Injection
 - ▶ Spring Framework
- ▶ Паттерны “Фабрика”, “Стратегия”, “Посетитель”, “Адаптер”, что-то вроде паттернов “Спецификация” и “Цепочка ответственности”

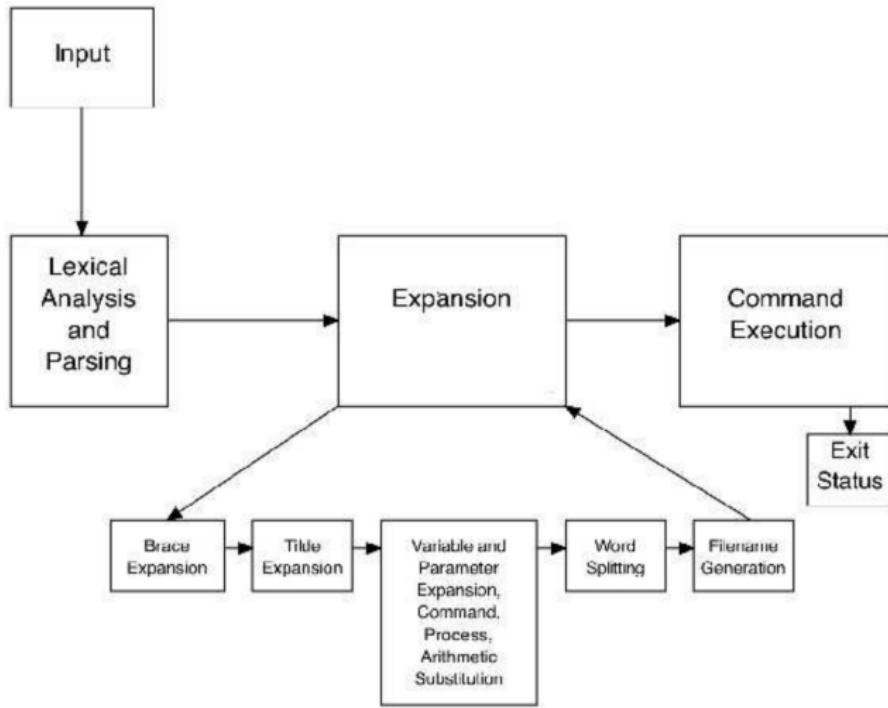
Плохие идеи

- ▶ Не выполняется принцип Keep It Simple Stupid
 - ▶ Неправильно говорить “строк кода написано”, правильно — “строк кода израсходовано”
- ▶ “Синтаксическое” разделение на пакеты, а не “семантическое”
 - ▶ Отсутствие модульности, антипаттерн “Big Ball of Mud”
- ▶ Хардкод основных параметров вычисления
- ▶ Нет юнит-тестов, только интеграционные; нет логирования
- ▶ 1663 строки кода и всего 40 строк комментариев
 - ▶ Отсутствие архитектурного описания

Bash

- ▶ Примерно 70К строк кода
- ▶ Исходный автор — Brian Fox, maintainer — Chet Ramey
- ▶ Первый релиз — 1989
- ▶ Написан на С
- ▶ Архитектурное описание — глава в *The Architecture of Open Source Applications*, написанная Chet Ramey

Архитектура Bash



Основные структуры данных

```
typedef struct word_desc {  
    char *word; /* Zero terminated string. */  
    int flags; /* Flags associated with this word. */  
} WORD_DESC;
```

```
typedef struct word_list {  
    struct word_list *next;  
    WORD_DESC *word;  
} WORD_LIST;
```

Ввод с консоли

- ▶ Библиотека Readline
 - ▶ независимая библиотека, но пишется в основном для Bash
- ▶ Цикл read/dispatch/execute/redisplay
- ▶ Dispatch table (или Keymap)
- ▶ Буфер редактирования, хитрый механизм расчёта действий для отображения
- ▶ Хранит все данные как 8-битные символы, но знает про Unicode

Синтаксический разбор

- ▶ Зависимый от контекста лексический анализ
for for in for; do for=for; done; echo \$for
- ▶ Использует lex + bison
- ▶ Подстановка alias-ов выполняется лексером
- ▶ Сохранение и восстановление состояния парсера

Подстановки

`${parameter:-word}`

раскрывается в *parameter*, если он установлен, и в *word*, если нет
`pre{one,two,three}post`

раскрывается в

`preonepost pretwopost prethreepost`

Ещё бывает подстановка тильды и арифметическая подстановка,
сопоставление шаблона

Исполнение команд

- ▶ Встроенные и внешние команды, обрабатываются единообразно
- ▶ Перенаправление ввода-вывода, отмена перенаправления
- ▶ Принимают набор слов
 - ▶ Иногда обрабатывают по-особому, например, присваивание в *export*
- ▶ Присваивание — тоже команда, но особая
- ▶ Перед запуском внешней команды — поиск в PATH, кеширование результатов
- ▶ Job control, foreground и background

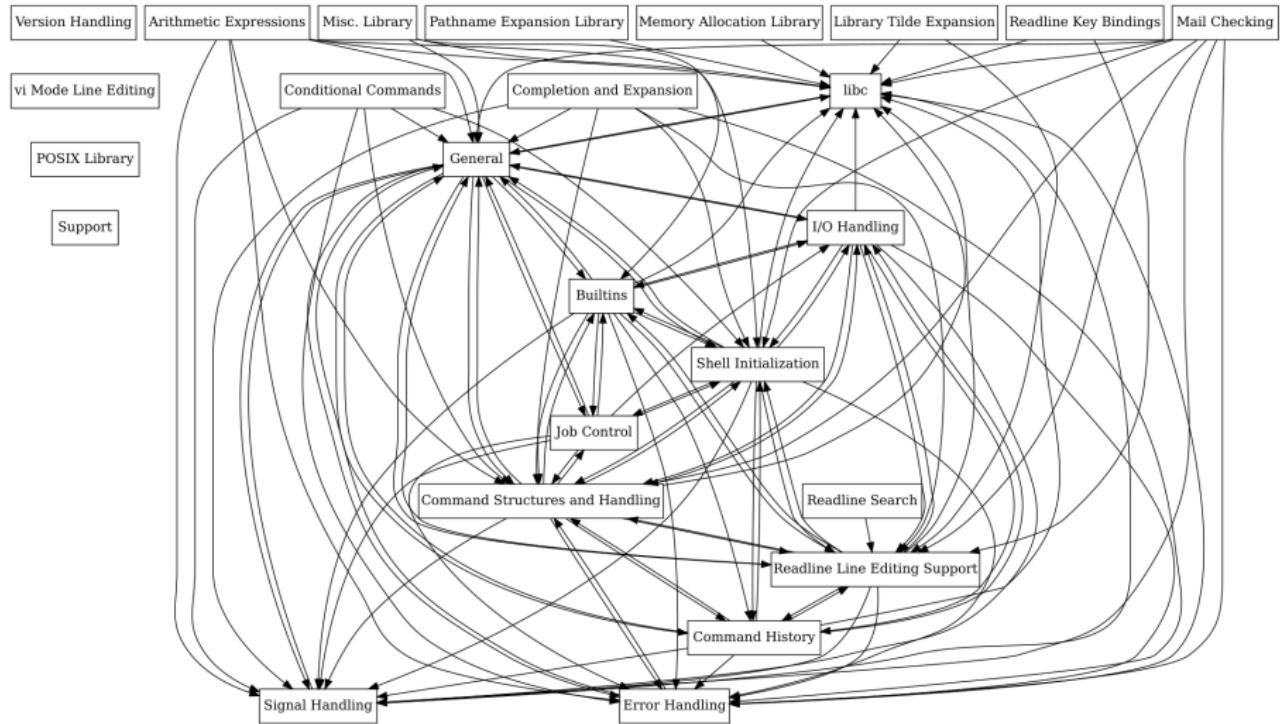
Lessons Learned

- ▶ Комментарии к коммитам со ссылками на багрепорты с шагами воспроизведения
- ▶ Хороший набор тестов, в Bash их тысячи
- ▶ Стандарты, как внешние на функциональность шелла, так и на код
- ▶ Пользовательская документация
- ▶ Переиспользование

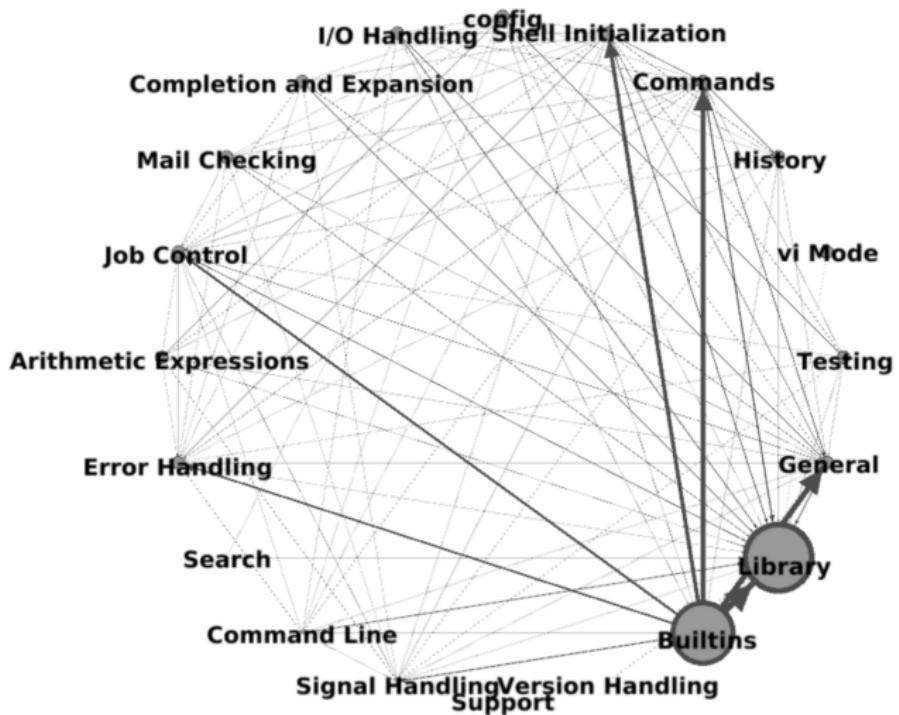
Архитектура Bash, на самом деле

- ▶ J. Garcia et al., *Obtaining Ground-Truth Software Architectures*
- ▶ 1 аспирант, 80 часов работы
- ▶ Верификация от Chet Ramey
- ▶ 70К строк кода, 200 файлов, 25 компонент
 - ▶ 16 — ядро, 9 — утилиты
- ▶ Структура папок почти не соответствует выделенным компонентам

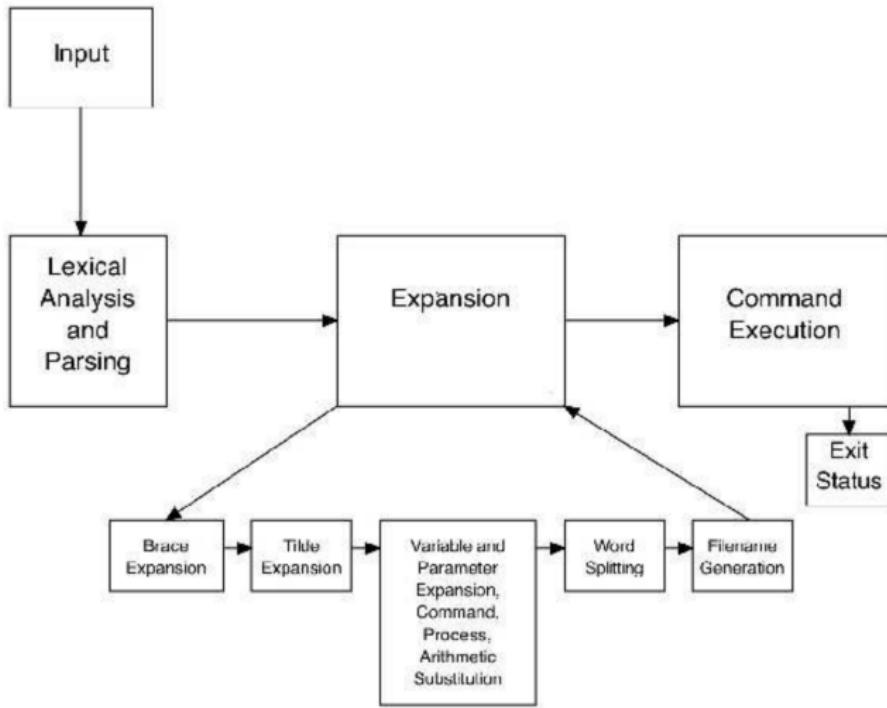
Архитектура Bash, на самом деле



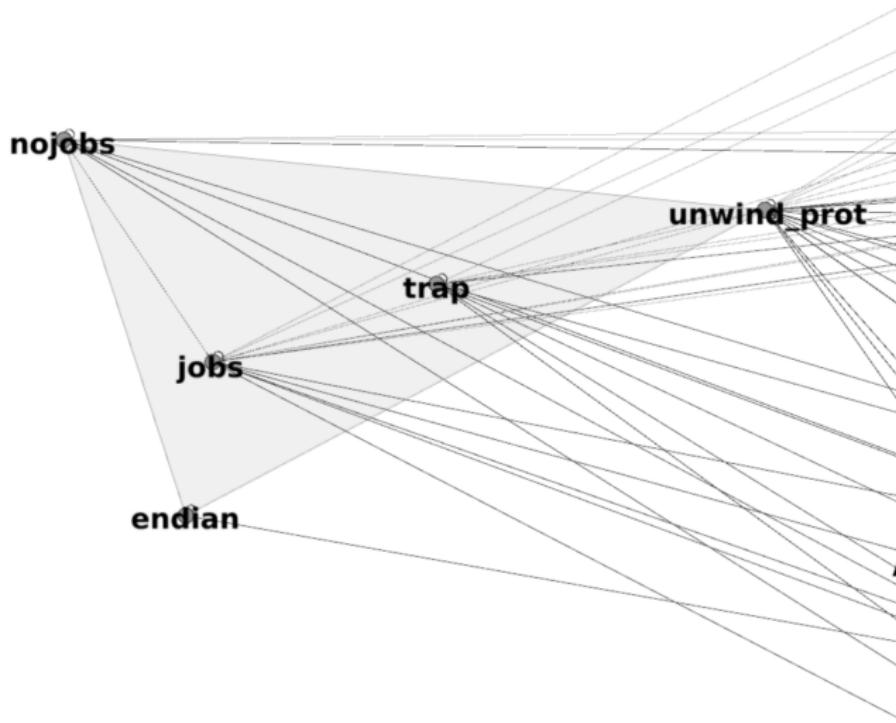
Результаты анализа кода



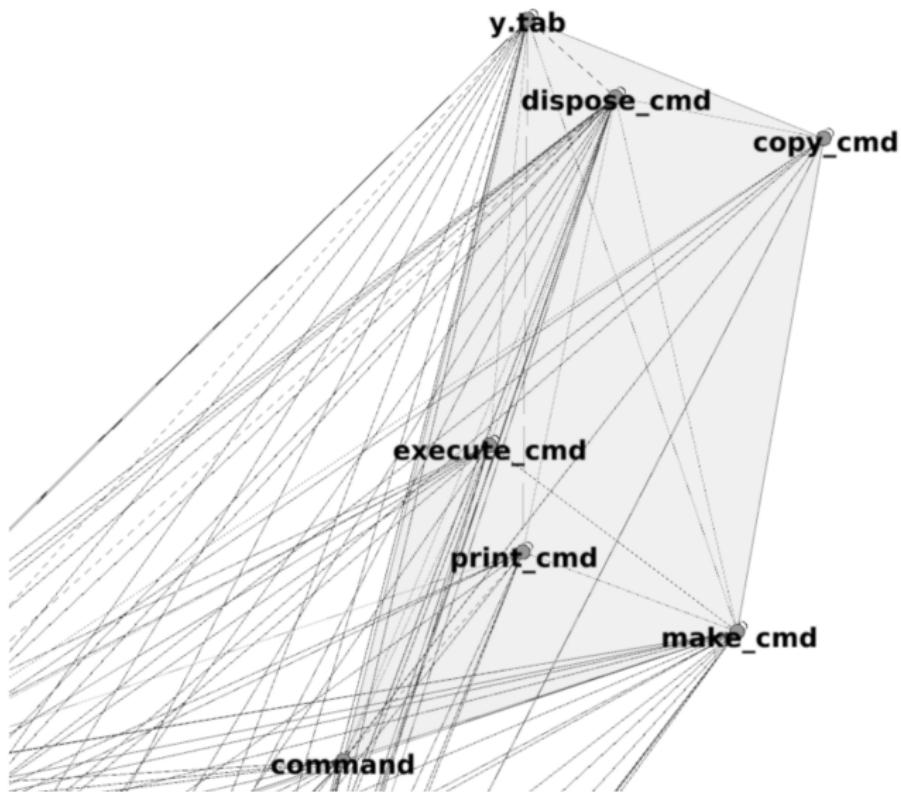
Сравним с исходной



Job Control



Commands



Git¹

- ▶ Распределённая VCS
- ▶ Linus Torvalds, 2005 год, драма с BitKeeper
- ▶ Architectural drivers
 - ▶ Распределённая разработка с тысячей коммитеров
 - ▶ Защита от порчи исходников
 - ▶ Возможность отменить мердж, смерджиться вручную
 - ▶ Высокая скорость работы

¹По гл. 10 <https://git-scm.com/book> и <http://aosabook.org/en/git.html>

Внутреннее устройство Git

Структура папки .git:

- ▶ HEAD
- ▶ index
- ▶ config
- ▶ description
- ▶ hooks/
- ▶ info/
- ▶ objects/
- ▶ refs/
- ▶ ...

Объекты

Git внутри — хеш-таблица, отображающая SHA-1-хеш файла в содержимое файла. Пример:

```
$ git init test
```

```
Initialized empty Git repository in /tmp/test/.git/
```

```
$ cd test
```

```
$ find .git/objects
```

```
.git/objects
```

```
.git/objects/info
```

```
.git/objects/pack
```

```
$ echo 'test content' | git hash-object -w --stdin
```

```
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

```
$ find .git/objects -type f
```

```
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Объекты (2)

Как получить сохранённый объект:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4  
test content
```

Версионный контроль:

```
$ echo 'version 1' > test.txt  
$ git hash-object -w test.txt  
83baae61804e65cc73a7201a7252750c76066a30  
$ echo 'version 2' > test.txt  
$ git hash-object -w test.txt  
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a  
$ find .git/objects -type f  
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a  
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30  
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Объекты (3)

Переключение между версиями файла:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 \  
  > test.txt
```

```
$ cat test.txt
```

```
version 1
```

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a \  
  > test.txt
```

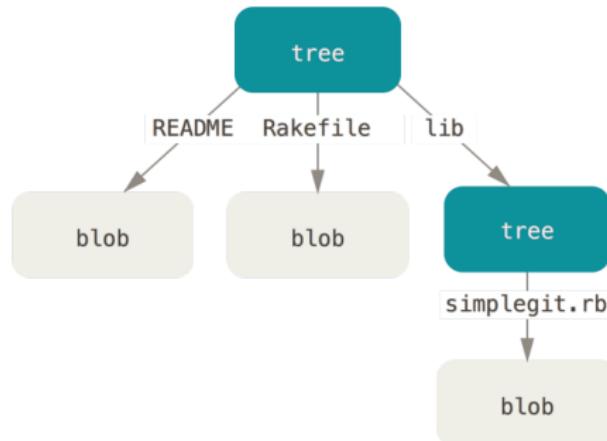
```
$ cat test.txt
```

```
version 2
```

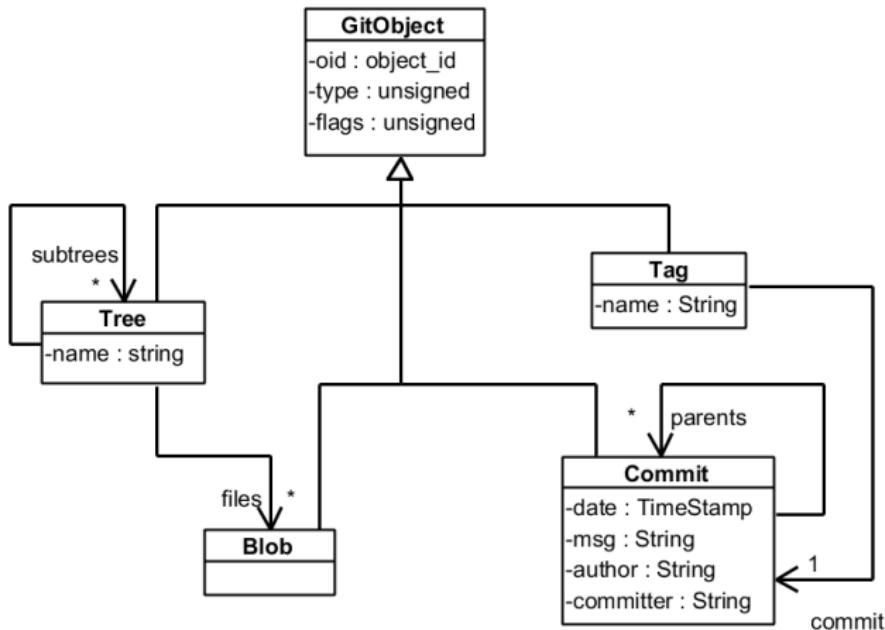
Деревья

blob (то, что мы видели раньше) хранит только содержимое файла, не хранит даже его имя. Решение проблемы — tree:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README
100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib
```



Какие ещё виды объектов бывают



Коммиты

tree-объекты могут хранить структуру файлов (как inode в файловой системе), но не хранят метаинформацию типа автора файла и даты создания. Это хранится в commit-объектах:

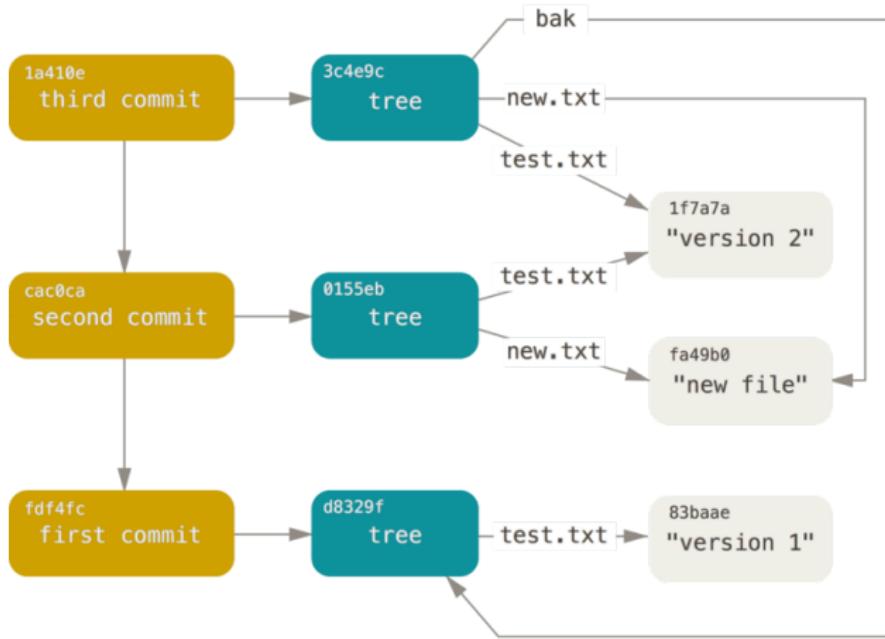
```
$ echo 'first commit' | git commit-tree d8329f  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

```
$ git cat-file -p fdf4fc3  
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
author Scott Chacon <schacon@gmail.com> 1243040974 -0700  
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700
```

first commit

Ещё коммит хранит список коммитов-родителей

Коммиты, как это выглядит



Ссылки

Теперь вся информация хранится на диске, но чтобы ей воспользоваться, нужно помнить SHA-1 хеши. На помощь приходят reference-ы.

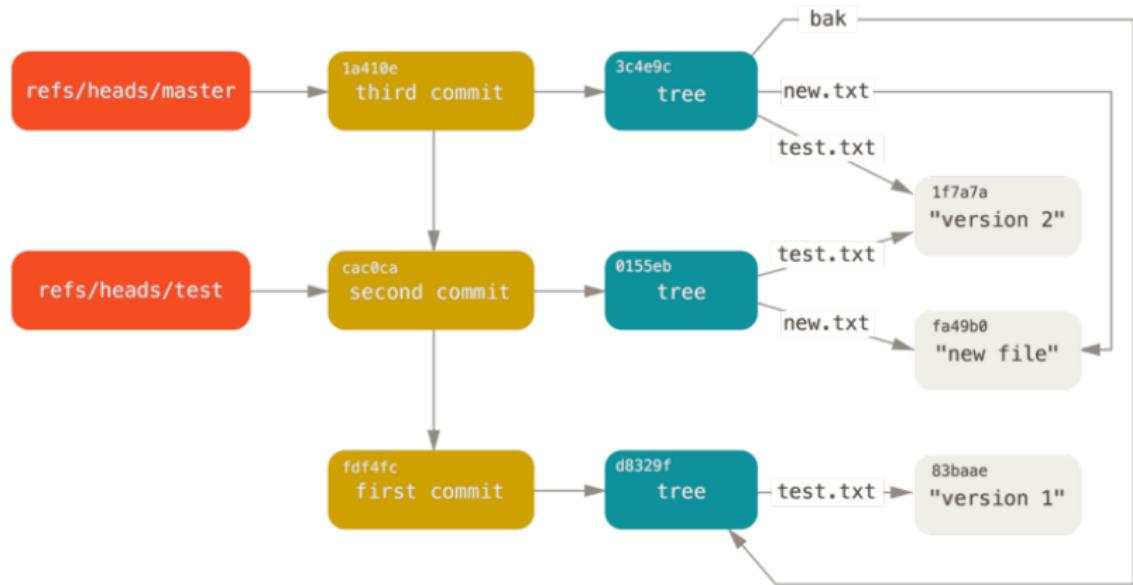
- ▶ .git/refs
- ▶ .git/refs/heads
- ▶ .git/refs/tags

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" \
> .git/refs/heads/master
```

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769ccbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

- ▶ Команда git update-ref

Ссылки, как это выглядит



HEAD

Теперь не надо помнить хеши, но как переключаться между ветками?

Текущая ветка хранится в HEAD. HEAD — символьическая ссылка, то есть ссылка на другую ссылку.

```
$ cat .git/HEAD
```

```
ref: refs/heads/master
```

```
$ git symbolic-ref HEAD refs/heads/test
```

```
$ cat .git/HEAD
```

```
ref: refs/heads/test
```

Тэги

Последний из объектов в Git — tag. Это просто указатель на коммит.

- ▶ Легковесный тэг:

```
git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Или просто git tag

- ▶ Аннотированный тэг:

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
```

```
object 1a410efbd13591db07496601ebc7a059dd55cfe9
```

```
type commit
```

```
tag v1.1
```

```
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700
```

test tag

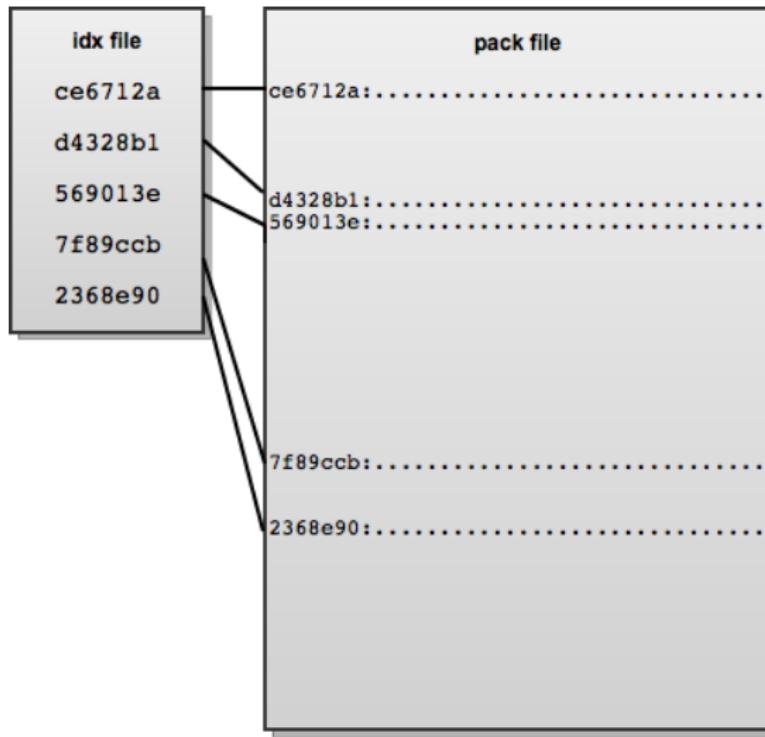
Packfiles

Пока что получалось, что все версии всех файлов в Git хранятся целиком, как они есть. Все они всегда сжимаются zlib, но в целом, если создать репозиторий, добавлять туда файлы, коммитить и т.д., все версии всех файлов будут в нём целиком. На помощь приходят .pack-файлы:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

Как оно устроено



Pack-файлы, подробности

- ▶ Упаковка происходит, когда:
 - ▶ Выполняется git push
 - ▶ Слишком много «свободных» объектов (порядка 7000)
 - ▶ Вручную вызвана git gc
- ▶ Используется дельта-компрессия
 - ▶ Последняя версия хранится целиком, дельты «идут назад»
- ▶ Можно заглянуть внутрь, git verify-pack
- ▶ Git может хитро перепаковывать pack-файлы

Reflog и восстановление коммитов

```
$ git reflog  
1a410ef HEAD@{0}: reset: moving to 1a410ef  
ab1afef HEAD@{1}: commit: modified repo.rb a bit  
484a592 HEAD@{2}: commit: added repo.rb
```

```
$ git log -g  
commit 1a410efbd13591db07496601ebc7a059dd55cfe9  
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)  
Reflog message: updating HEAD  
Author: Scott Chacon <schacon@gmail.com>  
Date: Fri May 22 18:22:37 2009 -0700
```

third commit

```
$ git branch recover-branch ab1afef
```

Как более капитально прострелить себе ногу

И что делать

```
$ git branch -D recover-branch  
$ rm -Rf .git/logs/
```

```
$ git fsck --full
```

Checking object directories: 100% (256/256), done.

Checking objects: 100% (18/18), done.

dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4

dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b

dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9

dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293

Git не удалит даже «висячие» объекты несколько месяцев, если его явно не попросить.

Lessons Learned

- ▶ Команды реализовывались как набор шелл-скриптов
 - ▶ Не портировать под Windows
 - ▶ Сложно интегрировать с IDE
 - ▶ Замедлило внедрение git-a
- ▶ Большой набор команд (включая plumbing) делает Git тяжёлым для изучения и усложняет сообщения об ошибках

Battle for Wesnoth²

- ▶ Пошаговая стратегия
- ▶ Порядка 200000 строк кода на C++
- ▶ 4 миллиона скачиваний
- ▶ 9/10 на Steam
- ▶ 2003 год



© <https://www.wesnoth.org/>

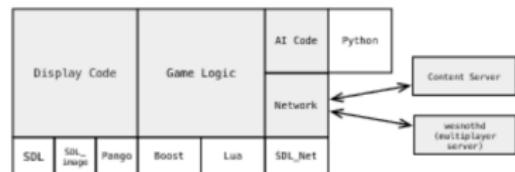
²По <http://aosabook.org>

Architectural Drivers

- ▶ Доступность для новых разработчиков и авторов контента
- ▶ В ущерб технической красоте
- ▶ Не nice to have, а условие выживания проекта в контексте широкого open-source сообщества из людей без каких-либо обязательств и разного технического уровня

Высокоуровневая архитектура

- ▶ Wesnoth Markup Language (WML)
- ▶ Минимизация зависимостей от сторонних библиотек
 - ▶ SDL (Simple Directmedia Layer) для видео и ввода/вывода
 - ▶ Простота использования и кроссплатформенность
 - ▶ Boost, Pango, zlib, Python, Lua, GNU gettext



Основные компоненты

- ▶ Парсер и препроцессор WML
- ▶ Базовый ввод-вывод — видео, звук, сеть
- ▶ GUI — виджеты
- ▶ Display module — игровая доска, юниты, анимация и т.д.
- ▶ ИИ
- ▶ Поиск пути (плюс утилиты для работы с гексагональной доской)
- ▶ Генератор карт
- ▶ Специализированные модули
 - ▶ Титульный экран
 - ▶ Storyline module — для проигрывания катсцен
 - ▶ Лобби — для мультиплеера
 - ▶ “Play game” module — управление основным игровым процессом
- ▶ Отдельно — wesnothd и content server

Wesnoth Markup Language

[unit_type]

```
id=Elvish Fighter
name=_ "Elvish Fighter"
image="units/elves-wood/fighter.png"
hitpoints=33
advances_to=Elvish Captain,Elvish Hero
{LESS_NIMBLE_ELF}
```

[attack]

```
name=sword
icon=attacks/sword-elven.png
range=melee
damage=5
```

[/attack]

[/unit_type]

Макросы

```
#define GOLD EASY_AMOUNT NORMAL_AMOUNT HARD_AMOUNT
#define EASY
gold={EASY_AMOUNT}
#endif
#define NORMAL
gold={NORMAL_AMOUNT}
#endif
#define HARD
gold={HARD_AMOUNT}
#endif
#endif
...
{GOLD 50 100 200}
```

Модель данных

- ▶ Всё сливается в один гигантский WML-документ
- ▶ Перезагружается при смене опций
- ▶ Всякие хаки на уровне препроцессора, чтобы не грузить вообще всё
- ▶ Классы unit и unit_type (архитектурный стиль Knowledge Layer)
- ▶ Фиксированный набор поддерживаемых движком атрибутов, задаваемых для каждого типа через WML
 - ▶ Нельзя описывать произвольное поведение через WML, хотели сохранить декларативность
- ▶ Класс attack_type
- ▶ Трейты, инвентарь

Мультиплеер

- ▶ Начальное состояние и команды
- ▶ Сервер просто пересыпает команды между клиентами
 - ▶ TCP/IP
- ▶ Replay
- ▶ Никакой защиты от читов
- ▶ Версии клиентов

Lessons Learned

- ▶ 250 тысяч строк на WML
- ▶ Сотни созданных пользователями кампаний
- ▶ 74 тысячи коммитов, 196 контрибуторов
- ▶ Сами разработчики смеются над WML
- ▶ В целом задача обеспечить доступность для модификации очень сложна