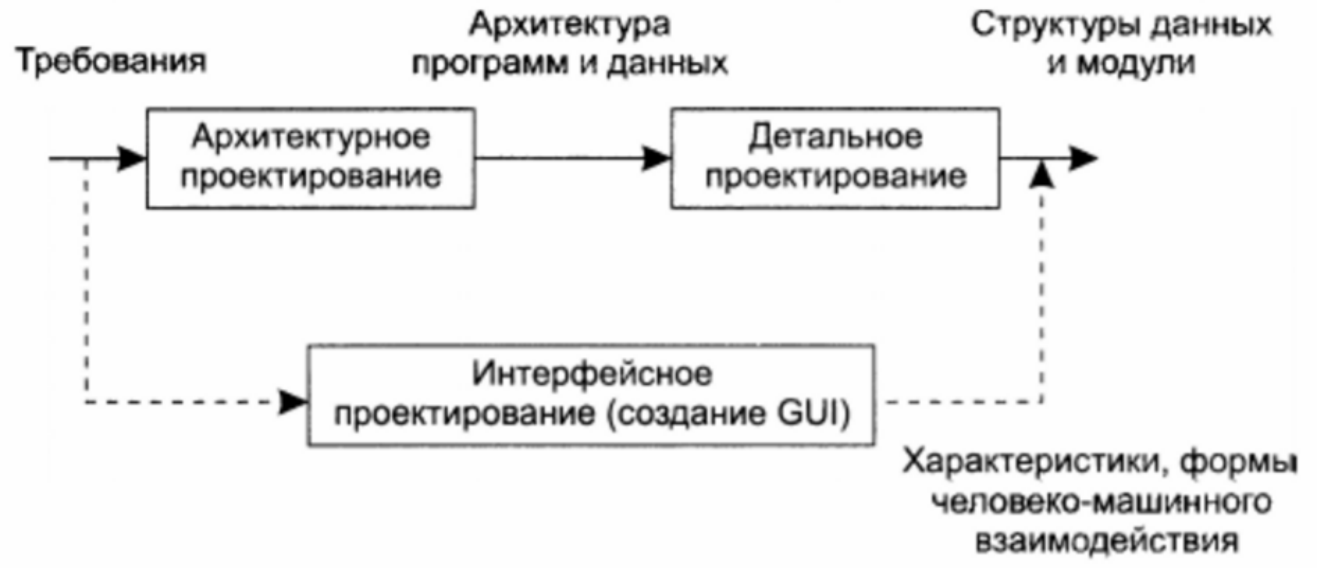
Проектирование ПО

Юрий Литвинов
yurii.litvinov@gmail.com

27.03.2017г

Проектирование ПО

Проектирование — деятельность, предшествующая разработке



Проектирование ПО

- Определение базового архитектурного стиля и структуры приложения
- Определение функционального поведения
- Осуществление декомпозиции на модули и планирование их взаимодействия
- Выбор стратегии и деталей реализации
  - Используется ли кодогенерация?
  - Используются ли сторонние фреймворки?
  - Используются ли сторонние библиотеки?
  - Какая часть функциональности будет реализовывать “вручную”?
- Вопросы распределённости/децентрализованности приложения
- Вопросы безопасности и прочие нефункциональные требования
- Вопросы локализации
- Вопросы размещения
- Вопросы обновления
- ...

Инструменты проектирования

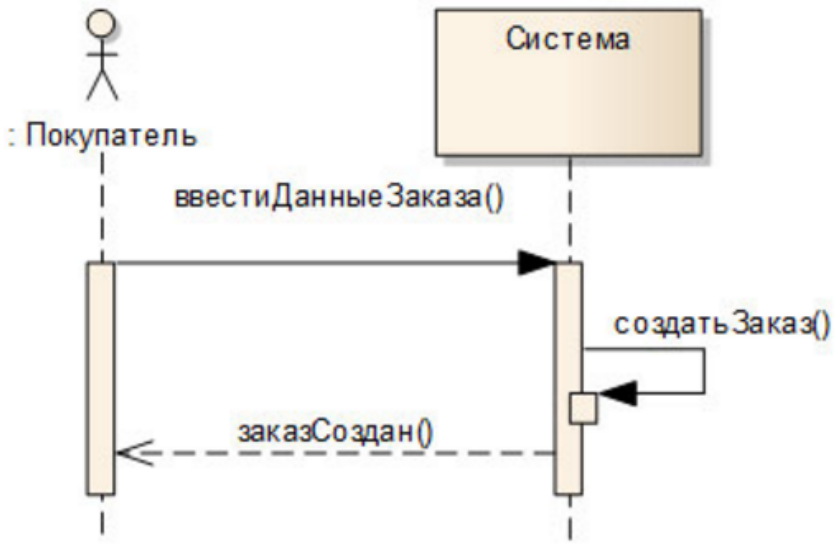
- Моделирование
  - Визуальное
  - Вербальное
  - Формальное
- Прототипирование
- Архитектурное описание
  - Неформальное
  - Формальное

Моделирование

- **Модель** — упрощённое подобие объектов или явлений
  - Позволяет изучать некоторые их свойства
- Свойства моделей
  - Сжатие информации
  - Целенаправленность
  - Субъективность
  - Ограниченность
  - Способность к расширению

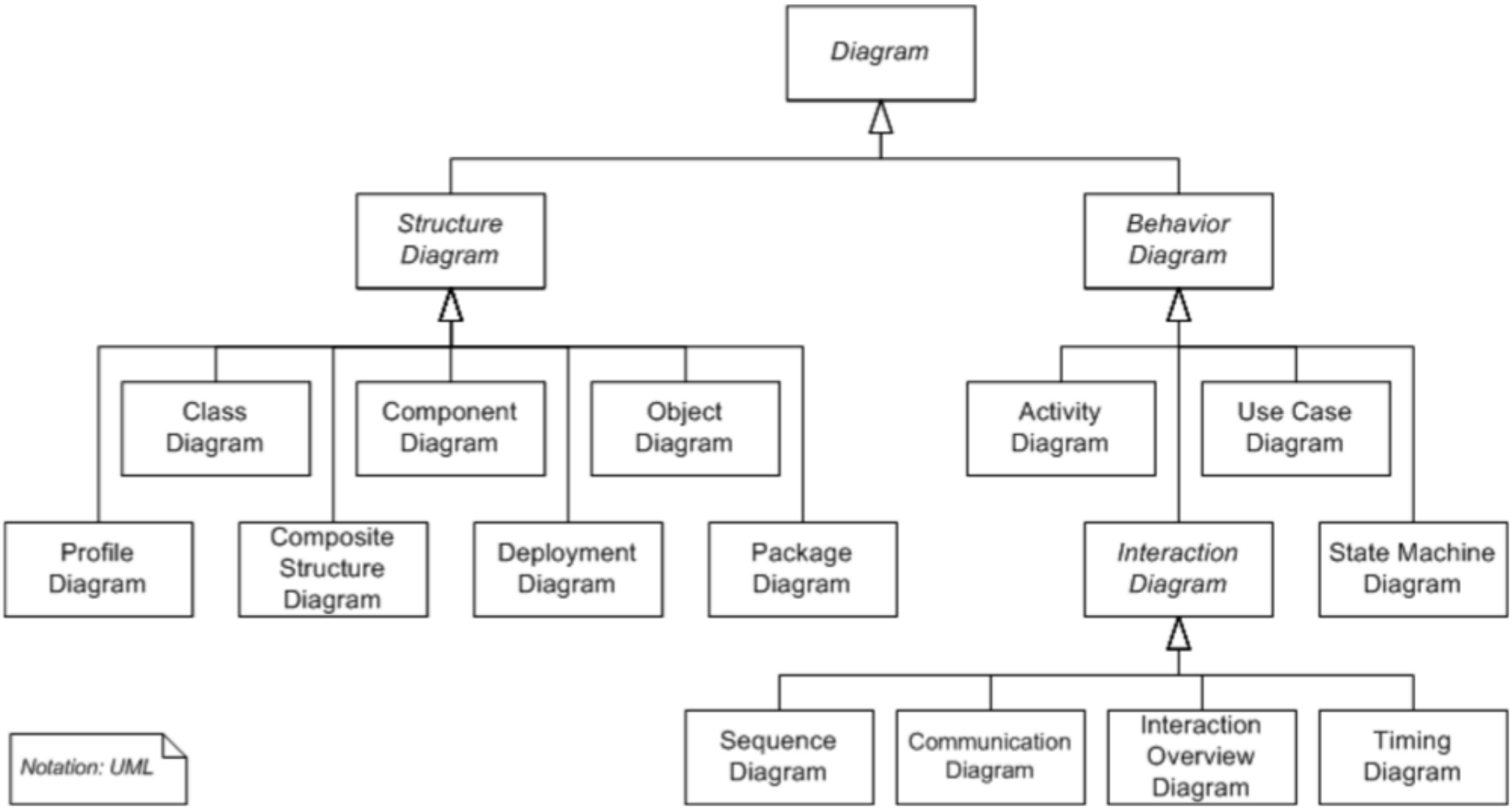
Визуальные модели

- Различный состав моделей и степень детальности
- Метафора визуализации
- Точка зрения моделирования
- Назначение
  - Одноразовые модели
  - Документация
  - Графические исходники

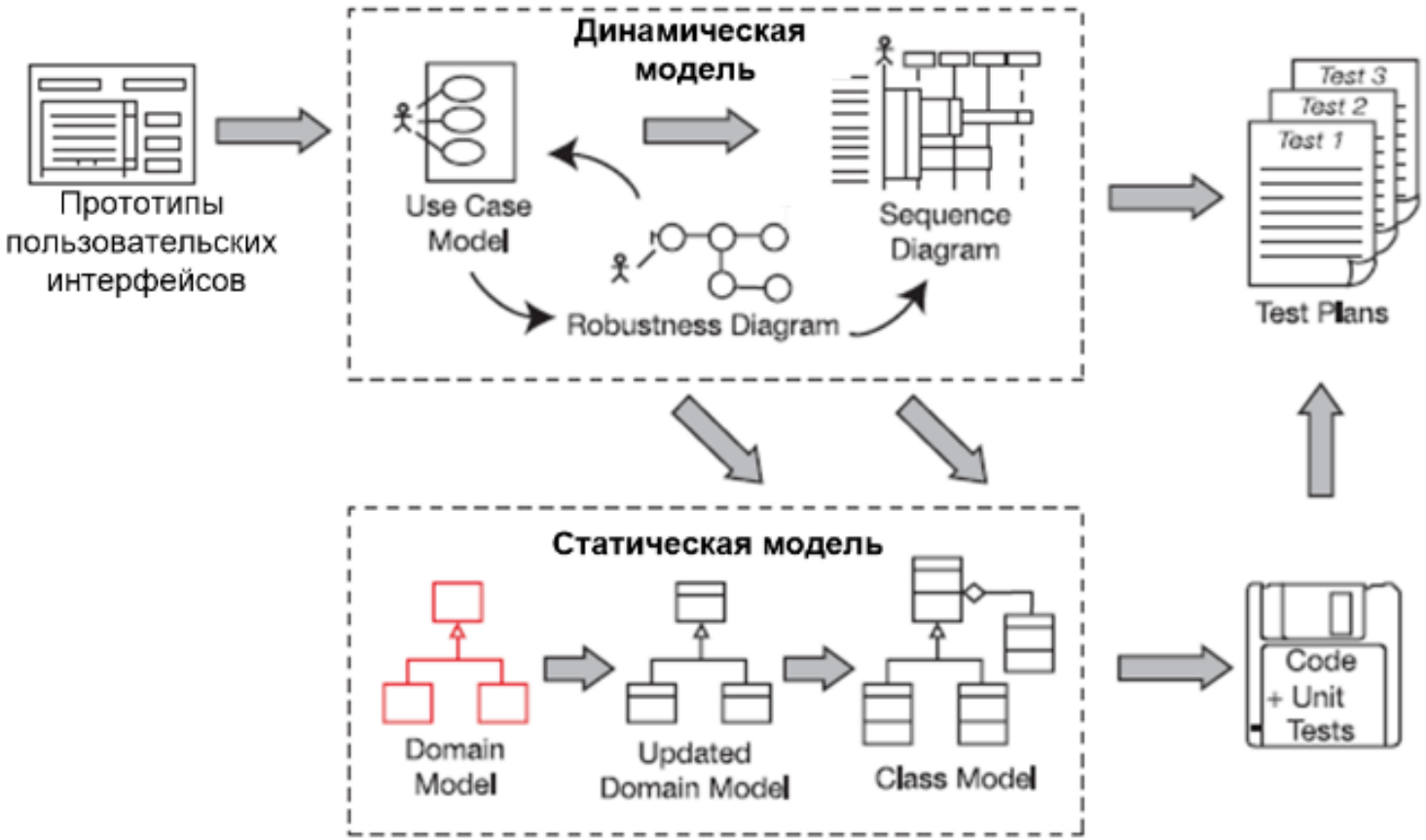


Язык UML

Unified Modeling Language



Представления системы



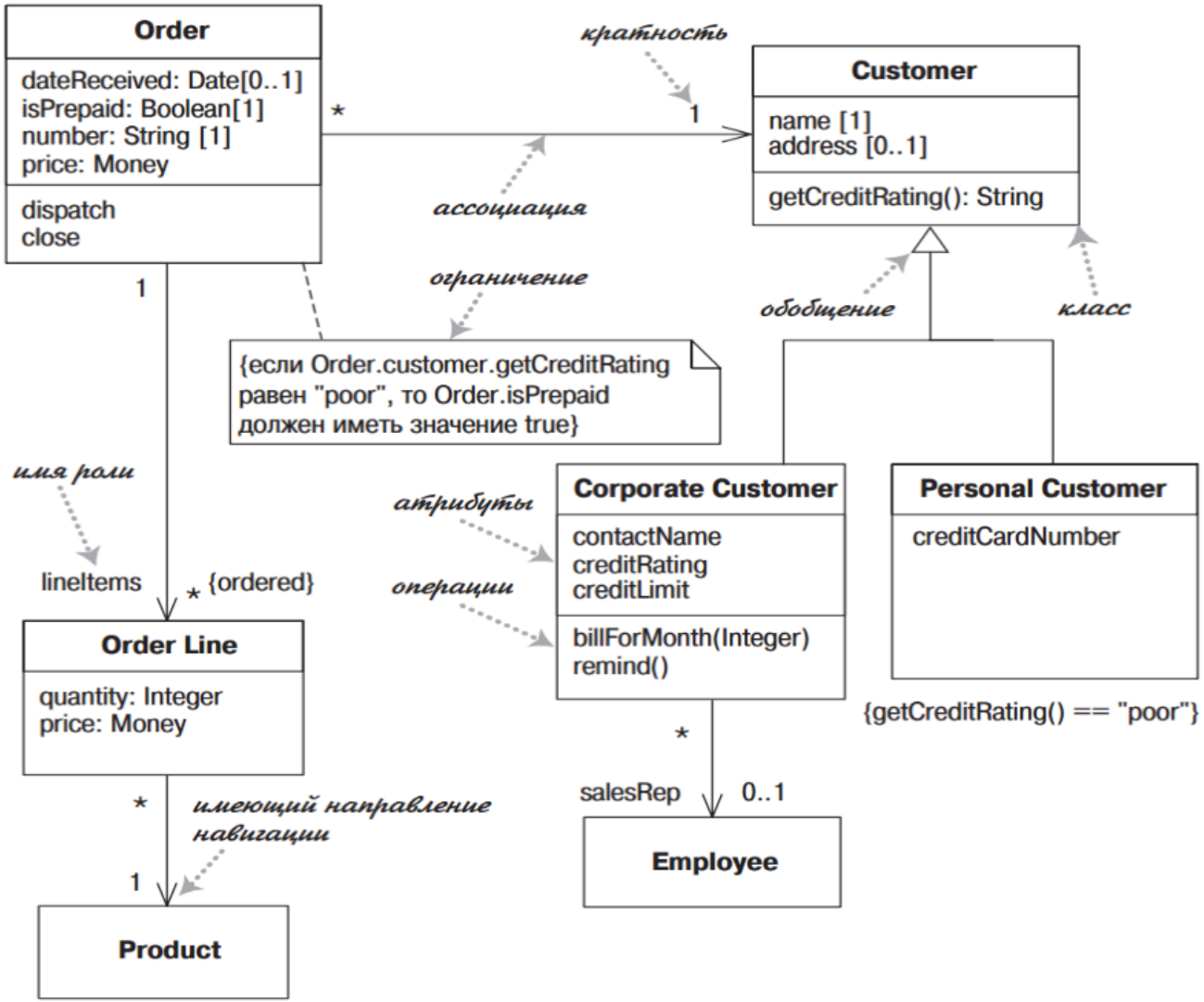
Моделирование структуры

- Отношения “часть-целое”
- Статические свойства
- Не рассматриваем
  - “Причина-следствие”
  - “Раньше-позже”
- Основные сущности — пакеты, классы и отношения между ними

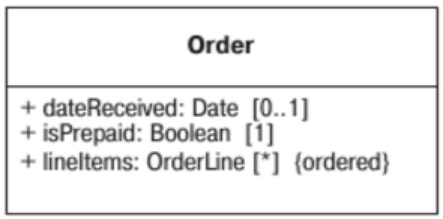
Что моделируем

- Структура связей между объектами во время выполнения программы
- Структура хранения данных
- Структура программного кода
- Структура компонентов в приложении
- Структура артефактов в проекте
- Структура используемых вычислительных ресурсов

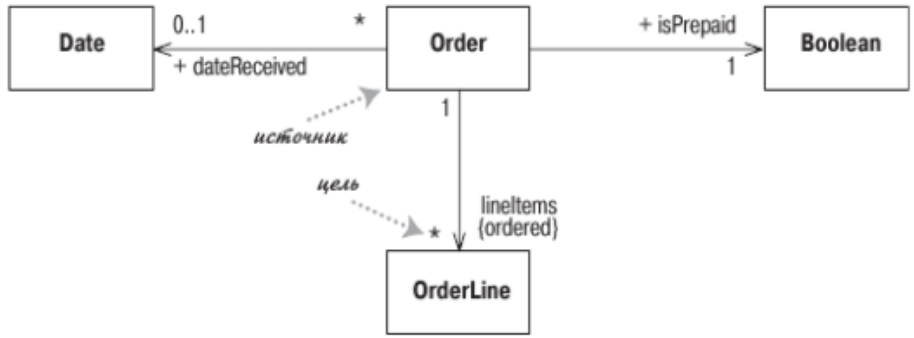
Диаграммы классов UML



Свойства



Атрибуты



Ассоциации

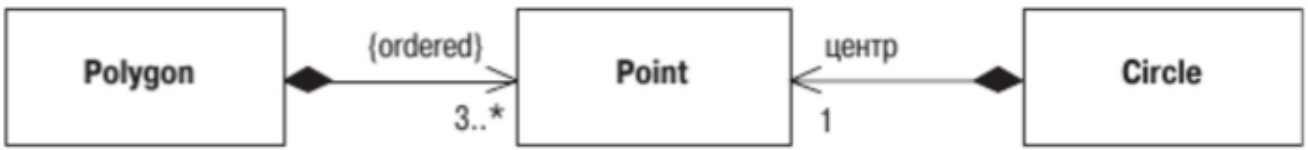
Синтаксис:

- 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..*

Агрегация и композиция

Агрегация – объект “знает” о другом (не управляет его временем жизни, имеет на него ссылку или указатель)

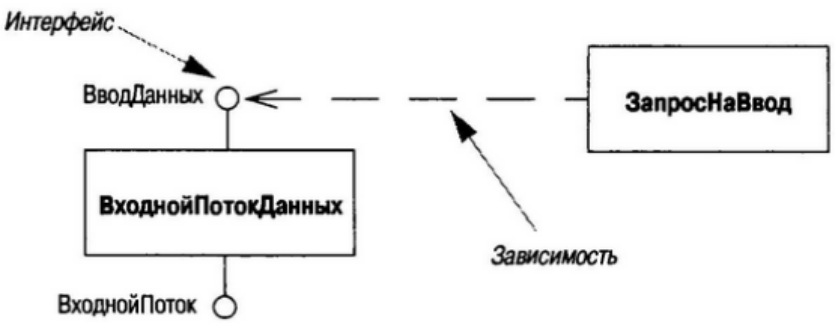
Композиция — объект владеет другим объектом (управляет его временем жизни, хранит его по значению или по указателю, делая delete)



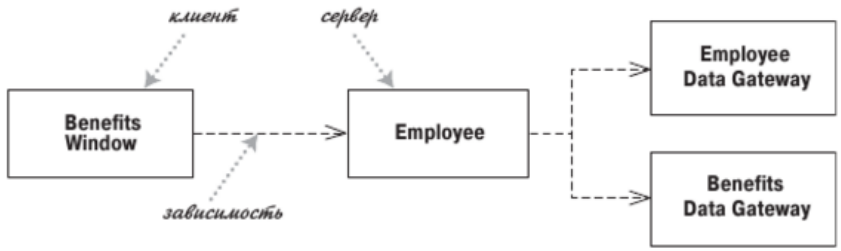
Уточнение обычной ассоциации, используется только если очень надо

Прочее

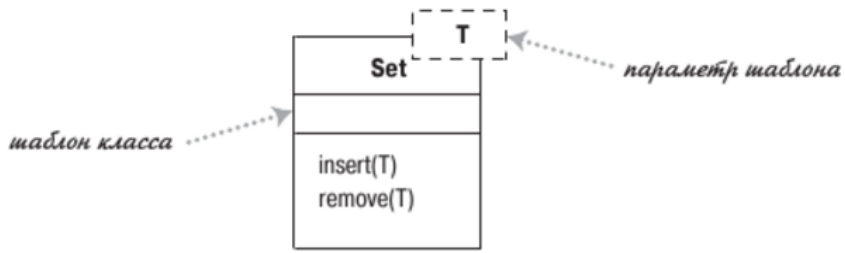
Интерфейсы



Зависимости

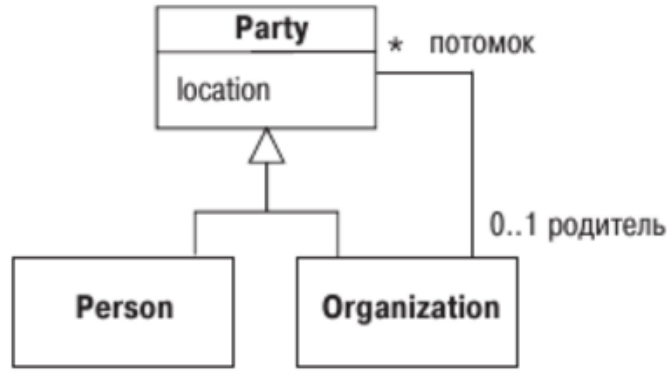


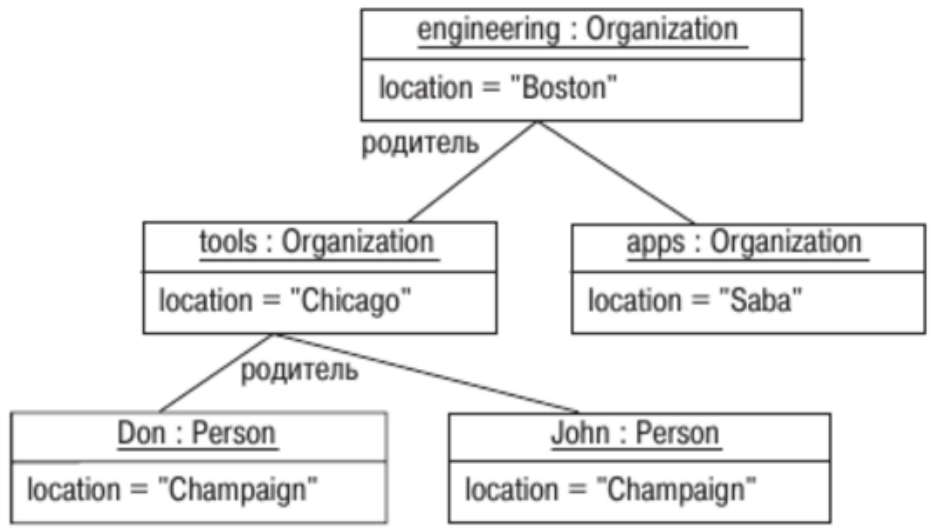
Шаблоны



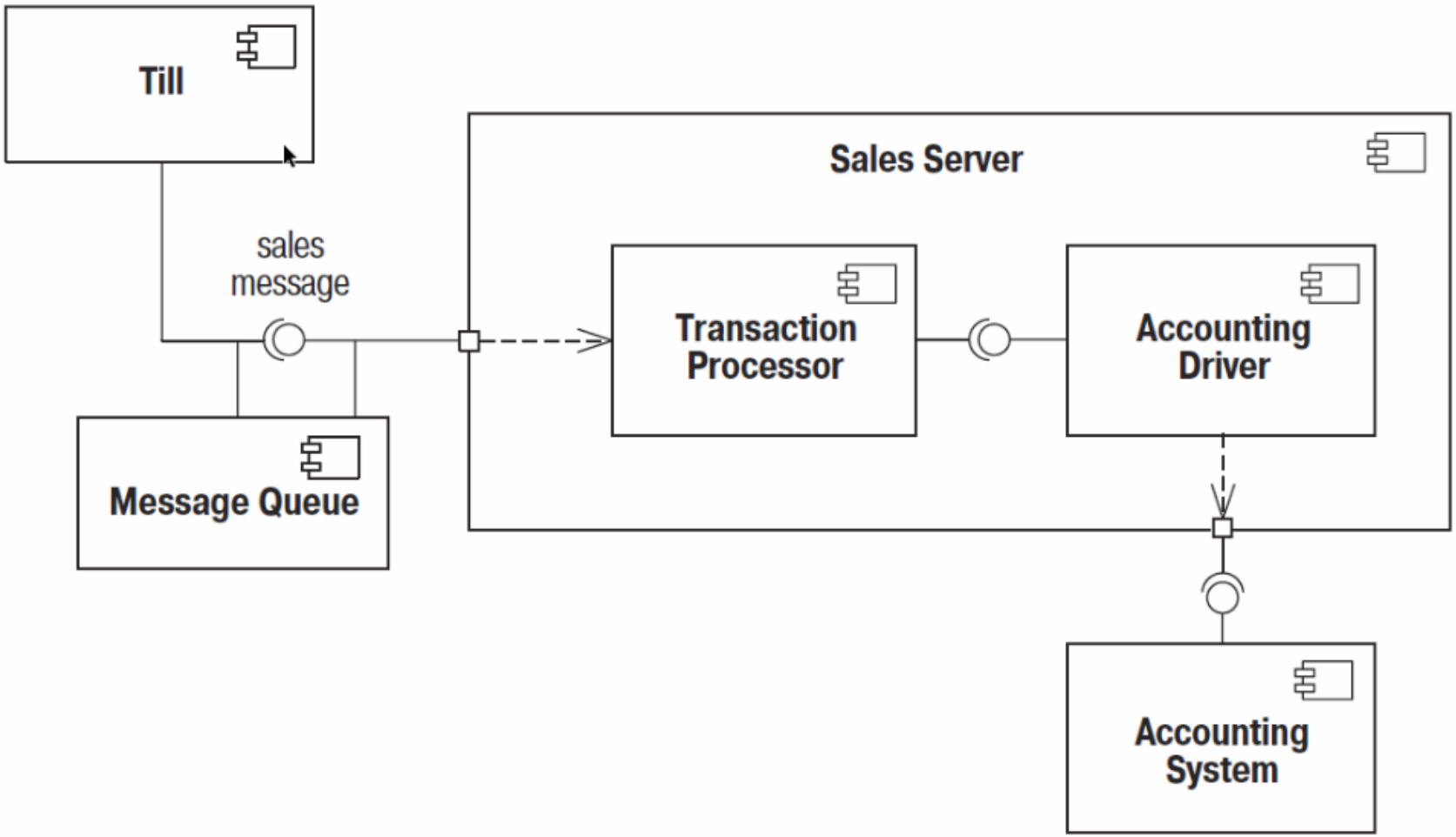
Диаграммы объектов

- snapshot структуры классов во время выполнения
- Используются обычно чтобы пояснить диаграмму классов
- Полезны на этапе анализа предметной области, ещё до диаграмм классов





Диаграммы компонентов



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

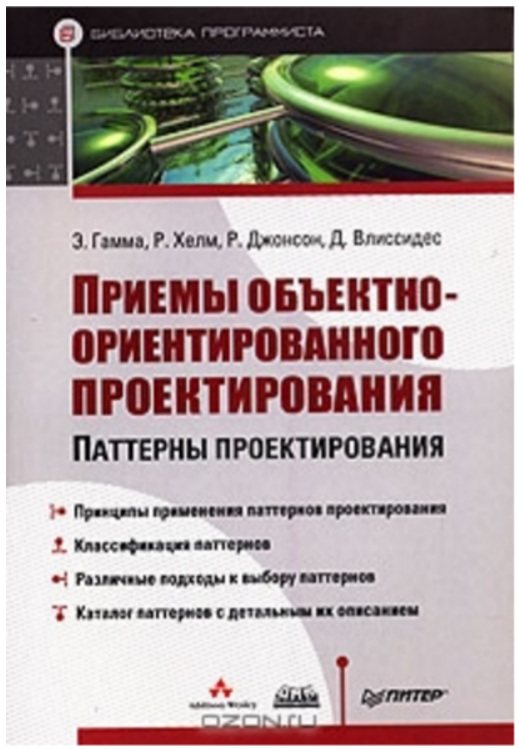
Примеры CASE-инструментов

- “Рисовалки”
  - Visio
  - Dia
  - SmartDraw
  - 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/

Паттерны проектирования

Книжка

Приемы объектно-ориентированного проектирования. Паттерны проектирования
Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес
Design Patterns: Elements of Reusable Object-Oriented Software



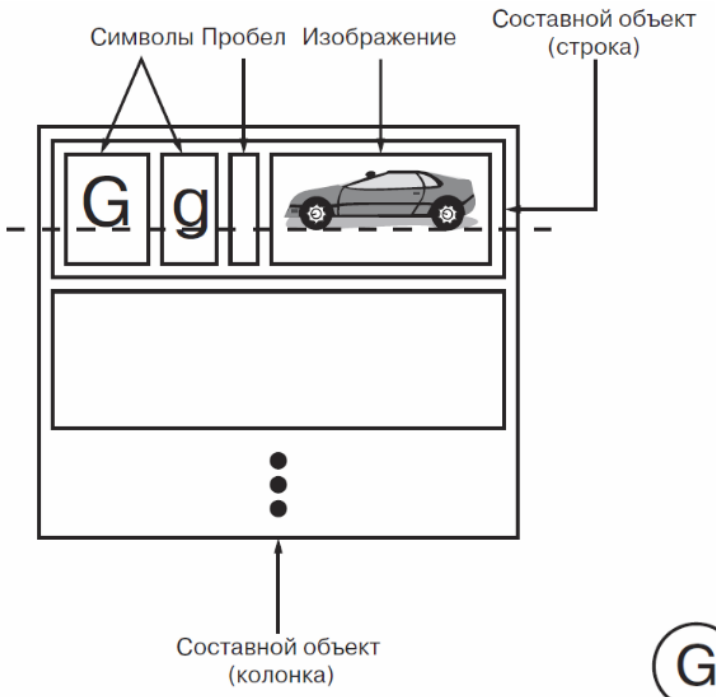
Паттерны проектирования

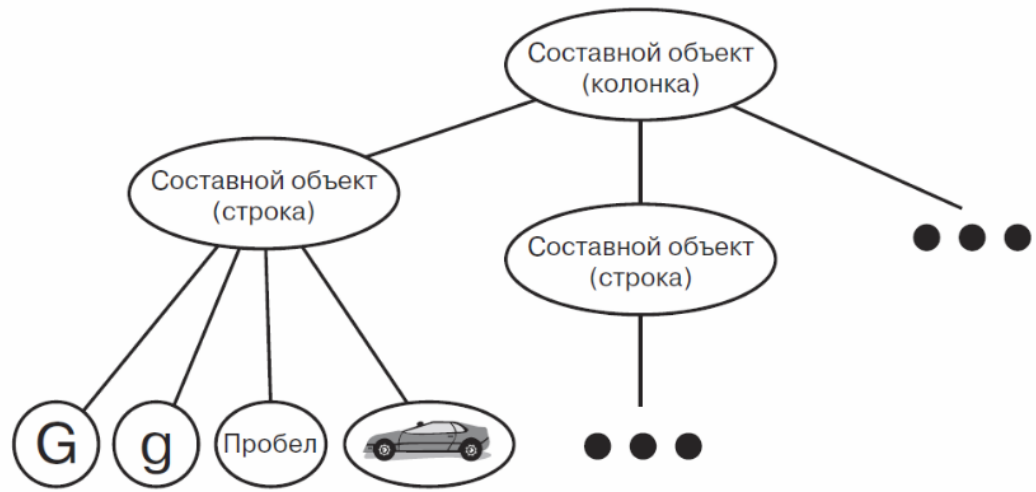
Паттерн проектирования — повторимая архитектурная конструкция, являющаяся решением некоторой типичной технической проблемы

- Подходит для решения целого класса проблем
- Переиспользуемость знаний
- Унификация терминологии
- Простота в изучении
- Опасность карго-культа!

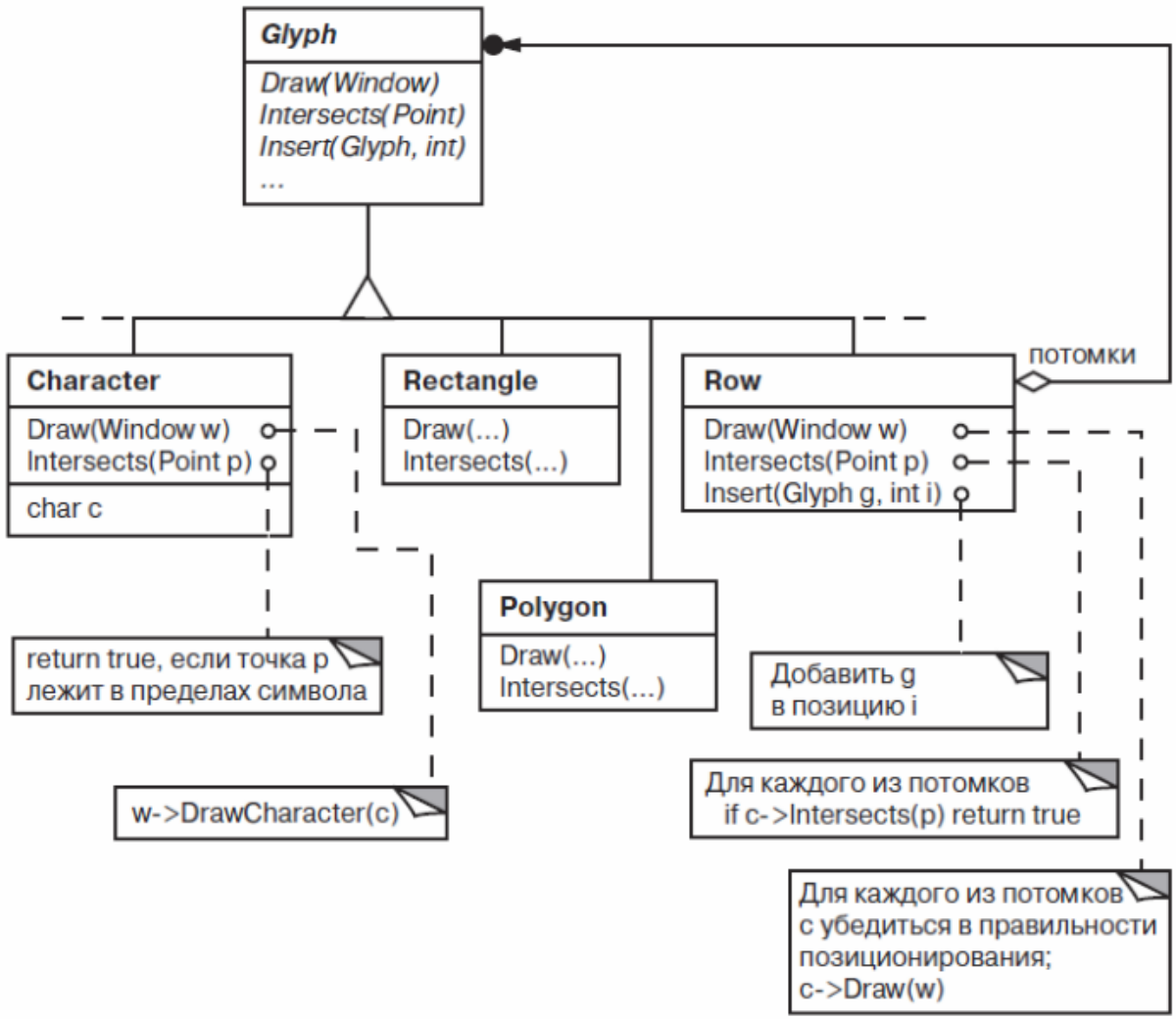
А ещё есть **антипаттерны** — часто встречающиеся неправильные решения типичной проблемы

Пример паттерна, компоновщик (1)

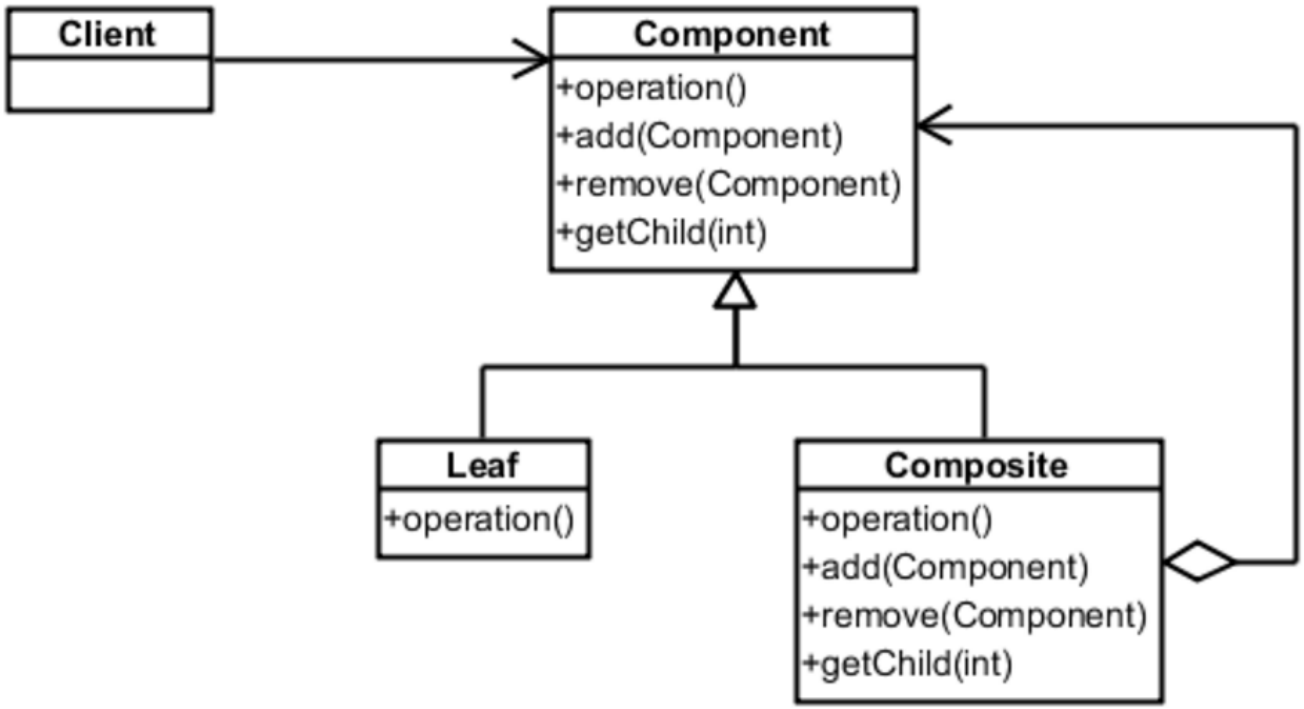




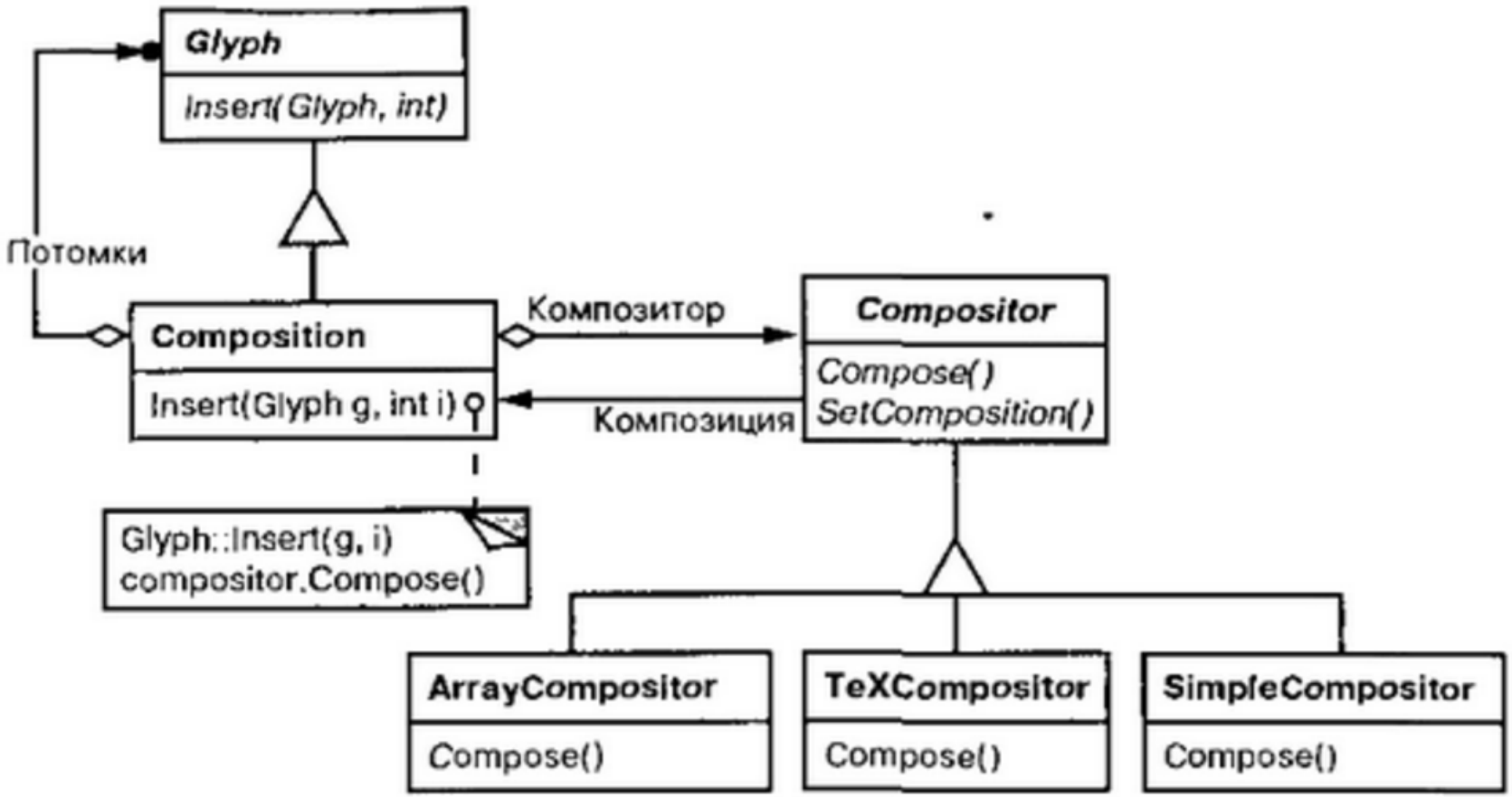
Пример паттерна, компоновщик (2)



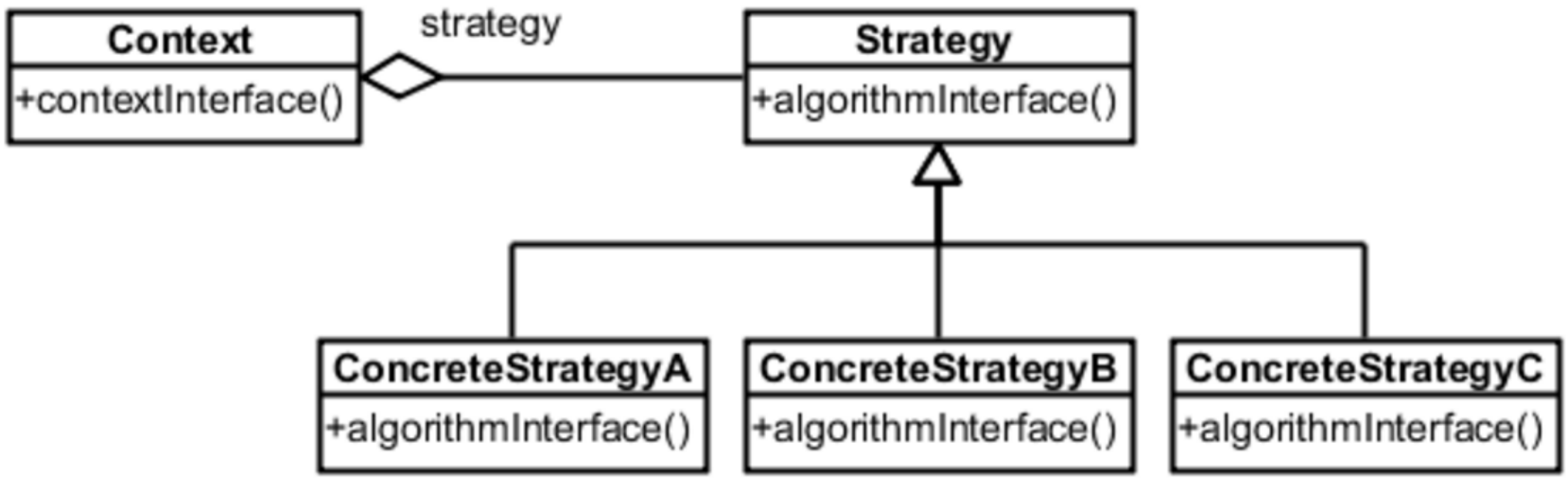
Пример паттерна, компоновщик (3)



Ещё пример, паттерн “Стратегия” (1)



Ещё пример, паттерн “Стратегия” (2)



Внезапно, тестирование

- Любая программа содержит ошибки
- Если программа не содержит ошибок, их содержит алгоритм, который реализует эта программа
- Если ни программа, ни алгоритм ошибок не содержат, такая программа даром никому не нужна

Тестирование не позволяет доказать отсутствие ошибок, оно позволяет лишь найти ошибки, которые в программе присутствуют

Виды тестов

- Модульные
- Интеграционные
- Системные

- Регрессионные
- Приёмочные
- Дымовые (smoke-test)

- UI-тесты
- Нагрузочные тесты
- ...

Модульные тесты

- Тест на каждый отдельный метод, функцию, иногда класс
- Пишутся программистами
- Запускаются часто (как минимум, после каждого коммита)
- Должны работать быстро
- Должны всегда проходить
- Принято не продолжать разработку, если юнит-тест не проходит
- Помогают быстро искать ошибки (вы ещё помните, что исправляли), рефакторить код (“ремни безопасности”), продумывать архитектуру (мешанину невозможно оттестировать), документировать код (каждый тест — это рабочий пример вызова)

Почему модульные тесты полезны

- Помогают искать ошибки
  - Особо эффективны, если налажен процесс Continuous Integration
- Облегчают изменение программы
  - Помогают при рефакторинге
- Тесты — документация к коду
- Помогают улучшить архитектуру
- НЕ доказывают отсутствие ошибок в программе

Best practices

- Независимость тестов
  - Желательно, чтобы поломка одного куска функциональности ломала один тест
- Тесты должны работать быстро
  - И запускаться после каждой сборки
    - Continuous Integration!
- Тестов должно быть много
  - Следить за Code coverage
- Каждый тест должен проверять конкретный тестовый сценарий
  - Никаких try-catch внутри теста
    - @Test(expected = NullPointerException.class)
    - Любая нормальная библиотека юнит-тестирования умеет ожидать исключения
- Test-driven development

Hamcrest

```
assertThat(someString, is(not(equalTo(someOtherString))));
assertThat(list, everyItem(greaterThan(1)));
assertThat(cat.getKittens(), hasItem(someKitten));
assertThat("test",
    anyOf(is("testing"), containsString("est")));
assertThat(x,
    allOf(greaterThan(0), lessThanOrEqualTo(10)));
```

Mock-объекты

- Объекты-заглушки, симулирующие поведение реальных объектов и контролирующие обращения к своим методам
  - Как правило, такие объекты создаются с помощью библиотек
- Используются, когда реальные объекты использовать
  - Слишком долго
  - Слишком опасно
  - Слишком трудно
  - Для добавления детерминизма в тестовый сценарий
  - Пока реального объекта ещё нет
  - Для изоляции тестируемого объекта
- Для mock-объекта требуется, чтобы был интерфейс, который он мог бы реализовать, и какой-то механизм внедрения объекта

Пример: Mockito

```
@Test
public void test() throws Exception {
    // Arrange, prepare behaviour
    Helper aMock = mock(Helper.class);
    when(aMock.isCalled()).thenReturn(true);
    // Act
    testee.doSomething(aMock);
    // Assert - verify interactions (optional)
    verify(aMock).isCalled();
}
```

Соотношение тестов