



# Информатика

## Разработка ПО.

### Проектирование, принципы

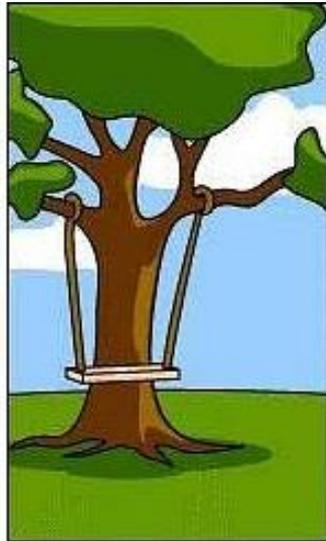
© Марченко Антон Александрович  
Абрамский Михаил Михайлович  
Аршинов Максим Вячеславович

2018 г.

# Типичный программный проект



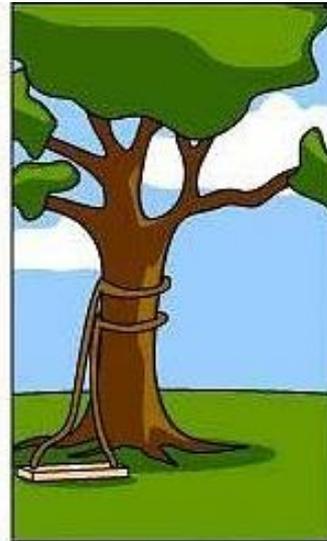
Как объяснил клиент  
чего он хочет



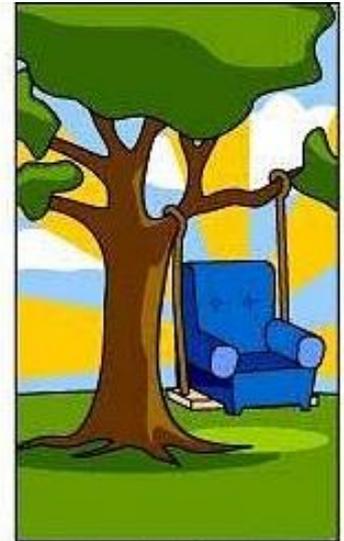
Как понял клиента  
начальник проекта



Как описал проект  
аналитик

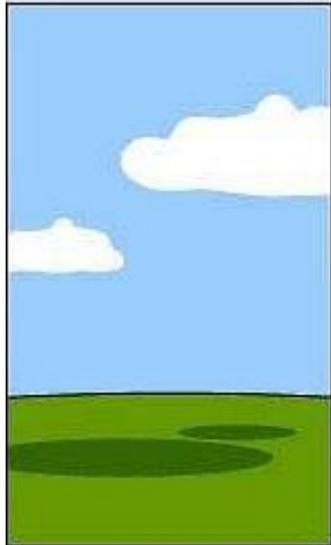


Как написал  
программист

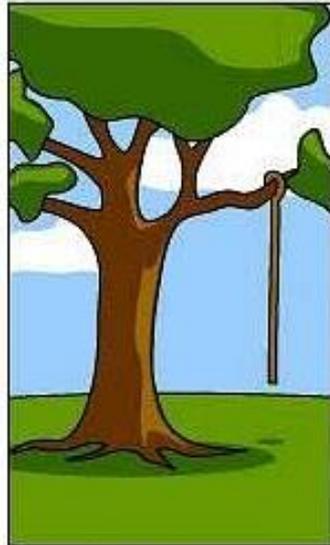


Как представил проект  
бизнес-консультант

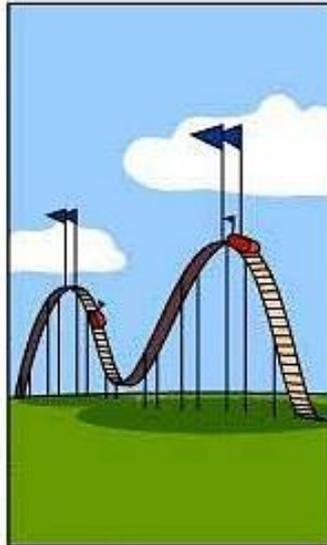
# Типичный программный проект



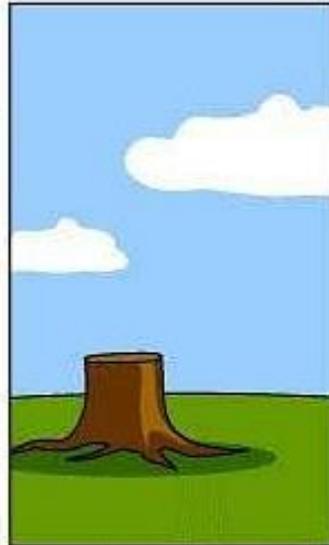
Как  
задокументировали  
проект



Какие фичи  
удалось внедрить



Как заплатил  
клиент



Как работала  
техническая  
поддержка



Что было нужно  
клиенту

# Разработка ПО

- ПО – больше чем программы
  - сложная динамическая система
  - согласованная с многими интерфейсами
  - развивающаяся во времени
  - изменяющаяся/эволюционирующая
- Разработка ПО ≠ программирование

# Разработка ПО

- Деятельность по созданию нового ПО
  - прототипирование
  - программирование
  - документирование
  - тестирование
  - поддержка
  - ...
- Часть программной инженерии

# Программная инженерия

Термин Software Engineering (SE) предложен Ф.Л. Бауэром в 1968г.

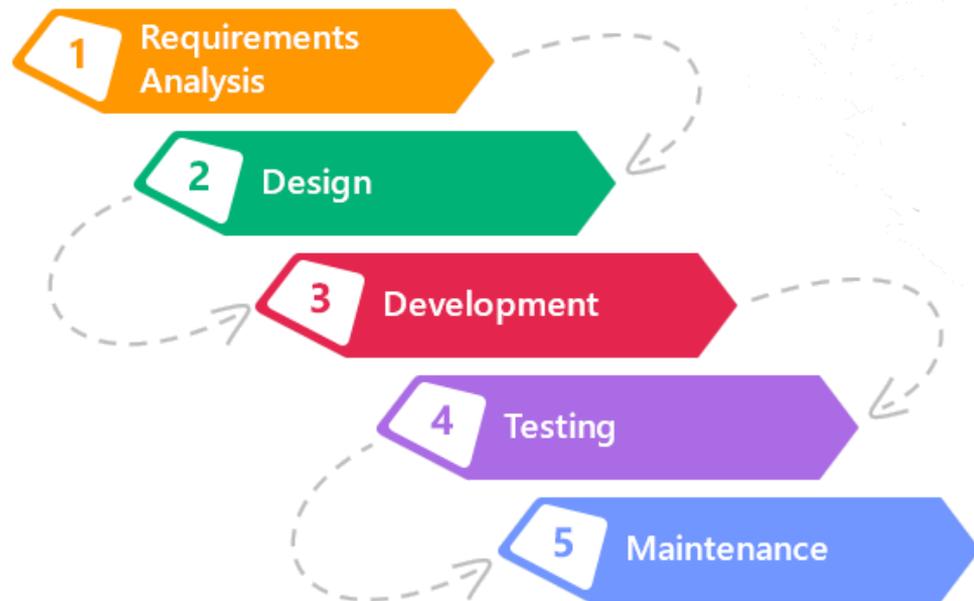
- **Определение ISO/IEC 2382-1**
  - Систематическое применение научных и технических знаний, методов и опыта для разработки, реализации, тестирования и документирования программного обеспечения.
- **Определение Иана Соммервиля** [кн. Software engineering]
  - инженерная дисциплина, охватывающая все аспекты создания ПО от начальной стадии разработки системных требований через создание ПО до его использования

# SE и информатика

- Информатика
  - подходы к программированию
    - теория и методы построения вычислительных и программных систем
- Программная инженерия
  - решение проблем производства ПО
  - основывается на фундаменте информатики

# Области SE

- Требования
- Проектирование
- Разработка
- Тестирование
- Поддержка и эксплуатация
- еще конфигурация, управление, процесс, методы и инструменты, качество



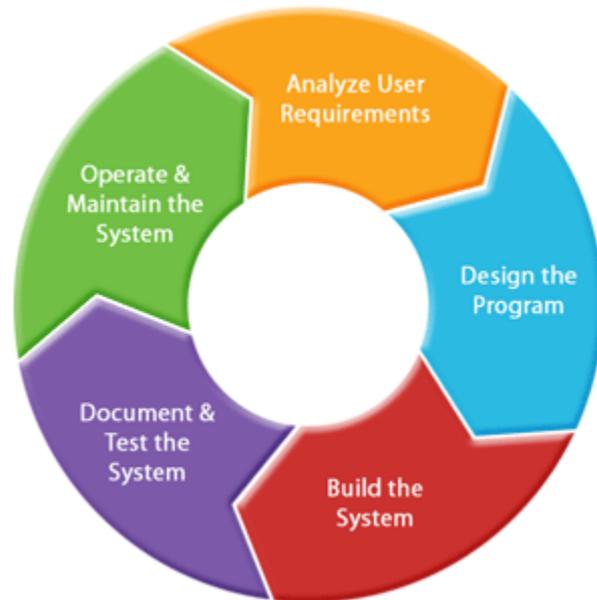
# Жизненный цикл

## Software development life cycle

описывает фазы развития программного проекта

Существует несколько моделей SDLC (методологий)

- каскадная,
- спираль,
- гибкая разработка,
- быстрое прототипирование,
- инкрементальная...



# Проблемы при разработке ПО

- Отсутствие четкой спецификации – подробного описания системы, полностью определяющего цели и функциональные возможности
- Недостаток прозрачности при недостаточном планировании структуры (архитектуры)
- Недостаток контроля над процессом
- Недостаток мониторинга
- Неконтролируемые изменения

# Спецификации

Все называют по-своему

- постановка задачи
- требования пользователя
- техническое задание
- функциональная спецификация
- архитектура системы

Говоря на своем языке, специалисты зачастую не понимают друг друга

# Решение

- Проблему можно решить при наличии единого, унифицированного средства создания спецификаций, достаточно простого и понятного для всех заинтересованных лиц

# UML

- Unified Modeling Language
- единый универсальный стандарт для визуального описания спецификаций
- язык объектно-ориентированного моделирования (визуальный)
- позволяет строить модель программных систем

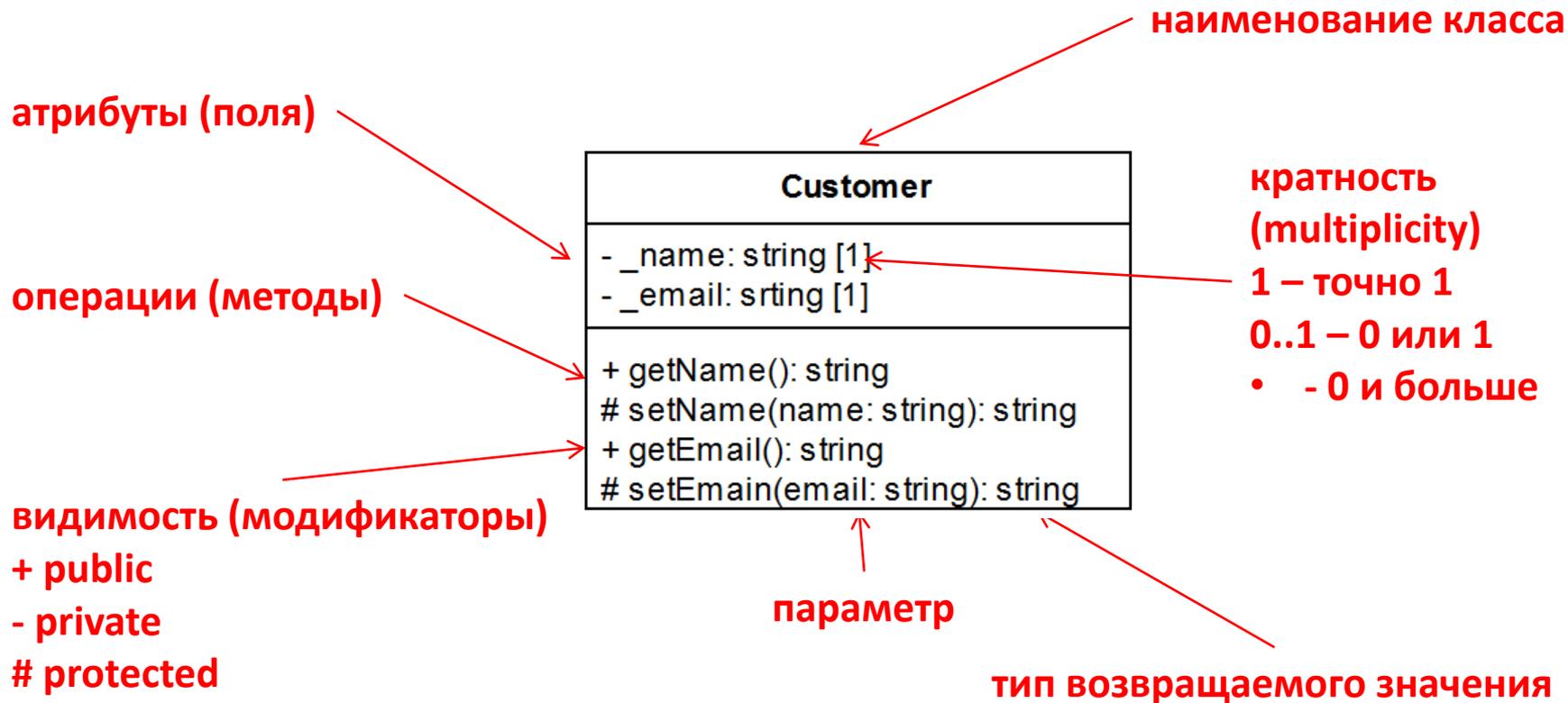
# Появление UML

- ООП требовало удобного инструмента для моделирования
- 1991. Описания «Трёх амиго» легли в основу языка Rumbaugh, Jacobson, Booch: The Unified Modeling Language Reference Manual
  - Грейди Буч (Grady Booch)
  - Джим Румбах (Jim Rumbaugh)
  - Айвар Якобсон (Ivar Jacobson)
- В 1995 UML был впервые продемонстрирован
- В 1997 стандартизировался

# UML: диаграммы

- **Логическая структура**
  - диаграмма классов
  - структурная, взаимодействия, компонентов
  - вариантов использования (use-case)
- **Поведение**
  - активности
  - состояний
  - последовательности, коммуникации
- **Физическая структура**
  - внедрения

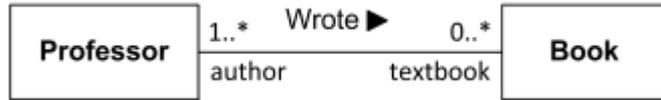
# Класс



# Ассоциация

- Структурная связь между классами (обычно ссылка)
- Бывает бинарной и n-арной
- Может иметь имя и навигацию
- Для классов могут быть указаны роли и кратность

# Ассоциация



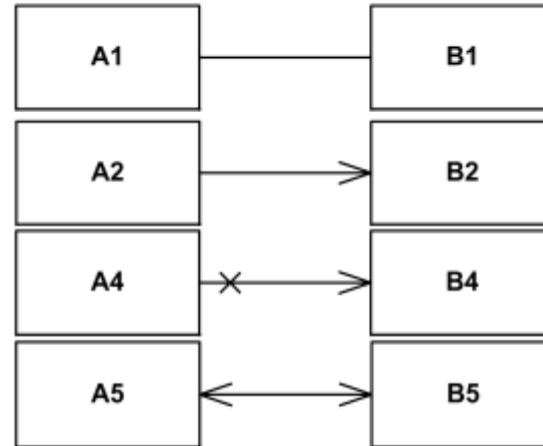
- Профессор в *роли* автора ассоциируется с учебником типа Книга
  - треугольник указывает как читать ассоциацию
- Класс может участвовать в разных ассоциациях в разных ролях
- Класс может иметь ассоциацию с самим собой

# Навигация в ассоциации

Навигация указывает достижимость класса из другого

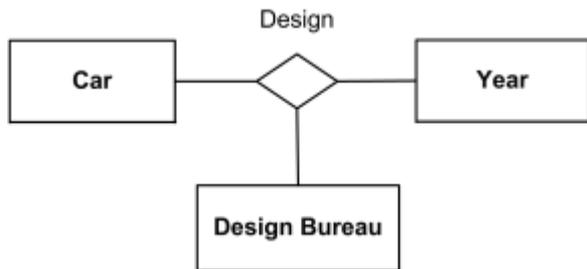
Может быть неопределена, разрешена и запрещена

- Неопределенная
- B2 достижим из A2
- A4 не достижим из B4
- A5 и B5 достижимы



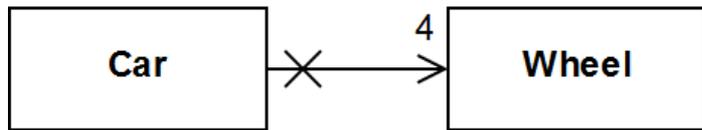
# n-арная ассоциация

- Указывает на структурную связь между 3 и более классами
- Обозначается ромбом



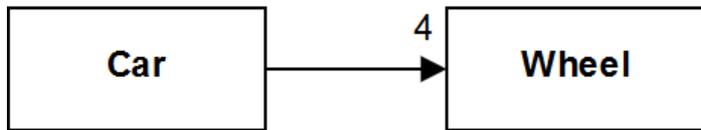
- пример тернарной ассоциации

# Пример ассоциации



- Явно указано что навигация запрещена
- Колеса достижимы из машины, но не наоборот
- Машина имеет ссылки на 4 колеса

# Пример ассоциации



- Что, если у машины кратность \*?
- Тогда одно колесо может быть прикреплено к нескольким машинам
- Если у машины кратность 1, то одно колесо может быть только на одной машине

# Выводы по ассоциации

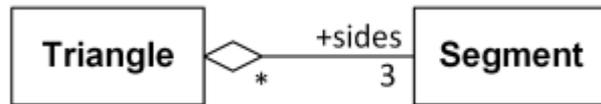
- Ассоциации – это то, что «склеивает» систему
- Ассоциация – связь
- Ассоциация описывает отношение между объектами во время исполнения

# Агрегация

- Бинарная **ассоциация**
- Связывает «целое» и «часть» (части)
- Только один конец ассоциации может быть агрегацией (обозначается ромбом)
- *«Части» могут одновременно содержаться в нескольких «целых»*

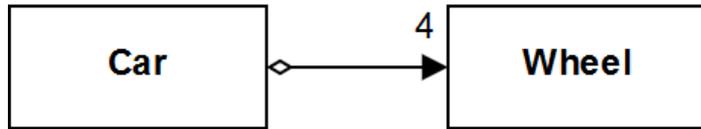
# Агрегация

- Бинарная ассоциация



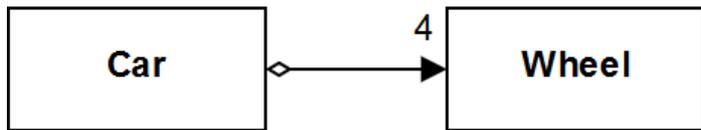
- Связывает «целое» и «часть» (части)
- Только один конец ассоциации может быть агрегацией (обозначается ромбом)
- «Части» могут одновременно содержаться в нескольких «целых»
- Не один сложный объект не может быть частью самого себя

# Пример



- Что означает агрегация в случае с машиной и колесами?
- Чем отличается от ассоциации?

# Пример



- Агрегация может привести к ошибкам проектирования (несогласованности)
- Используйте аккуратно

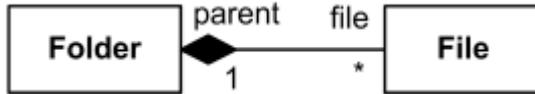
# Композиция

- Бинарная ***ассоциация***
- ***Сильная форма агрегации***
- Обозначает отношения между «целым» и «частью» («частями»)
- ***«Часть» может содержаться только в одном «целом»***
- Срок жизни «целого» и «части» совпадает

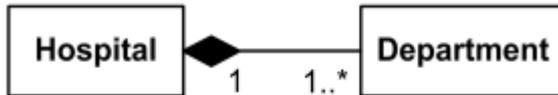
# Композиция

- **Обозначается закрашенным ромбом**

- Папка может содержать много файлов. При удалении папки удаляются и файлы



- У больницы есть несколько отделений. Каждое отделение принадлежит только одной больнице. Отделения закрываются вместе с больницей



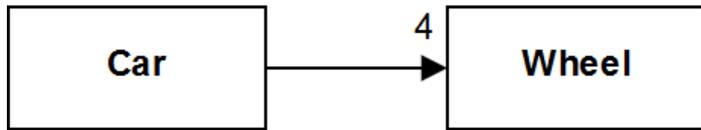
- У отдела есть персонал. Каждый сотрудник может быть членом отдела или быть «сам по себе»



# Пример

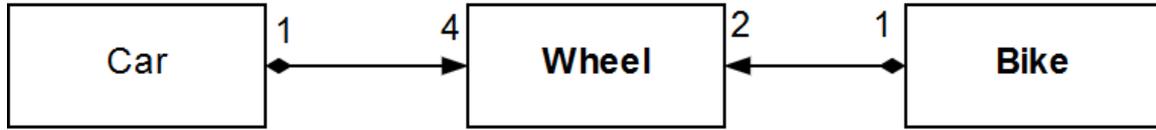


- Чем отличается от ассоциации?

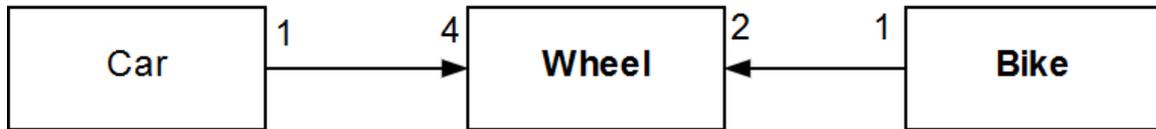


- Кратность композиции 0..1
- У экземпляра может быть только один владелец

# Пример

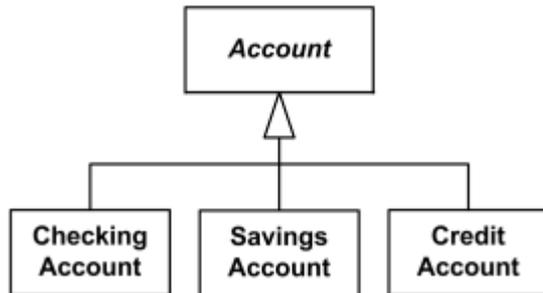


- Колеса могут быть частью машины и мотоцикла
- С композицией одно колесо не может быть частью машины и мотоцикла одновременно
- С ассоциацией такого ограничение на «одного владельца» – нет!



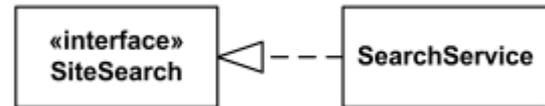
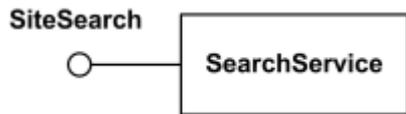
# Обобщение

- Реализуется наследованием
- Указывает на отношение между базовым и производным классом (суперклассом и подклассом)



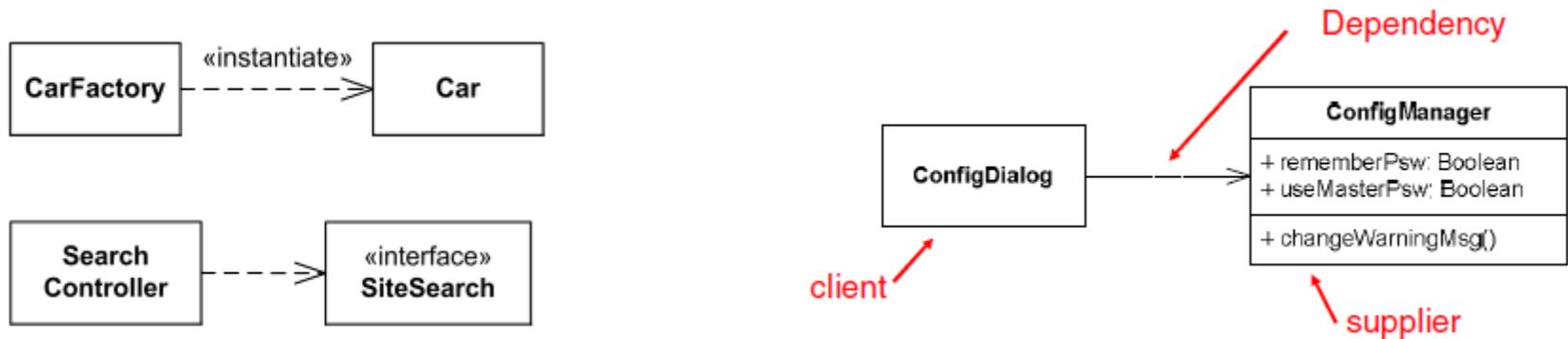
# Реализация

- Отношение между *спецификацией* и *реализацией*
- В чем отличие между интерфейсом и абстрактным классом?
- Может обозначаться по-разному



# Зависимость

- Направленное отношение между элементами, обозначается пунктирной стрелкой
- Изменения в одном влекут изменения в другом



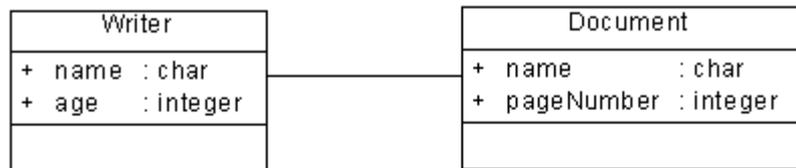
# Отношения

- Ассоциация: У А есть ссылка на экземпляры В (поля)
- Агрегация: часть может существовать отдельно от целого, может приводить к ошибкам проектирования
- Композиция: часть может принадлежать одному целому или быть сама по себе
- Обобщение: реализуется наследованием, экземпляр А может использоваться там, где ожидается В
- Реализация: А реализует интерфейс В
- Зависимость: А зависит от В, если изменения в В повлекут изменения в А

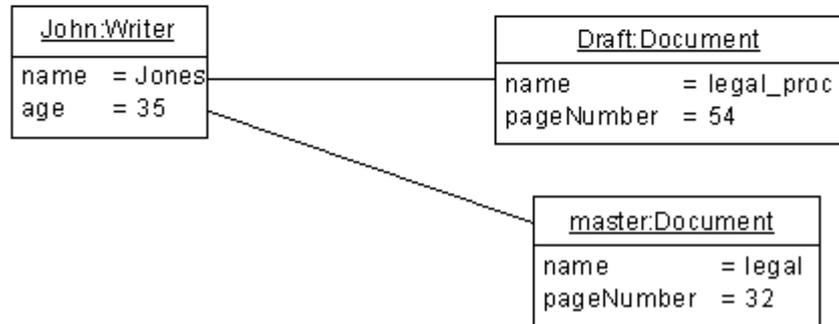
# Классы и объекты

- Диаграмма классов - отношения между классами
- Диаграмма объектов - отношения между конкретными экземплярами классов

*Class diagram*



*Object diagram*



# Принципы проектирования

Пять основных принципов объектно-ориентированного программирования и проектирования

**S.O.L.I.D.**

«...акроним, введённый Майклом Фэзерсом для пяти принципов, предложенных Робертом Мартинов в начале 2000-х» - Wiki

# S.O.L.I.D.

- Критерии проверки системы на изменяемость и расширяемость
- Руководства для проведения рефакторинга плохо спроектированного кода

# Расшифровка акронима

- S – single responsibility
- O – open/closed
- L – Liskov substitution
- I – interface segregation
- D – dependency inversion

# Признаки плохой модели

Читайте про «Код с душком»:  
Model smell, Code Smell

## Критерии плохого дизайна (Тепляков)

Несколько примеров:

- Слишком большие/маленькие классы
- Одно изменение затрагивает несколько компонент.
- После изменений перестаёт работать то, что раньше работало
- Трудно выделить компоненты для повторного использования
- Трудно добиться корректного поведения, а выполнить некорректные действия – легко
- Неоправданная сложность, зависимость от модулей, не дающих непосредственной выгоды
- Трудно разобраться в проекте, анализировать его

# Жесткость (Rigidity)

- Характеристика программы, затрудняющая внесение в неё даже самых простых изменений
- Дизайн жесткий, если единственное изменение вызывает каскад изменений в зависимых модулях
- Чем больше модулей приходится изменять, тем жёстче дизайн

# Хрупкость (Fragility)

- Склонность программы к повреждениям во многих местах при внесении единственного изменения
- Зачастую проблемы возникают в местах, не имеющих концептуальных связей с местом изменений
- Исправление одних проблем ведет к появлению других

*Команда разработчиков превращается в собаку, гонящуюся за собственным хвостом*

# Косность/неподвижность (Immobility)

- Программная система содержит части, которые могли бы оказаться полезными в других системах
- Но усилия и риски, сопряженные с попыткой отделить эти части от оригинальной системы, слишком велики  
слишком тяжело «выпутать» код для повторного использования

# Вязкость (Viscosity)

- Внесение изменений, относящихся к некоторому аспекту программной системы, без нарушения заложенных в проект принципов («хака») связано с большими затратами времени и усилий
- Если сохранить дизайн системы труднее, чем воспользоваться «хаком», то вязкость дизайна высока

# Ненужная сложность

- Система содержит элементы, неиспользуемые в текущий момент
- Часто случается, когда разработчики стараются предвидеть будущие изменения требований и вставлять в программу средства для их поддержки
- YAGNI

# Ненужная повторяемость (Copy/Paste)

- Происходит это так: Ивану нужно написать некий код. Полазив по разным частям программы, он находит подходящий фрагмент. Копирует его, вставляет в свой модуль и вносит необходимые изменения.
- Но Иван не знает, что код, который он извлек мышкой, поместил туда Петр, взявший его из модуля, написанного Светой.
- Свете первой пришлось написать такой код, но она знала, что этот процесс очень похож на другой. Она где-то отыскала другой код, скопировала его в свой модуль и модифицировала.
- Удачи с поддержкой 😊

# Непрозрачность/Плохая читабельность

- Трудность программного кода для понимания
- Код может быть чистым, ясным и выразительным или грязным, темным и запутанным
- Развивающийся (эволюционирующий) код со временем становится менее прозрачным, нужно поддерживать его в чистоте и следить за ясностью и выразительностью

# Дизайн – компромисс

- Борьба с жесткостью, хрупкостью и косностью может увеличить сложность и снизить прозрачность
- Жесткость может повысить надежность
- Стоимость снижения вязкости может быть выше, чем стоимость поддержки
- И другое...

# S.O.L.I.D.

- **SRP Принцип единственной обязанности**  
У класса должна быть только одна причина для изменения
- **OSP Принцип открытости/закрытости**  
Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для модификации
- **LSP Принцип подстановки Лисков**  
Должна быть возможность вместо базового типа подставить любой его подтип
- **DIP Принцип инверсии зависимости**  
Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций
- **ISP Принцип разделения интерфейсов**  
Клиенты не должны вынужденно зависеть от методов, которыми не пользуются. Интерфейсы принадлежат клиентам, а не иерархиям

# S: Single responsibility

Принцип единственной ответственности  
Том Демарко и Мейлир Пейдж-Джонс

***Не должно быть больше одной причины  
для изменения класса (Роберт Мартин)***

Если у класса больше одной ответственности,  
он будет меняться часто, дизайн будет хрупким,  
изменения - непредсказуемыми

# Пример: валидация данных

Если осуществлять проверку данных в самом классе, то может возникнуть потребность в изменении класса при использовании его экземпляров в разных клиентах

Решение: делегировать валидацию стороннему объекту, чтобы основной объект не зависел от реализации валидатора

# Пределные случаи (ошибки)

- Крайний пример несоблюдения принципа – God object
  - божественный объект – знает и умеет всё
- Крайний пример соблюдения принципа - необоснованная сложность, размазывание логики

# O: Open/closed

Программные сущности должны быть **открыты для расширения** (добавления поведения), **но закрыты для модификации** (расширение поведения не сопряжено с изменениями в исходном коде модуля)

**Модель должна быть устойчива к изменениям**

Закрытость – стабильность интерфейсов

Открытость – можно изменять поведение без изменения исходного кода класса (пересборки)

# Разные взгляды

- **Бертран Мейер**
  - Изменять модули нужно только для исправления багов.
  - Новая функциональность поставляется в новых модулях.
  - Это было давно и языки программирования и инструменты разработки были убогими.
- **Крейг Ларман**
  - Существующие клиенты не должны пострадать от модификации модуля
- **Роберт Мартин**
  - Должно быть просто изменить поведение программы путем добавления нового кода, а не изменения существующего.
  - Конфликтует с «ненужной сложностью».
  - Нет формальных способов гарантировать это, если вы не видите будущего.

# Open/closed principle

- **Protected Variation**

- Определите точки предполагаемых изменений и создайте надежные интерфейсы вокруг них.
- Паттерн «Фасад» или Public API может помочь.
- <https://plus.google.com/+SergeyTeplyakov/posts/c2LATUEHZvL>

- **Реализация:**

- Полиморфизм,
- Делегаты,
- Внедрение зависимостей

# Примеры ошибок:

- Использование конкретных объектов без выделения абстракций
- Привязка к абстракции, но выстраивание логики в зависимости от конкретных типов (is, typeof)

Решение: выделять абстракции, привязываться к ним, не требовать дополнительной информации о конкретных типах

# Крайние случаи (ошибки)

- Крайний случай несоблюдения – отсутствие абстракций
- Крайний случай соблюдения – чрезмерное число уровней абстракции – «фабрика фабрик»

# L: Liskov substitution

Принцип подстановки/замещения Барбары Лисков

**Вместо базового типа всегда должна быть возможность подставить его подтип**

Реализация «правильного» наследования

Требовать меньше, гарантировать больше

# Примеры ошибок

- Ошибочное наследование
  - Неочевидное изменение поведения, несогласующееся с иерархией
  - Изменение интерфейса при наследовании
- Анти-пример – непонятное наследование (или слишком много или совсем нет)

# I: Interface segregation

**Клиенты не должны зависеть от методов,  
которыми не пользуются**

Идея – предоставить удобный интерфейс с точки зрения различных клиентов

Не заставлять клиенты реализовывать методы, которые им не нужны

# Примеры ошибок

- Вынуждение наследников знать и делать слишком много
- «Жирный» интерфейс: все функциональные возможности в одном интерфейсе (клиенты разделены, а интерфейс – нет)
- Анти-пример: тысяча интерфейсов (интерфейс на каждый метод)

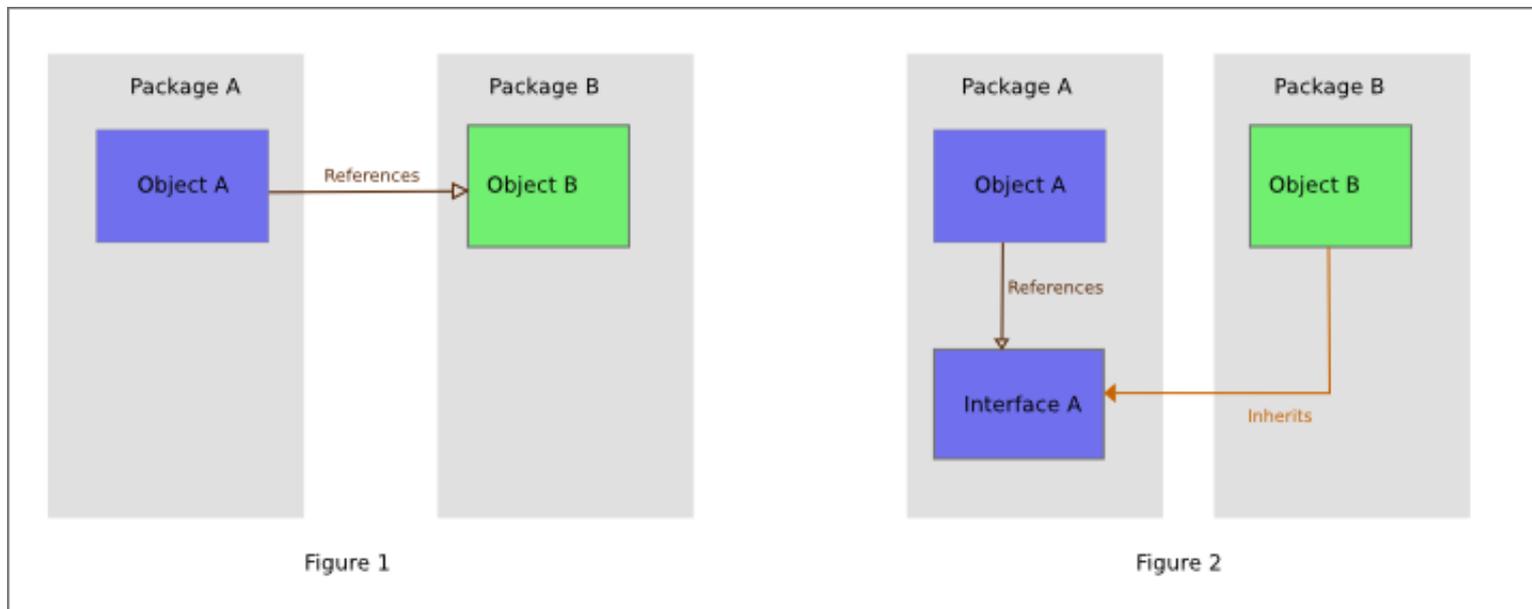
# D: Dependency inversion

Принцип инверсии зависимостей

**Модули верхнего уровня не должны зависеть  
от модулей нижнего,  
должны зависеть от абстракций**

Идея – сделать ключевые или изменчивые зависимости явными

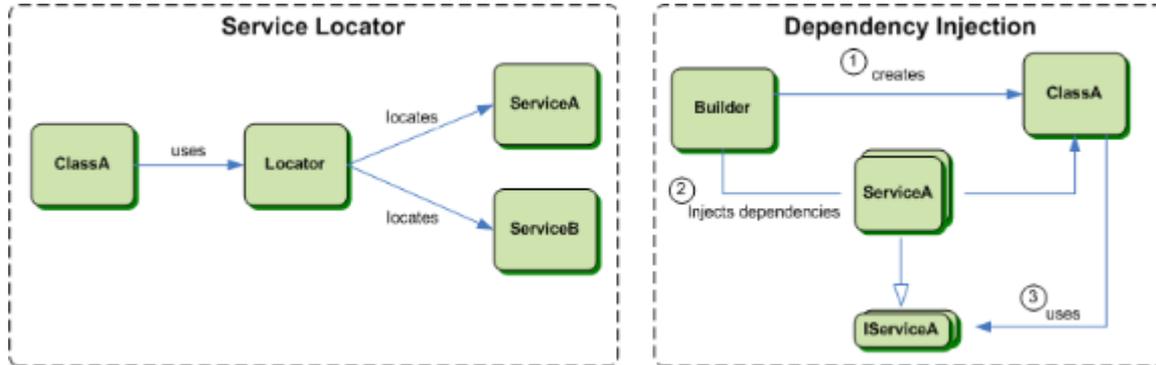
# В чём инверсия



Картинка для Java, но суть - одна

# DI, IoC

- Инверсия зависимостей (Dependency Inversion)
- Инверсия управления (Inversion Of Control)
  - Service Locator
  - Dependency Injection (через конструктор, метод, интерфейс)



# Примеры ошибок

Синглтоны, создание ключевых зависимостей в закрытых методах

*Анти-пример:* выделение интерфейса для каждого класса, передача всего и вся через конструкторы (невозможно понять где логика)

# Примеры ошибок

Синглтоны, создание ключевых зависимостей в закрытых методах

*Анти-пример:* выделение интерфейса для каждого класса, передача всего и вся через конструкторы (невозможно понять где логика)

# R.C.C.A.S – звучит не так круто, как SOLID

- REP Принцип эквивалентности повторного использования и выпуска  
Единица повторного использования равна единице выпуска.
- CCP Принцип общей закрытости  
Все классы внутри пакета должны быть закрыты относительно изменений одного и того же вида. Изменение, затрагивающее пакет, должно затрагивать все классы в этом пакете и только в нем.
- CRP Принцип совместного повторного использования  
Все классы внутри компонента используются совместно. Если вы можете повторно использовать один класс, то можете использовать и все остальные.
- ADP Принцип ацикличности зависимостей  
В графе зависимостей между пакетами не должно быть циклов.
- SDP Принцип устойчивых зависимостей  
Зависимости должны быть направлены в сторону устойчивости. SAP Принцип устойчивых абстракций  
Пакет должен быть столь же абстрактным, сколь и устойчивым.

# Принцип эквивалентности повторного использования и выпуска (Reuse/Release Equivalence Principle – REP)

- **Единица повторного использования равна единице выпуска.**
- Нужно еще, чтобы автор обеспечил возможность отказаться от использования новых версий. А то, не дай бог, он выпустит новую версию как раз тогда, когда у вас трещит по швам график, или внесет изменения, несовместимые с вашей системой.
- Но если вы решите отказаться от новой версии, то автор должен гарантировать поддержку старой версии в течение какого-то периода времени. Может быть, всего три месяца, может быть, год – это вопрос обсуждаемый.
- Если автор не соглашается поддерживать старые версии, то вы всерьез задумайтесь, стоит ли использовать такой код и целиком зависеть от капризов автора.

# Принцип совместного повторного использования (Common Reuse Principle – CRP)

- **Все классы внутри компонента используются совместно. Если вы можете повторно использовать один класс, то можете использовать и все остальные.**
- Синоним High Cohesion.

# Принцип общей закрытости (Common Closure Principle – CCP)

- **Все классы внутри компонента должны быть закрыты относительно изменений одного и того же вида. Изменение, затрагивающее компонент, должно затрагивать все классы в этом компоненте и только в нем.**
- Это не что иное как принцип единственной обязанности (SRP) в применении к компонентам. SRP говорит, что у класса не должно быть более одной причины для изменения, а CCP – что то же самое справедливо и для компонента.

# Принцип ацикличности зависимостей (Acyclic Dependencies Principle – ADP)

- **В графе зависимостей между компонентами не должно быть циклов.**
- Вы целый день упорно трудились, отладили какую-то часть программы, а придя на следующее утро, обнаружили, что она уже не работает? А почему не работает?
- Потому, что кто-то вчера задержался дольше, чем вы, и изменил нечто, от чего зависит ваша часть!

# Принцип устойчивых зависимостей (Stable-Dependencies Principle – SDP)

- **Зависимости должны быть направлены в сторону устойчивости.**
- Компонент, который предполагается изменять, не должен зависеть от компонентов, изменение которых затруднено! Иначе изменчивый компонент тоже будет трудно модифицировать.

# Принцип устойчивых абстракций (Stable-Abstractions Principle – SAP)

- **Компонент должен быть столь же абстрактным, сколь и устойчивым.**
- В сочетании принципы SAP и SDP дают аналог принципа DIP для компонентов.
- Действительно, SDP говорит, что зависимости должны быть направлены в сторону устойчивости, а SAP – что из устойчивости вытекает абстрактность. Следовательно, зависимости направлены в сторону абстрактности.

# Ссылки

<https://habrahabr.ru/post/208454/>

<http://sergeyteplyakov.blogspot.co.id/2013/04/blog-post.html>

<https://blogs.msmvps.com/jonskeet/2013/03/15/the-open-closed-principle-in-review/>

<https://8thlight.com/blog/uncle-bob/2013/03/08/AnOpenAndClosedCase.html>



Вопросы?

*e-mail:* [marchenko@it.kfu.ru](mailto:marchenko@it.kfu.ru)

© Марченко Антон Александрович  
Абрамский Михаил Михайлович  
Аршинов Максим Вячеславович

2018 г.



# Информатика

## Разработка ПО.

### Проектирование, принципы

© Марченко Антон Александрович  
Абрамский Михаил Михайлович  
Аршинов Максим Вячеславович

2018 г.