



Самарский государственный аэрокосмический университет
имени академика С.П. Королёва

Объектно-ориентированное программирование

Отношения между типами и особенности разработки

Занятие 11

© Составление,
А.В. Гаврилов, 2014
А.П. Порфирьев, 2015

Самара
2015

План лекции

- Отношения между классами
- Объектно-ориентированный дизайн
- Принципы дизайна



Отношения между классами

- Наследование
- Агрегация
- Зависимость
- Композиция
- Ассоциация
- Метакласс



Наследование

Особенности

- Использование одним классом структуры и/или поведения другого класса
- Дочерний класс специализирует родительский класс
- Образуется отношение «общее-частное» между классами

Реализация на практике

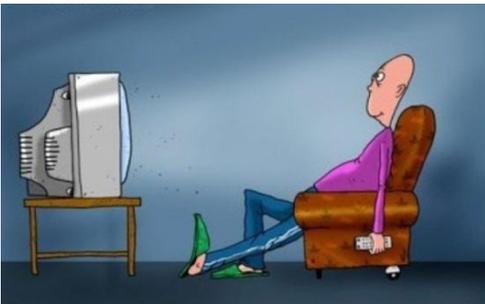
- Часть синтаксиса языка
- Возможен особый синтаксис для множественного наследования
- Особый синтаксис для абстрактных классов
- Возможен особый синтаксис для полностью абстрактных классов



Зависимость

Особенности

- Изменение в одном классе (независимом) может влиять на другой класс (зависимый)
- Зависимый класс как-то использует независимый
- Отношение направленное по своей природе



Реализация на практике

- У зависимого класса есть операция, сигнатура которой содержит параметр, имеющий тип независимого класса
- В ходе выполнения операции зависимого класса иным способом получается и используется объект независимого класса



Ассоциация

Особенности

- Семантическая связь классов
- Мощность
 - Один к одному
 - Один ко многим
 - Многие к одному
 - Многие ко многим



Новая стиральная машина с сушкой

Реализация на практике

- У класса есть поле типа другого класса
- Мощность
 - «К одному» – поле имеет просто тип класса
 - «Ко многим» – поле имеет тип массива или коллекции класса



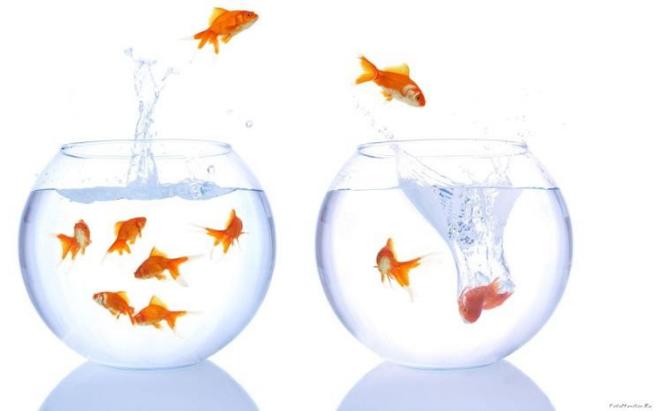
Агрегация

Особенности

- Можно считать частным случаем ассоциации
- Образуется отношение «целое-часть» между объектами
- Агрегат (контейнер) – большой внешний объект
- Строгая направленность

Реализация на практике

- Всё почти так же, как и в случае ассоциации
- Ссылки между объектами не могут образовывать циклы



Композиция

Особенности

- Можно считать частным случаем агрегации
- Время жизни объектов-частей определяется объектом-контейнером и не может превышать время жизни объекта-контейнера
- Объект-часть не существует самостоятельно

Реализация на практике

- Всё почти так же, как в случае агрегации
- Создание и уничтожение объектов-частей происходит только в ходе выполнения операций



Метакласс

Особенности

- Метакласс – это класс, объектами которого являются классы
- Корневой метакласс – единственный объект, являющийся своим собственным классом

Реализация на практике

- Очень сильно зависит от языка
- В некоторых языках существует только один метакласс – корневой



Ещё одна проблема разработки программ в ООП

- Этапы разработки (условно)
 - Определение модели данных
 - Определение алгоритма в виде последовательности операций
 - Реализация на языке программирования
- Проблема
 - Структурной единицей программы является класс
 - Из-за инкапсуляции модель данных связана с алгоритмом
 - Разделить данные и алгоритмы между классами можно далеко не единственным способом...



Object-oriented design

- Объектно-ориентированное проектирование – планирование системы как совокупности взаимодействующих объектов с целью решения программной задачи
- Для решения задачи необходимо разделить её на части и выбрать ответственных за них
- Инкапсуляция разделяет ответственности (responsibility) между классами
- Результат проектирования – распределение ответственностей и активностей по классам



Характеристики дизайна

■ Coupling

(связанность, зависимость)

- Характеристика взаимосвязи модулей
- Степень того, насколько модуль зависит от других модулей
- Мера ресурсов, требующихся при внесении изменений

■ Cohesion

(связность, сцепленность, сфокусированность, сосредоточенность)

- Степень того, насколько модуль сфокусирован на решение одной задачи
- Степень того, насколько элементы модуля гармонизированы, подходят друг другу



Виды связанности

- **Связанность содержимого (content coupling)-**
 - Один модуль изменяет или полагается на внутренние особенности другого модуля (например, используются локальные данные другого модуля)
 - Изменение работы второго модуля приведёт к переписыванию первого
- **Связанность через общее (common coupling)-**
 - Два модуля работают с общими данными (например, глобальной переменной)
 - Изменение разделяемого ресурса приведёт к изменению всех работающих с ним модулей



Виды связанности

- Связанность через внешнее (external coupling)
 - Два модуля используют навязанный извне формат данных, протокол связи и т.д.
 - Обычно возникает из-за внешних сущностей (инструментов, устройств и т.д.)
- Связанность по управлению (control coupling)+
 - Один модуль управляет поведением другого
 - Присутствует передача информации о том, что и как делать



Виды связанности

- Связанность по структурированным данным (data-structured coupling, stamp coupling)
 - Модули используют одну и ту же структуру, но каждый использует только её часть (части могут и не совпадать)
 - Изменение структуры может привести к изменению модуля, который изменённую часть даже не использует
- Связанность через данные (data coupling)+
 - Модули совместно используют данные, например, через параметры
 - Элементарные фрагменты маленькие и только они используются модулями совместно



Виды связанности

- Связанность по сообщениям (message coupling)
 - Модули общаются только через передачу параметров или сообщений
 - Состояние децентрализовано
- Отсутствие связанности (no coupling)
 - Модули вообще никак не взаимодействуют



Виды сфокусированности

- **Случайная (coincidental cohesion)**
 - Части модуля сгруппированы «от фонаря»
 - Единственное, что их объединяет – сам модуль
- **Логическая (logical cohesion)**
 - Части модуля логически относятся к одной проблеме
 - При этом части могут различаться по своей природе
- **Временная (temporal cohesion)**
 - Части модуля обычно используются в программе в одно время, рядом
- **Процедурная (procedural cohesion)**
 - Части модуля всегда используются в определённом порядке



Виды сфокусированности

- По взаимодействию (*communication cohesion*)
 - Части модуля работают над одними и теми же данными
- По последовательности действий (*sequential cohesion*)
 - Результат работы одной части модуля является исходными данными для другой
- Функциональная (*functional cohesion*)
 - Части модуля направлены на решение одной чёткой задачи, за которую отвечает модуль

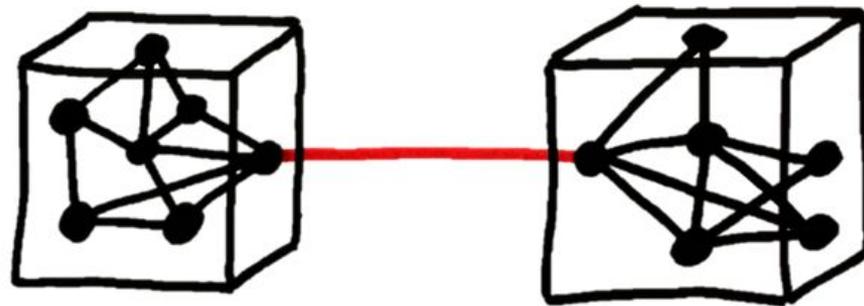


Плохой дизайн

- Высокая связанность
 - Эффект кругов по воде (или снежной лавины) при внесении изменений
 - Сборка модулей требует большего времени и/или затрат из-за связей между модулями
 - Конкретный модуль может быть тяжело тестировать и/или повторно использовать
- Слабая сфокусированность
 - Сложности с пониманием модулей
 - Сложности с поддержкой системы, т.к. логически согласованные изменения могут потребовать изменения многих модулей
 - Сложности с повторным использованием модуля, поскольку большинству приложений именно такой модуль не нужен



Хороший дизайн



Loose coupling
High cohesion

- Правда, эти требования друг другу противоречат
- И следует это из самой природы ООП
- Так что придётся искать компромисс



Принципы SOLID

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle



Принципы SOLID



<http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/>



Single responsibility principle

- Принцип единственности ответственности
 - Каждый класс должен иметь единственную ответственность
 - Эта ответственность должна быть полностью инкапсулирована в этом классе
- Все сервисы класса должны быть направлены исключительно на обеспечение его ответственности
- Принцип обеспечивает высокую сфокусированность
 - Класс имеет единственную причину для изменения
 - При плохой сфокусированности модификация различных ответственностей приводит к комбинаторному взрыву



Single responsibility principle

```
public class Service{
    public static Image getImage(String fileName) {
        ...
    }
    public static void saveImage (Image img) {
        ...
    }
    public static void sendEmailMessage(String email,
        String msg) {
        ...
    }
    public static void selectDataFromTable(String connect,
        String tableName) {
        ...
    }
}
```



Single responsibility principle

```
public class ImageService{  
    public static Image getImage(String fileName) { ... }  
    public static void saveImage (Image img) { ... }  
}
```

```
public class EmailService{  
    public static void sendEmailMessage(String email,  
                                         String msg) {...}  
}
```

```
public class DataBaseService{  
    public static void selectDataFromTable(String connect,  
                                           String tableName) {  
  
        ...  
    }  
}
```



Single responsibility principle



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should



Open/closed principle

- Принцип открытости/закрытости
 - Программные сущности (классы, модули и т.п.) должны быть открыты для расширения, но закрыты для изменения
 - Сущности могут изменять своё поведение без изменения их исходного кода
- Принцип открытости/закрытости по Мейеру
 - Однажды разработанная реализация класса в дальнейшем требует только исправления ошибок
 - Новые или изменённые функции требуют создания нового класса
 - Рекомендуется наследование реализации
 - Реализация используется повторно
 - Тип не обязан использовать повторно
- Полиморфный принцип открытости/закрытости
 - Используются полностью абстрактные типы
 - Реализация может быть изменена, или многие реализации могут использоваться полиморфно
 - Рекомендуется наследование от полностью абстрактных типов
 - Тип используется повторно и не изменяется
 - Реализации должны соответствовать типу



Open/closed principle

```
public class MessageSender {  
    ...  
    public void send(Message msg) {  
        if(msg.getType == MessageType.SMS) sendSMS(msg);  
        else  
            if(msg.getType == messageType.EMAIL) sendEmail(msg);  
    }  
    ...  
}
```



Open/closed principle

```
public class MessageSender {  
    ...  
    public void send(Message msg) {  
        msg.send();  
    }  
    ...  
}
```

```
public abstract class MessageSender {  
    public abstract void send();  
}
```

```
public class SMSMessage  
    extends MessageSender {  
    public void send() {  
        ....}  
}
```

```
public class EMAILMessage  
    extends MessageSender {  
    public void send() {  
        ....}  
}
```



Open/closed principle



OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat



Liskov substitution principle

- Принцип подстановки Барбары Лисковой
 - Пусть $q(x)$ является свойством, верным относительно объектов x некоторого типа T . Тогда $q(y)$ должно быть верным для объектов y типа S , где S является подтипом типа T
 - (Р.С. Мартин) Использующие базовый тип функции должны иметь возможность использовать подтипы базового типа не зная об этом
 - (Б. Мейер) Дочерний класс не должен нарушать контракт родительского класса
- Требования к сигнатурам операций дочерних классов
 - Типы аргументов не должны быть уже
 - Типы возвращаемых значений не должны быть шире
 - Не должны выбрасываться новые типы исключений, кроме случаев, когда новое исключение является подтипом исключения из родительской сигнатуры
 - Область доступа операции не должна сужаться



Liskov substitution principle

```
public class Rectangle{  
    double height, width;  
    // методы set и get  
}
```

```
public class MyClass {  
    public static Rectangle zoomRectangle(Rectangle rect,  
        double zoomHeight, double zoomWidth) {  
        rect.setHeight(rect.getHeight() * zoomHeight);  
        rect.setWidth(rect.getWidth() * zoomWidth);  
        return rect;  
    }  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle();  
        r1.setHeight(5);  
        r1.setWidth(2);  
        zoomRectangle(r1, 1, 2);  
        System.out.println("Area: " + r1.getHeight() * r1.getWidth());  
    }  
}
```

Area: 20



Liskov substitution principle

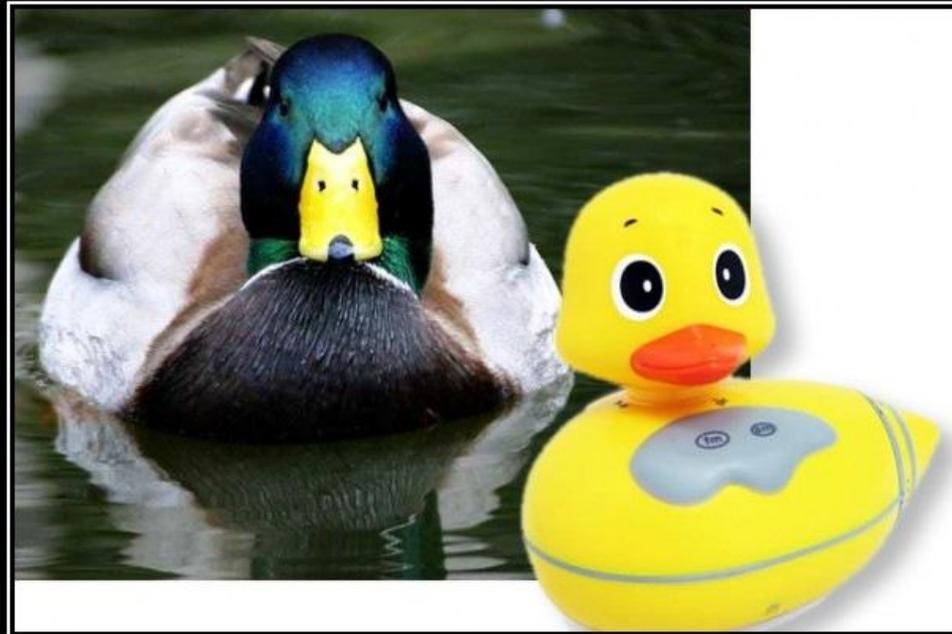
```
public class Square extends Rectangle{  
    public void setHeight(double h) {height = width = h;}  
    public void setWidth(double w) {height = width = w;}  
}
```

```
public class MyClass {  
    public static Rectangle zoomRectangle(Rectangle rect,  
        double zoomHeight, double zoomWidth) {  
        rect.setHeight(rect.getHeight() * zoomHeight);  
        rect.setWidth(rect.getWidth() * zoomWidth);  
        return rect;  
    }  
    public static void main(String[] args) {  
        Rectangle r1 = new Square();  
        r1.setHeight(5);  
        r1.setWidth(2);  
        zoomRectangle(r1, 1, 2);  
        System.out.println("Area: " + r1.getHeight() * r1.getWidth());  
    }  
}
```

Area: 16



Liskov substitution principle



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

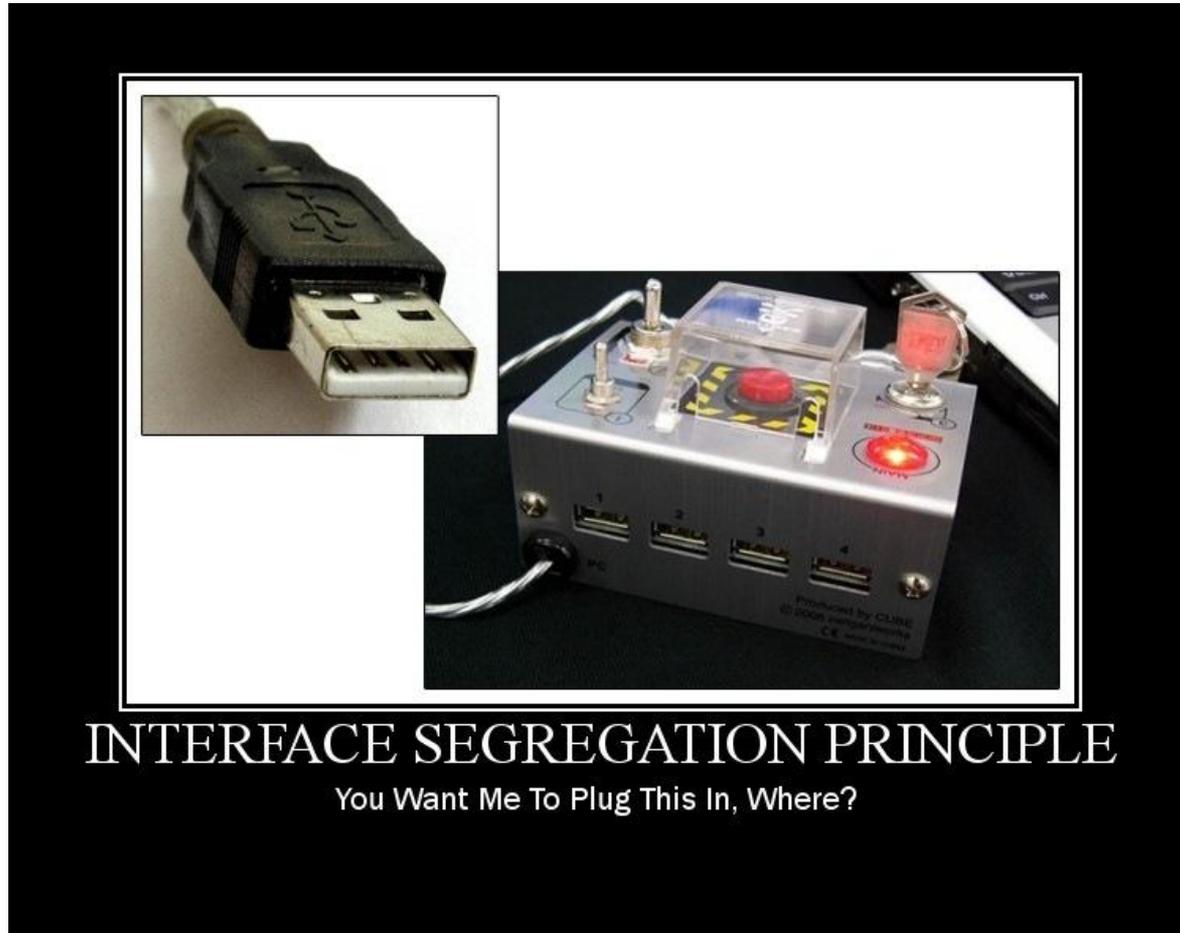


Interface segregation principle

- Принцип разделения интерфейсов
 - Клиенты не должны зависеть от методов, которые они не используют
 - Интерфейсы должны быть сфокусированными
- Большие интерфейсы должны разделяться на более мелкие и узкоспециальные
- Такие интерфейсы (полностью абстрактные типы) скорее имеют роль тегов, чем организуют свою иерархию наследования



Interface segregation principle



Dependency inversion principle

- Принцип инверсии зависимостей
 - 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
- Высокоуровневые и низкоуровневые компоненты взаимодействуют через абстрактный интерфейс
 - Интерфейс описывается как часть высокоуровневого компонента
 - Получается, что низкоуровневый компонент зависит от высокоуровневого, а не наоборот
 - Это позволяет заменять низкоуровневые компоненты, не изменяя высокоуровневых
- Следование принципу снижает связанность
- Значительно упрощает разработку сложных систем



Dependency inversion principle

```
public class Man {  
    private Car car;  
    public void gotoWork() {  
        car = new Car();  
        car.go(100, 200);  
        // какие-то действия  
    }  
}
```

```
public class Car {  
    private int currentX, currentY;  
    public void go(int x, int y) {  
        currentY = y;  
        currentX = x;  
    }  
    ...  
}
```



Dependency inversion principle

```
public class Man {
    private Mode mode;
    public void setModeOfMovement(Mode newMode) {
        this.mode = newMode;
    }
    public void gotoWork() {
        mode.go(100, 200);
        // какие-то действия
    }
}
```

```
public interface Mode {
    void go(int x, int y);
}
```

```
public class Car implements Mode {
    public void go(int x, int y) {.....};
}
```

```
public class Teleport implements Mode {
    public void go(int x, int y) {.....};
}
```



Dependency inversion principle



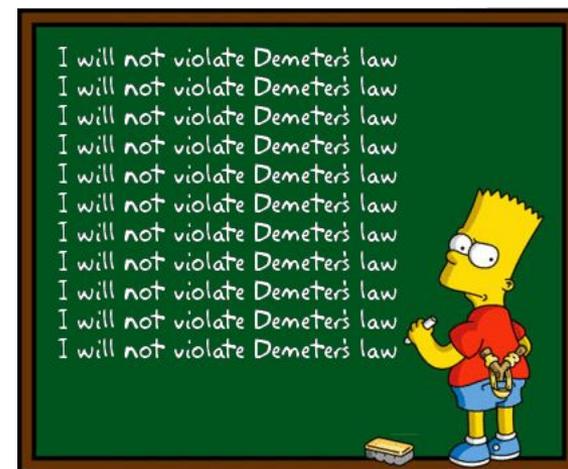
DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?



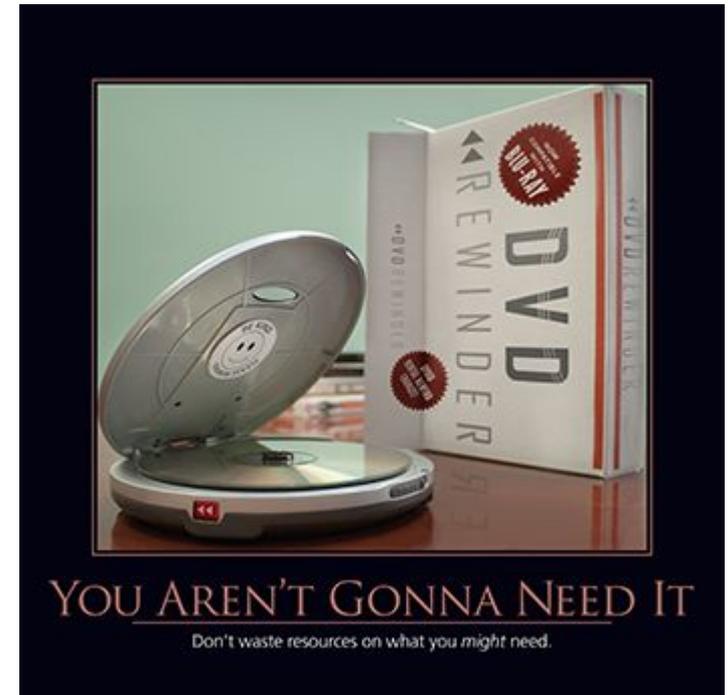
Law of Demeter

- Закон Деметры (принцип наименьшего знания)
 - Каждый модуль должен обладать ограниченным знанием о других модулях: должен знать только о модулях, которые имеют к нему непосредственное отношение
 - Каждый модуль должен взаимодействовать только с известными ему модулями и не должен «разговаривать с незнакомыми»
 - Каждый модуль должен обращаться только к своим непосредственным друзьям
- Объект А может вызвать сервис (метод) объекта В, но не может использовать объект В для получения доступа к объекту С, чтобы использовать его методы
- Метод `m()` объекта `O` может вызывать только методы следующих объектов
 - Сам объект `O`
 - Объекты-параметры метода `m()`
 - Любые объекты, созданные в ходе выполнения `m()`
 - Объектов, непосредственно ассоциированных с `O`
 - Глобальные переменные, доступные `O` в контексте `m()`



Принцип YAGNI

- You ain't gonna need it
 - Реализуйте что-то, только если оно вам действительно нужно
 - Не реализуйте что-то, использование чего вы только предвидите
- Требуется разумный баланс со здравым СМЫСЛОМ

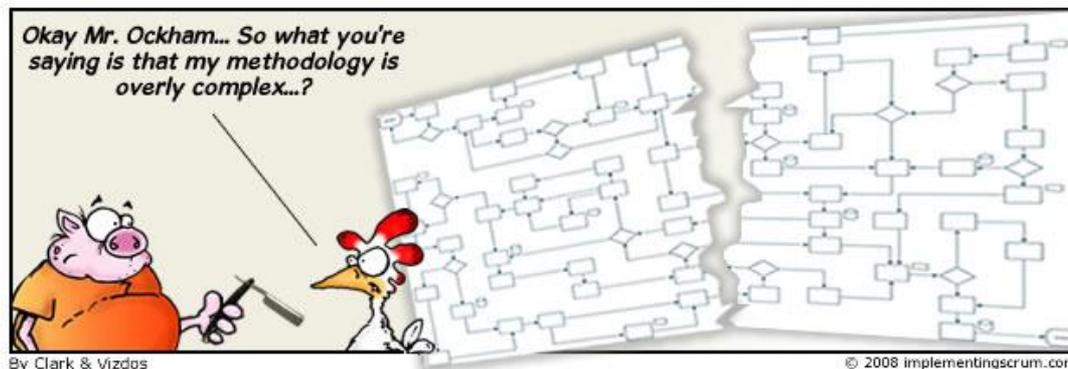


Принцип KISS

- Keep it simple, stupid
 - Keep it simple and stupid
 - Keep it short and simple
- Простота должна быть одной из основных целей в ходе разработки
- Следует уходить от необоснованных сложностей



Keep. It. Simple. Stupid.



Спасибо за внимание!

Дополнительные источники

- Мейер, Б. Объектно-ориентированное конструирование программных систем [Текст] / Бертран Мейер. – М. : Русская редакция, 2005. – 1204 с.
- Мейер, Б. Почувствуй класс: учимся программировать хорошо с объектами и контрактами [Текст] / Бертран Мейер. – М. : Интернет-университет информационных технологий, 2011. – 776 с.
- Мартин, Р.К. Быстрая разработка программ. Принципы, примеры, практика [Текст] / Роберт К. Мартин, Джеймс В. Ньюкирк, Роберт С. Косс. – М. : Издательский дом «Вильямс», 2004. – 752 с.
- Мартин, Р. Чистый код. Создание, анализ и рефакторинг [Текст] / Роберт Мартин. – СПб : Питер, 2011. – 464 с.
- Мартин, Р. Идеальный программист. Как стать профессионалом разработки ПО [Текст] / Роберт Мартин. – СПб : Питер, 2012. – 224 с.

