Основные принципы объектно-ориентированного проектирования

Введение

- Сложность ПО
- Языки предоставляют абстракции для решения проблем
- Мощность и результирующая сложность
- Разница между описанием проблемы и решением проблемы

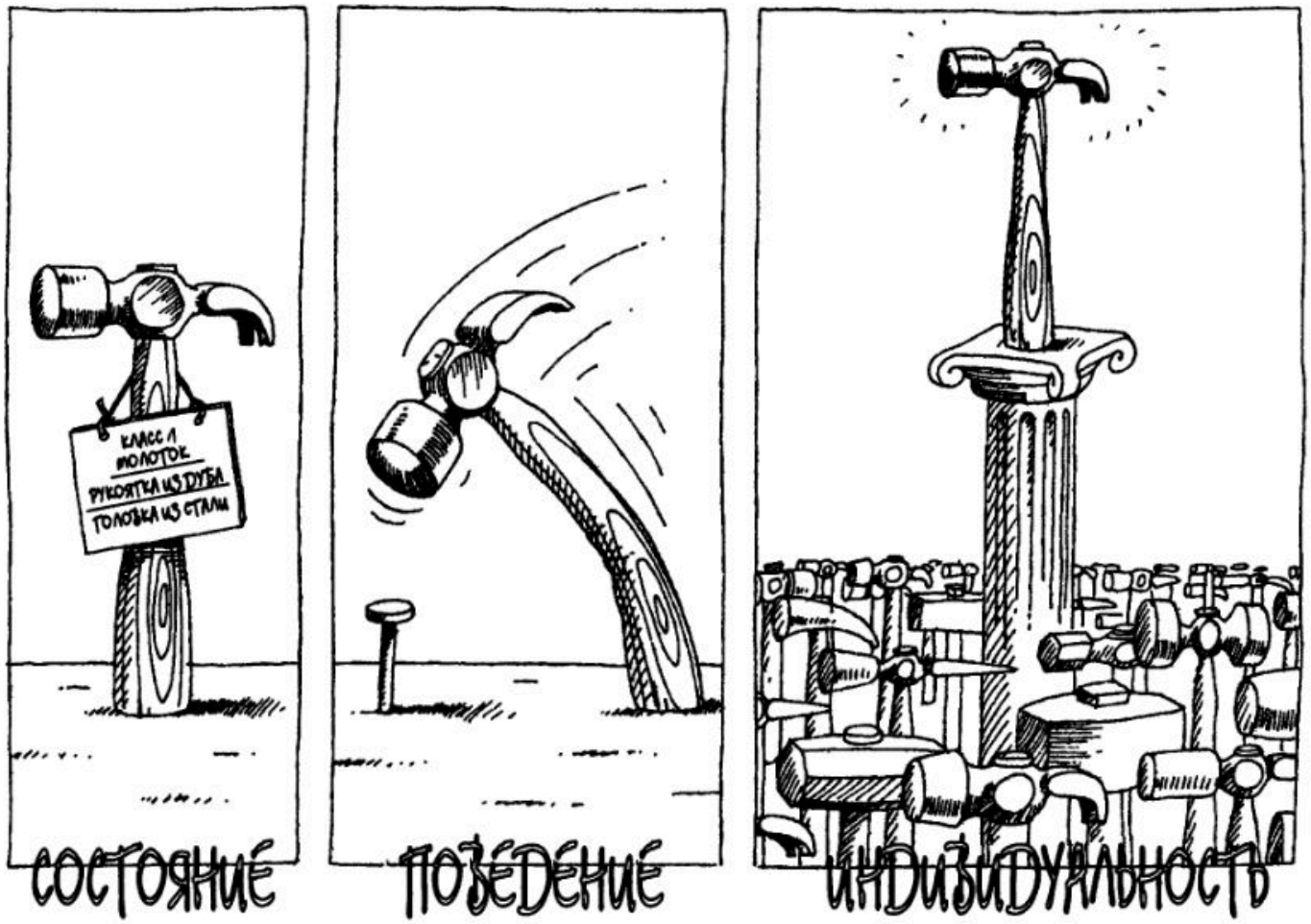
Стили программирования

- Процедурно-ориентированный (алгоритмы)
- Объектно-ориентированный (объекты)
- Функционально-ориентированный (функции)
- Логико-ориентированный (предикаты)
- Ориентированный на ограничения (инварианты)

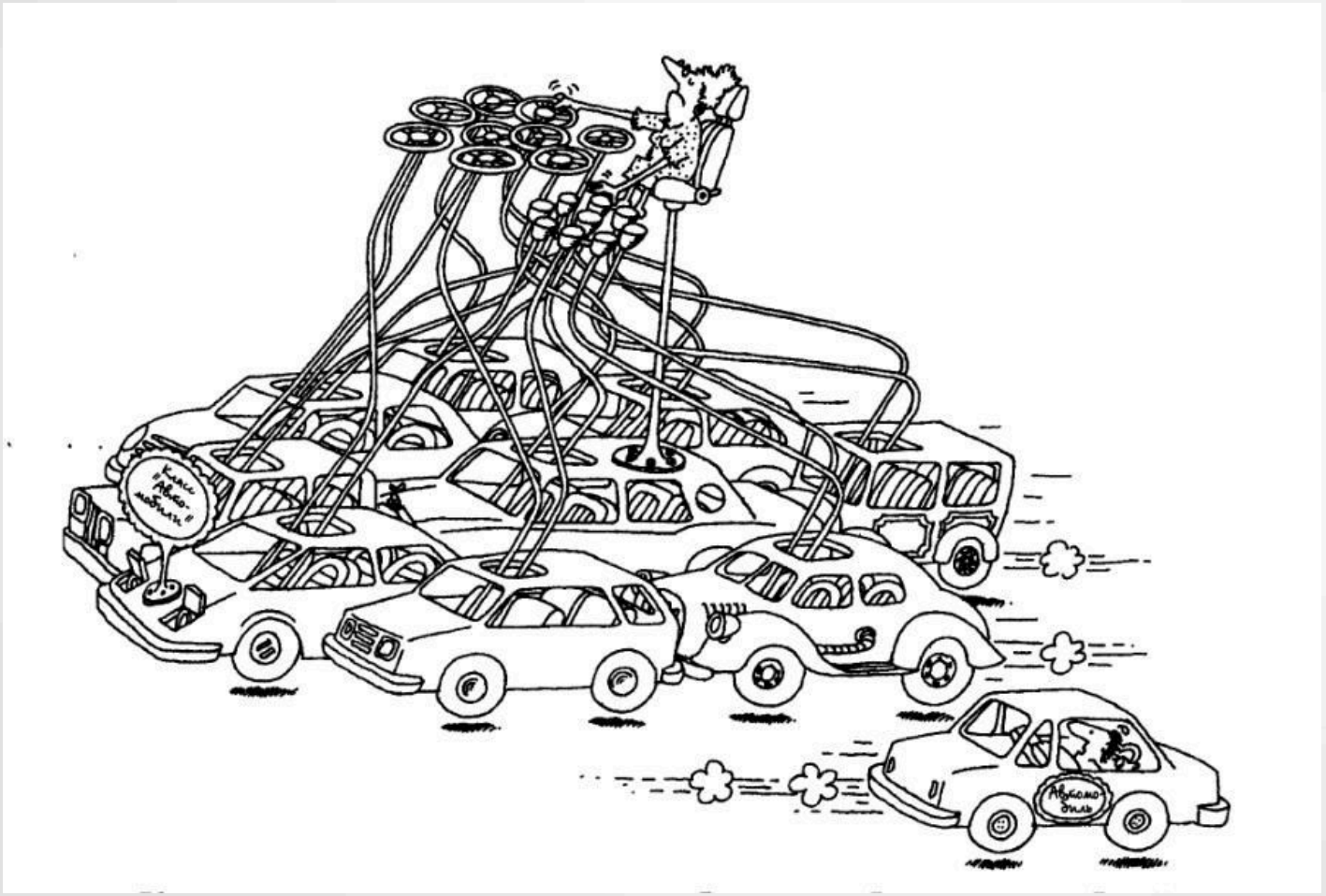
Стили программирования - 2

- Подходят для различных задач
- Императивность и декларативность
- Языки сочетают в себе разные стили

Объекты


КЛАСС 1
МОЛОТОК
РУКОЯТКА ИЗ ДУБА
ГОЛОВКА ИЗ СТАЛИ
СОСТОЯНИЕ
ПОВЕДЕНИЕ
ИНДИВИДУАЛЬНОСТЬ

Классы

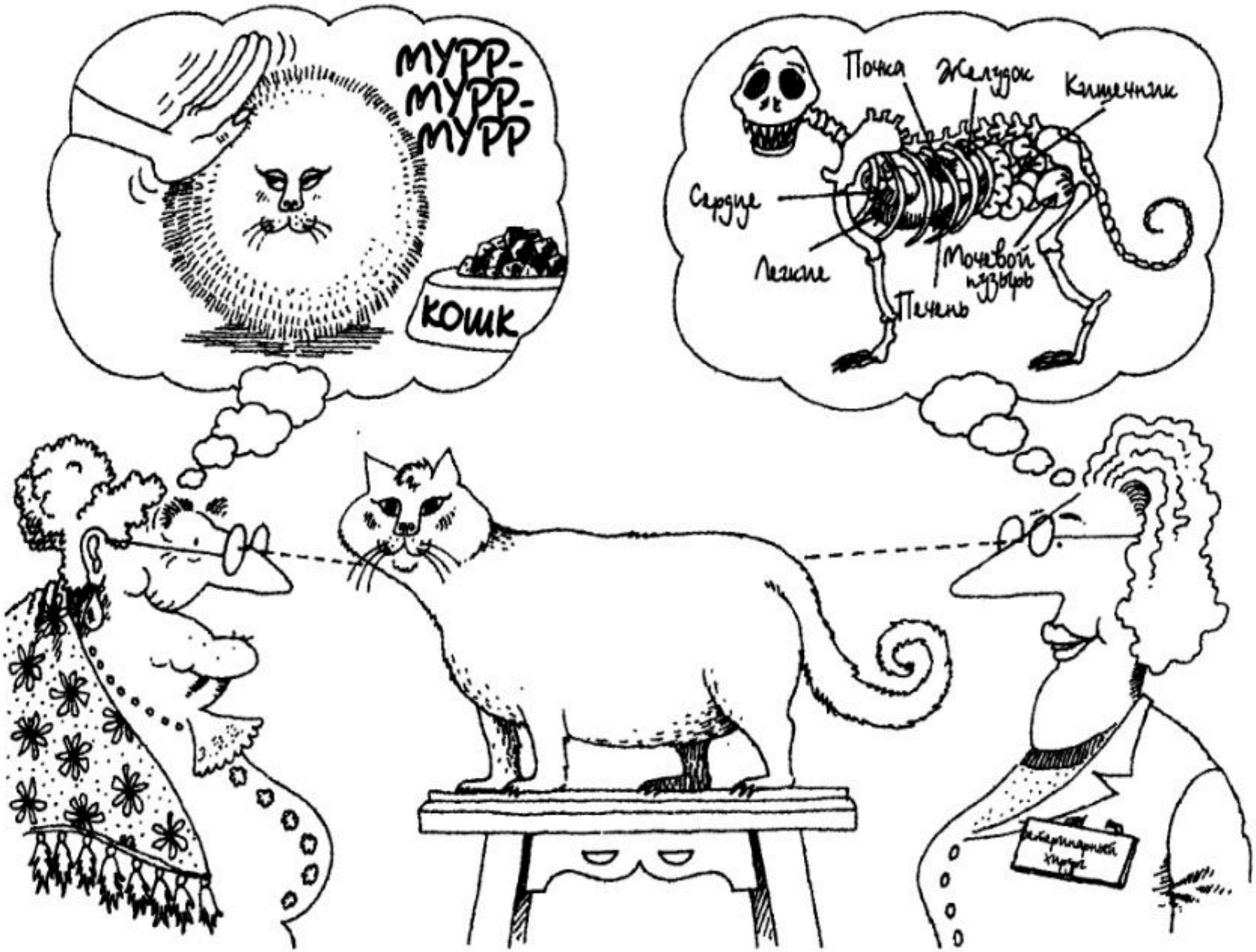


Основные характеристики ООП

- Абстракция
- Инкапсуляция
- Модульность
- Иерархия

Основные характеристики ООП

- **Абстракция**
- Инкапсуляция
- Модульность
- Иерархия


МУРР-МУРР-МУРР
КОШК
Почка
Желудок
Кишечник
Сердце
Легкие
Печень
Мочевой пузырь

Абстракция

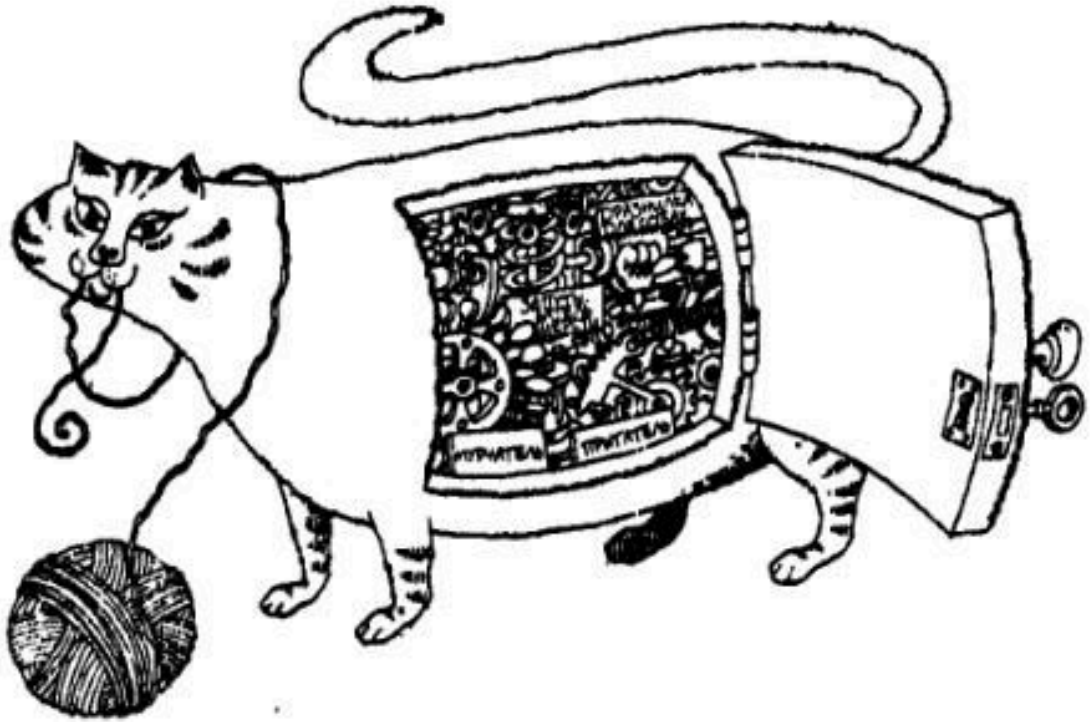
- Один из основных методов борьбы со сложностью
- Оставляет существенные признаки и отбрасывает несущественные
- Принцип минимальных обязательств
- Принцип наименьшего удивления
- “Дырявые” абстракции

Виды абстракций

- Абстракция сущности
- Абстракция действия
- Абстракция виртуальной машины
- Произвольная абстракция

Основные характеристики ООП

- Абстракция
- **Инкапсуляция**
- Модульность
- Иерархия



Инкапсуляция

- Отделение реализации абстракции от ее поведения
- Интерфейс и его реализация
- Определяет четкие границы между разными абстракциями
- Необходимое условие для развития

Основные характеристики ООП

- Абстракция
- Инкапсуляция
- **Модульность**
- Иерархия


Инструкция по сборке
Сердце
Клетки кожи
Конструктор
ДЕТАЛИ КОШКИ
Задняя нога
Мускатель
Мускатель

Модульность

- Разделение системы на внутренне связные, но слабо связанные между собой модули
- Возможные причины: скорость разработки, разные жизненные циклы, разные команды
- Упрощает повторное использование
- Java: package and jar

Иерархия

- Упорядочивание абстракций
- Структура классов (“is-a”)
- Структура объектов (“has-a”)

SOLID principles

- Принципы объектно-ориентированного проектирования
- Предложены Робертом Мартином
- Аббревиатура SOLID появилась позже

SOLID principles

- Single Responsibility Principle
- Open for extension, closed for modification
- Liskov Substitution Princliple
- Interface Segregation Principle
- Dependency Inversion Princliple


SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

SOLID

- Single Responsibility Principle
- Каждый класс должен выполнять единственную функцию
- Тесно связан с именованием
- “Волшебный” класс


OPEN CLOSED PRINCIPLE
Open Chest Surgery Is Not Needed When Putting On A Coat

SOLID

- Open for Extension, Closed for modification
- Поведение программного компонента может быть расширено
- Исходный код не должен измениться

Пример

```
public class AreaCalculator  {
    public double area(Rectangle[] shapes) {
        double area = 0;
        for (Rectangle rectangle : shapes)  {
            area +=
  rectangle.getWidth()*rectangle.getHeight();
        }
        return area;
    }
 }
```

Пример

```
public double area(Object[] shapes)  {
        double area = 0;
        for (Object shape : shapes)    {
            if (shape instanceof Rectangle) {
                Rectangle rectangle = (Rectangle) shape;
                area += rectangle.getWidth() * rectangle.getHeight();
            } else {
                Circle circle = (Circle) shape;
                area += circle.getRadius()*circle.getRadius()*Math.PI;
            }
        }  return area;
    }
```

Пример

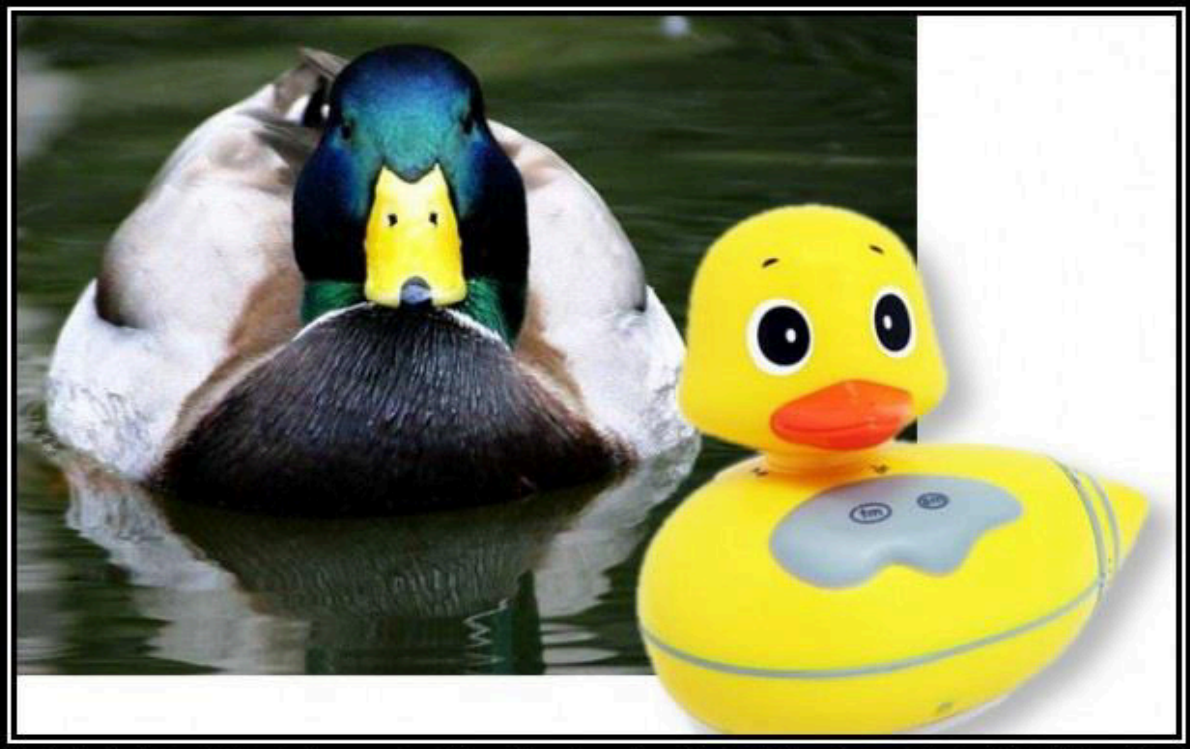
```
abstract class Shape {
    public abstract double area();
}
```

Пример

```
class Circle extends Shape {
    private double radius;
    public double getRadius() {
        return radius;
    }
     @Override
    public double area() {
        return radius* radius*Math.PI;
    }
}
```

Пример

```
public double area(Shape[] shapes)    {
    double area = 0;
    for (Shape shape : shapes)  {
       area += shape.area();
    }
    return area;
}
```



LISKOV SUBSTITUTION PRINCIPLE

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

SOLID

- Liskov substitution principle
- Введен Барбарой Лискова (1987)
- Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом.

Пример (1/2)

```
public class Rectangle {
    private double width, height;
    private double topLeftX,topLeftY;

    public double getWidth() { return width;}
    public void setWidth(double width) {this.width = width;}

    public double getHeight() {return height;}
    public void setHeight(double height) {this.height = height;}
    }
```

Пример (1/2)

```
class Square extends Rectangle { }
```

Инварианты

- Square: width == height

  тогда как

- Rectangle: width не зависит от height

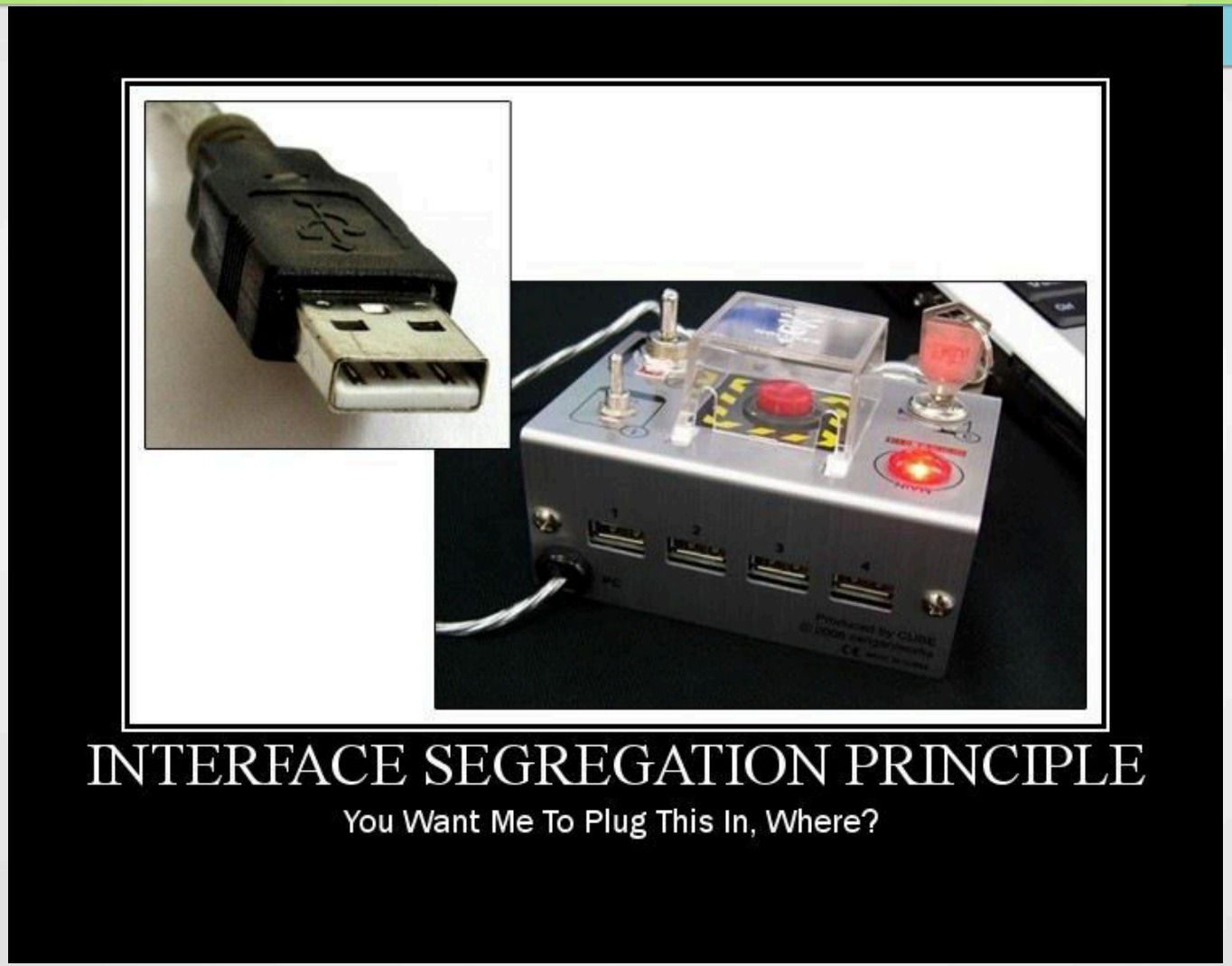
Расчет площади

```
public void testSquare(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    assertEquals(r.area(), 20);
}
```

- Модель должна рассматриваться в контексте общей структуры
- С точки зрения поведения: Square is not Rectangle

Design by contract

- Предусловия не могут быть усилены в подклассе
- Постусловия не могут быть ослаблены в подклассе
- Нет дополнительных исключений, кроме подклассов


INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

SOLID

- Клиенты не должны попадать в зависимость от тех методов интерфейса, которыми они не пользуются
- Универсальные интерфейсы должны быть разделены на несколько специализированных

Design by contract

- Предусловия не могут быть усилены в подклассе
- Постусловия не могут быть ослаблены в подклассе
- Нет дополнительных исключений, кроме подклассов

Пример

```
public interface Animal {
    void fly();
    void run();
    void bark();
}
```

Пример

```
public class Bird implements Animal {
   public void bark() { /* do nothing */ }
   public void run() {
      // write code about running of the bird
    }
    public void fly() {
        // write code about flying of the bird
    }
}
```

Пример

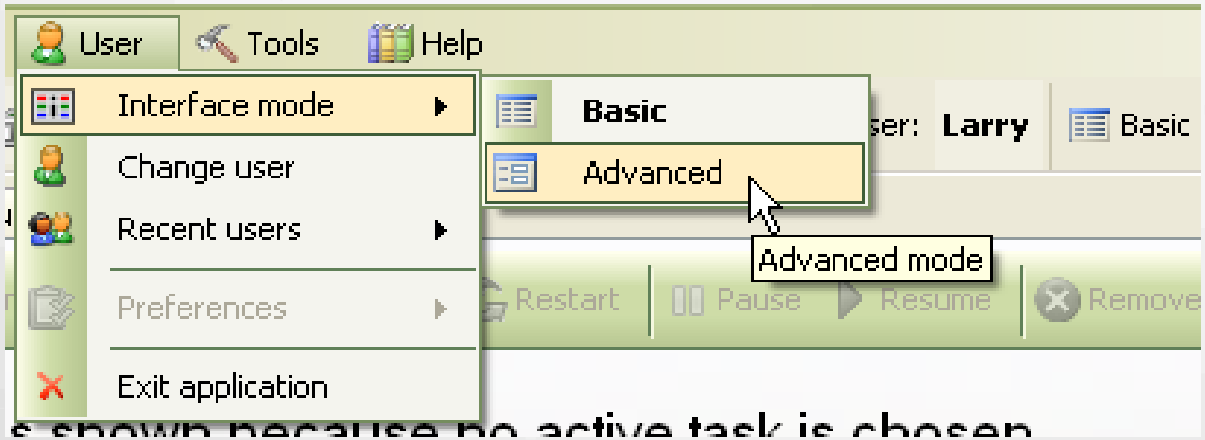
```
public class Cat implements Animal {

    public void fly() { throw new Exception("Undefined cat
    property"); }


    public void bark() { throw new Exception("Undefined cat
    property"); }


    public void run() {

        // write code about running of the cat

    }

}
```

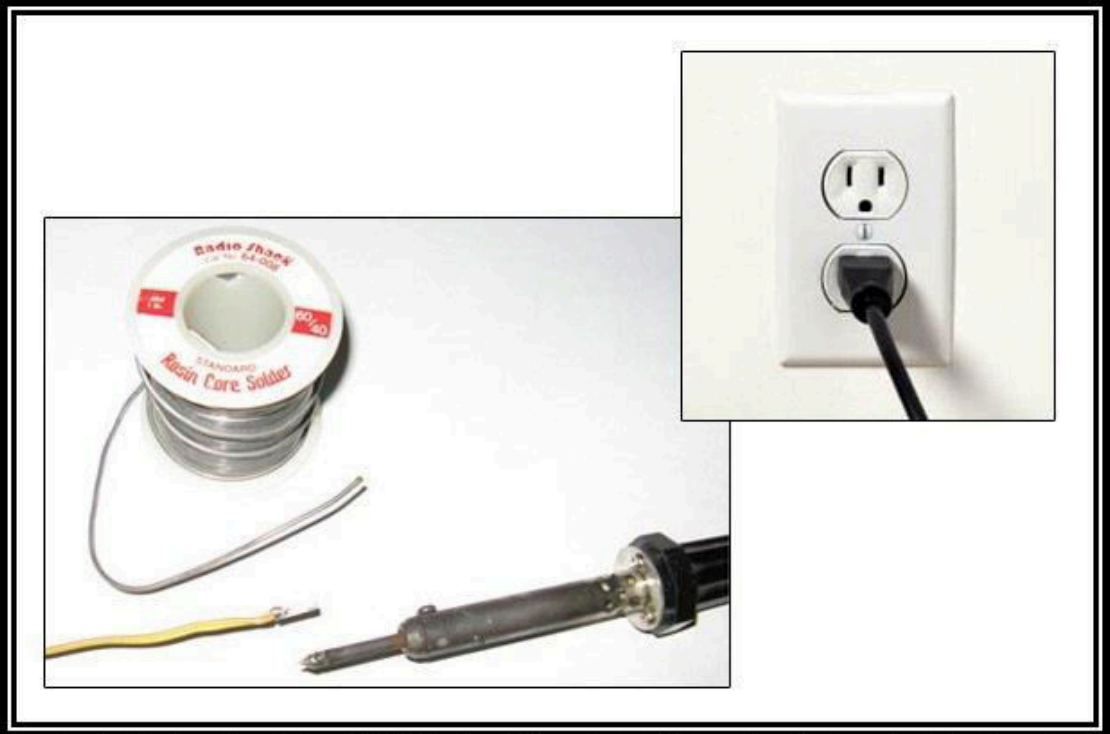
Пример

```
public interface Flyable {
  void fly();
}
public interface Runnable {
    void run();
}
public interface Barkable {
    void bark();
}
```

Другие примеры

- На объекте могут вызываться разные методы в разные моменты жизненного цикла: разные интерфейсы

- 



DEPENDENCY INVERSION PRINCIPLE

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

SOLID

- The Dependency Inversion
- 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

Повторное использование

Наследование vs композиция

Картинки

- ООП: Гради Буч (ООА и ООП с примерами приложений на C++ - 3е издание – изд. Символ)
- SOLID: Derick Bailey