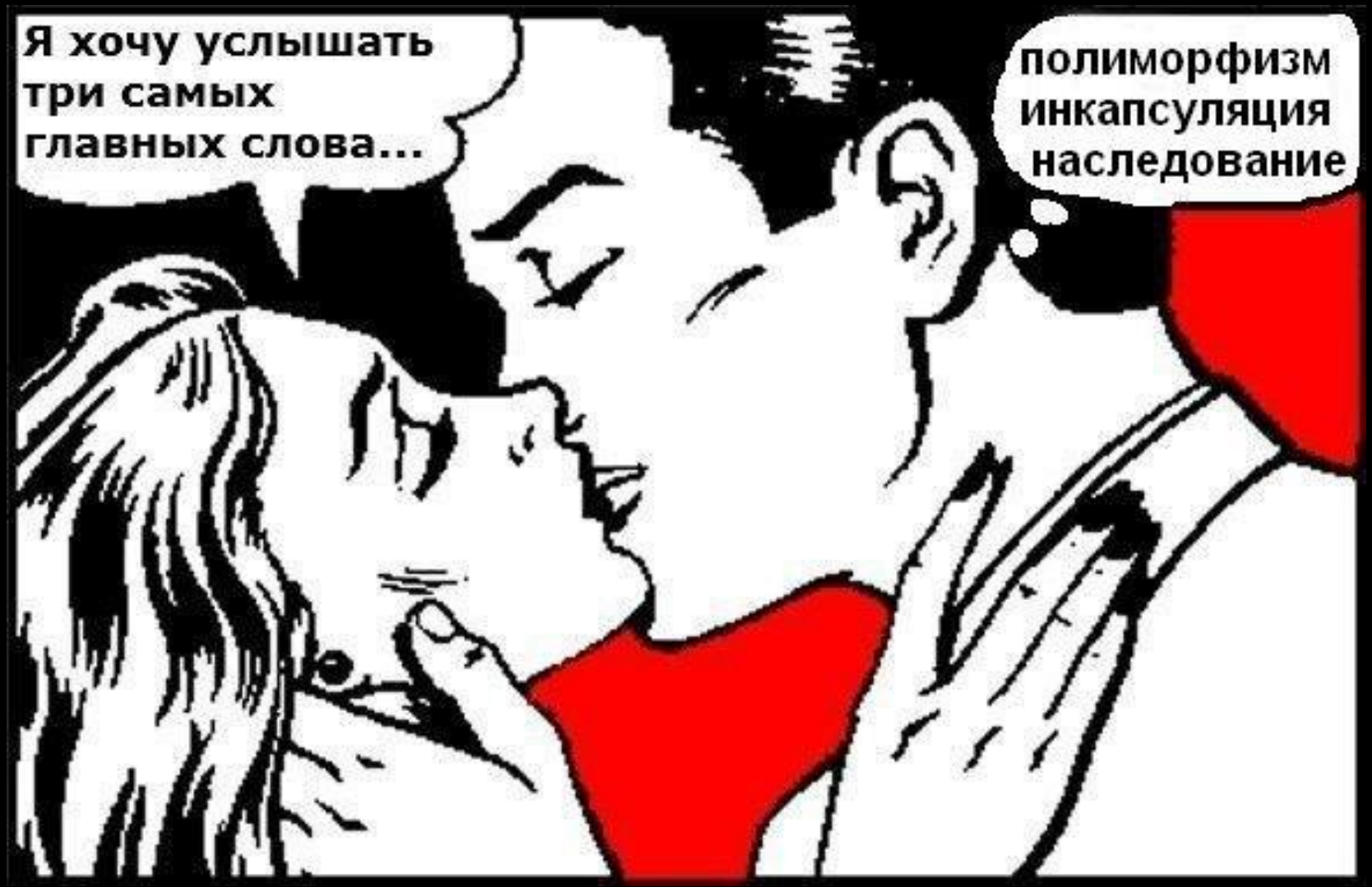
SOLID

ООП

- Переиспользуемость
- Читаемость
- Отражение реальности
- Документируемость
- Тестирование (TDD)

ООП


Я хочу услышать
три самых
главных слова...
полиморфизм
инкапсуляция
наследование

ООП

Гибкая разработка

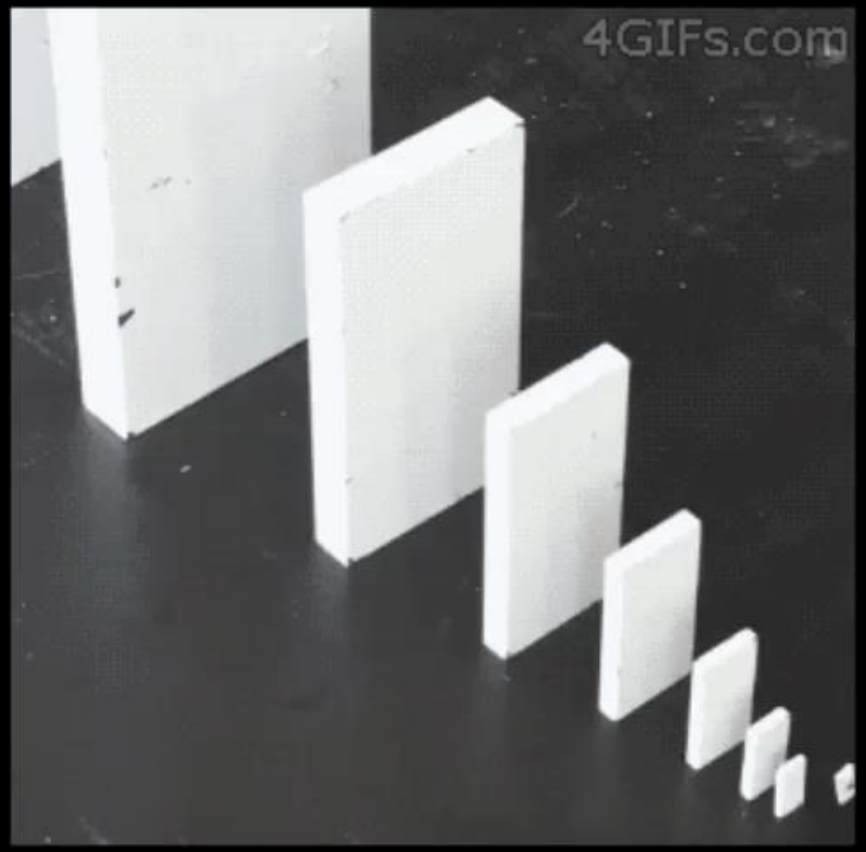
- Итеративная разработка
- Динамическое формирование требований
- Быстрые коммуникации
- Самоорганизация команды
- Сосредоточение на текущем состоянии системы

Гибкая архитектура

- Жесткость
- Хрупкость
- Вязкость
- Ненужная сложность
- Дублирование
- Непрозрачность

Жесткость

- Изменение одного модуля программы влечет изменение других



Хрупкость

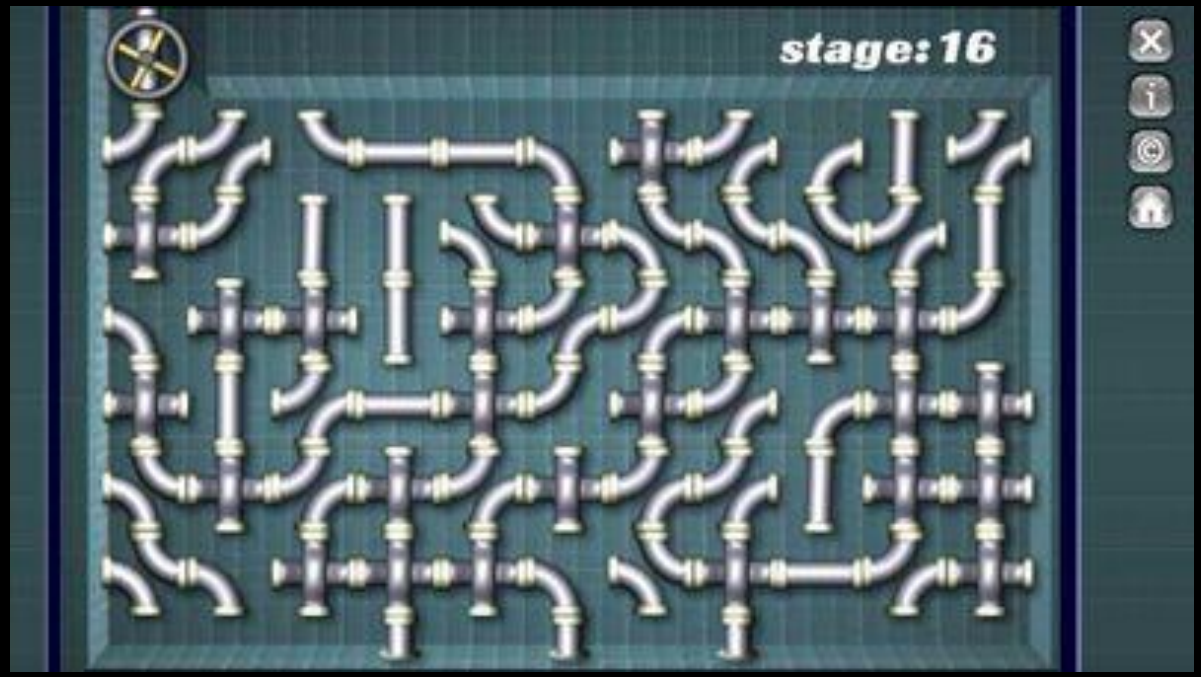
- Изменения одного модуля ведут к краху других

Вязкость

- Когда хаки делать легче чем архитектурные изменения

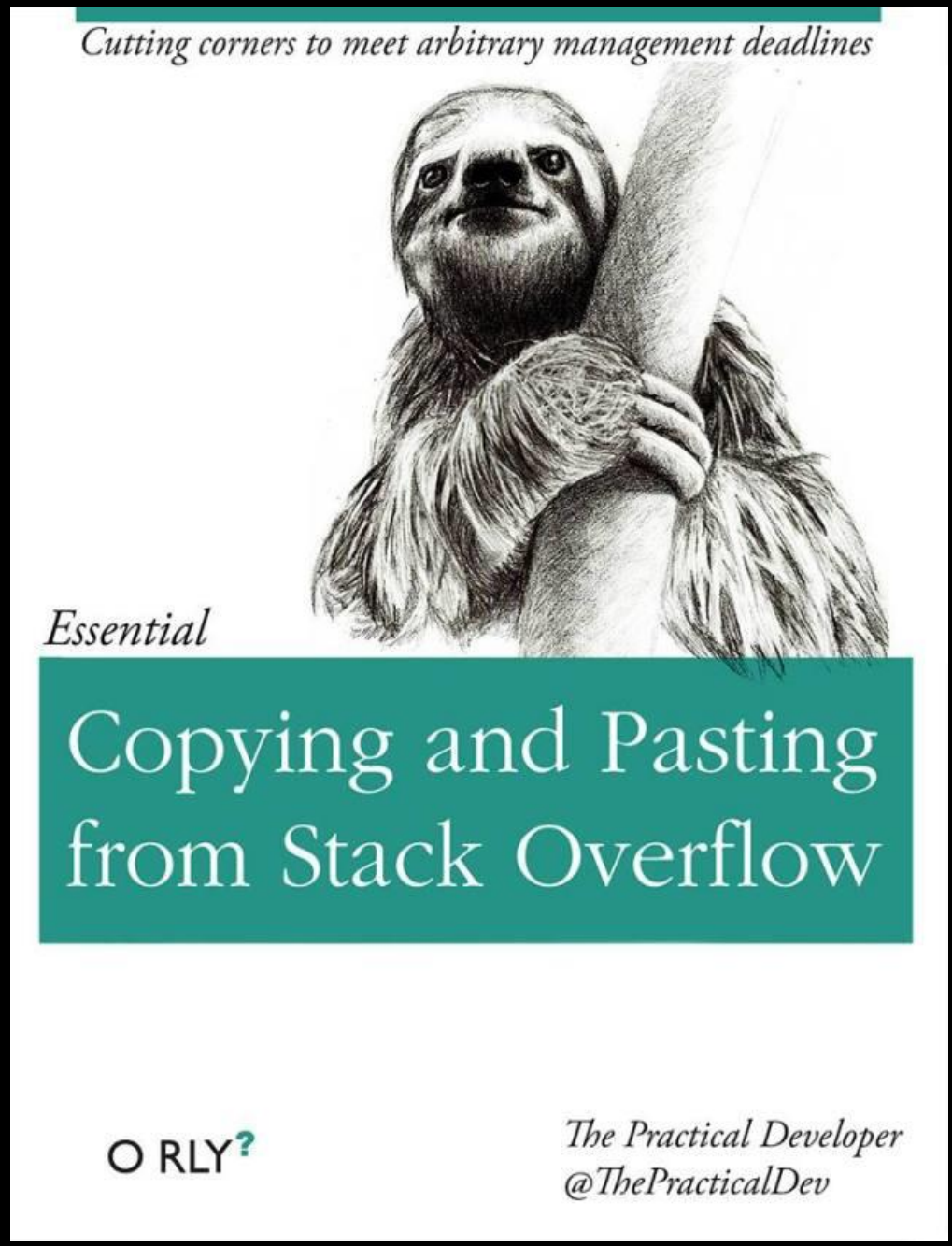
Ненужная сложность

- Неиспользуемые компоненты



Дублирование

- Дублирование



Непрозрачность

- Трудность понимания, необходимость комментариев

SOLID

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

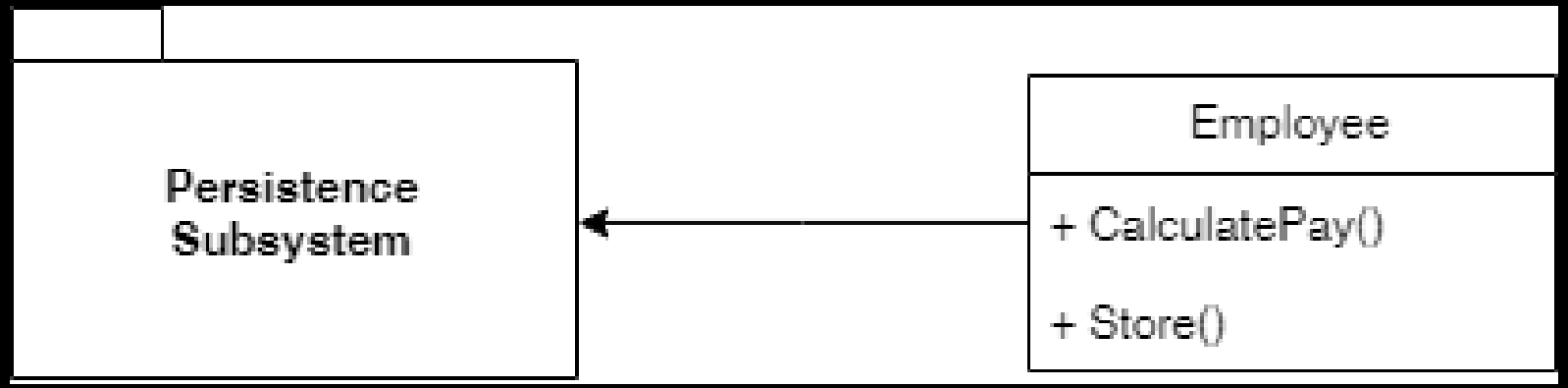
Single responsibility principle

- У класса должна быть только одна причина для изменения

Single responsibility principle

```
public class Rectangle
{
    public void Draw(){}
    public double GetLength(){}
}
```

Single responsibility principle



Open/closed principle

- Классы должны быть открыты для расширения, но закрыты для модификации

Open/closed principle

```
--shape.h---------------------------------
enum  ShapeType  {circle,  square};
struct  Shape {
  ShapeType  itsType;
};
--circle.h--------------------------------
struct  Circle {
  ShapeType  itsType;
  double  itsRadius;
  Point  itsCenter;
};
void  DrawCircle(struct  Circle*);
--square.h--------------------------------
struct  Square {  ShapeType  itsType;
  double  itsSide;
  Point  itsTopLeft;
};
void  DrawSquare(struct  Square*);
```

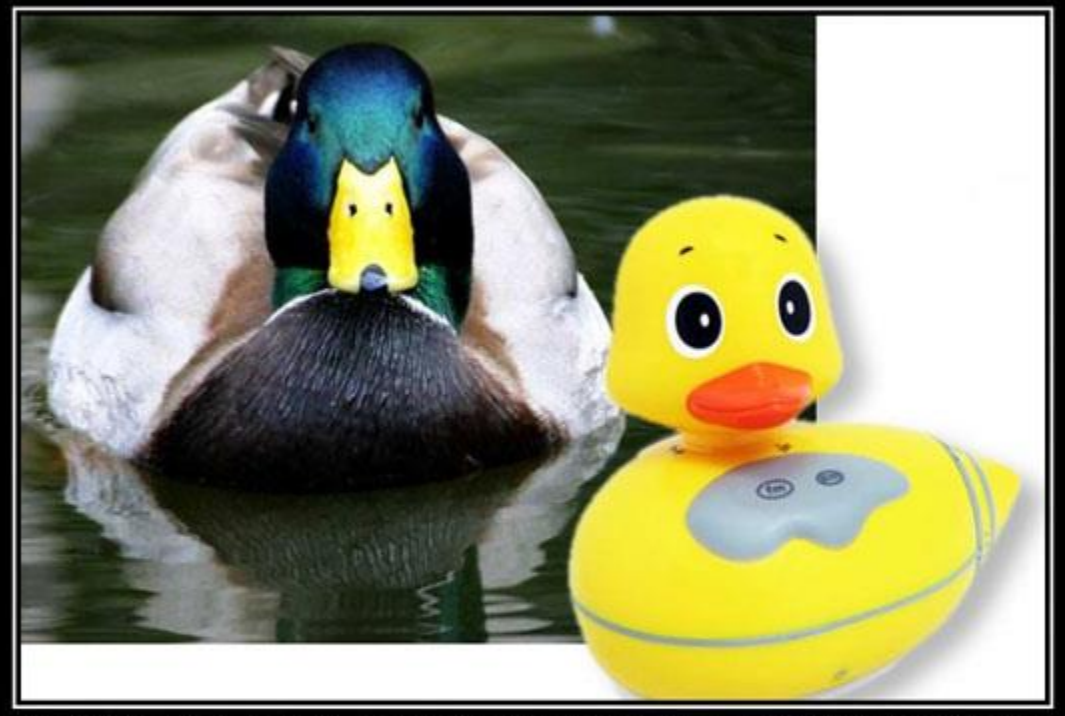
```
--drawAllShapes.cc-----------------------
typedef  struct  Shape  *ShapePointer;
void  Draw (ShapePointer  list[],  int  n){
  int  i;
  for  (i=0;  i<n;  i++)
  {
    struct  Shape*  s  =  list[i];
    switch  (s->itsType)
    {
      case  square:
        DrawSquare((struct  Square*)s);
        break;
      case  circle:
        DrawCircle((struct  Circle*)s);
                break;
     }
}
```

Open/closed principle

```
public interface IShape {
    void Draw();
}

public class Square : IShape
{
    public void Draw() {
        // draw a square
    }
}

public class Circle : IShape{
    public void Draw() {
        // draw a circle
    }
}
```

```
public class Drawer {
    public void Draw(List<IShape> shapes){
        foreach(var shape in shapes)
            shape.Draw();
    }
}
```

Liskov substitution principle



- Вместо базового типа можно использовать любой его подтип

Liskov substitution principle

```
public class Rectangle
{
    public double Width { get;set; }
    public double Height { get;set; }
}

public class Square : Rectangle
{
}
```

Liskov substitution principle

```
public class Square : Rectangle
{
    public new double Width
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public new double Height
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }
}
```

Liskov substitution principle

```
void SomeAction(Rectangle r)
{
    r.Width = 32;
}
```

Liskov substitution principle

```
public class Rectangle
{
    public virtual double Width { get;set; }
    public virtual  double Height { get;set; }
}
```

```
public class Square : Rectangle
{
    public override double Width
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public override double Height
    {
        set
        {
            base.Width = value;
            base.Height = value;
        }
    }
}
```

Liskov substitution principle

```
void R(Rectangle r)
{
    r.Width = 5;
    r.Height = 5;
    if (r.Area() != 20)
        throw new Exception("Неправильная площадь!");
}
```

Следствие из LSP: невозможно установить правильность модели, рассматриваемой изолированно. Правильность модели можно выразить только в терминах её клиентов.

Dependency inversion principle

- Абстракции не зависят от деталей, детали зависят от абстракций

Dependency inversion principle

```
public class Button
{
    private Lamp lamp;
    public void Poll()
    {
        if (/* какое-то условие */)
            lamp.TurnOn();
    }
}
```

Dependency inversion principle

```
public interface IButtonServer
{
    void TurnOn();
    void TurnOff();
}

public class Lamp : IButtonServer
{
    public void TurnOn() {}
    public void TurnOff() {}
}
```

```
public class Button
{
    private readonly IButtonServer buttonServer;

    public Button(IButtonServer buttonServer)
    public void Poll()
    {
        if (/* какое-то условие */)
            buttonServer.TurnOn();
    }
}
```

Interface segregation principle

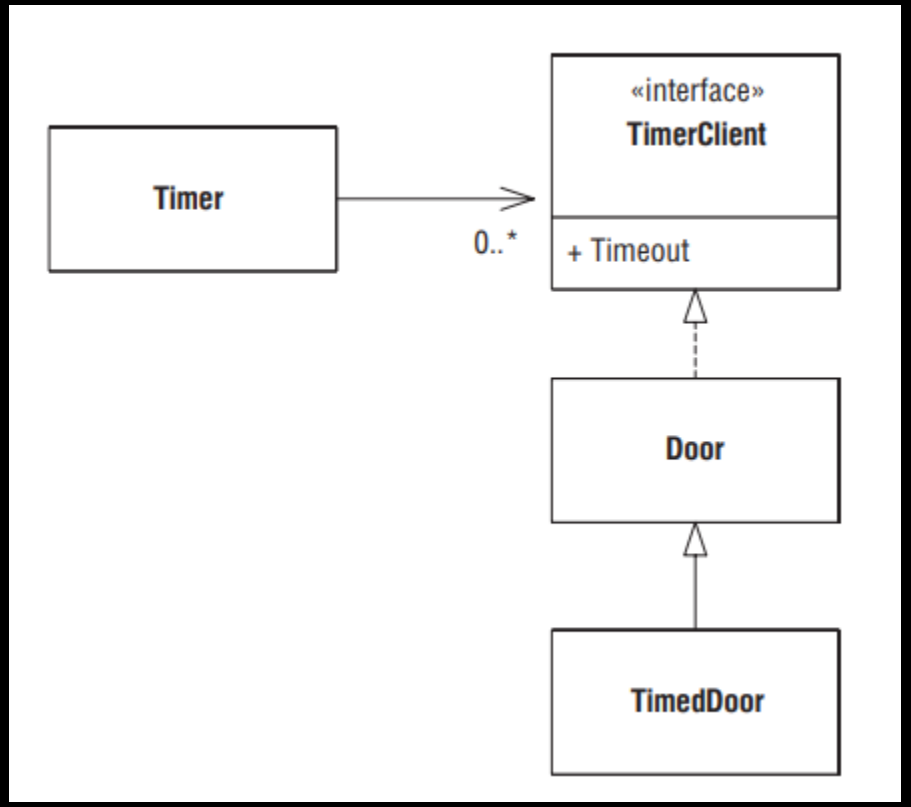
- Клиенты не должны вынужденно зависеть от методов, которыми не пользуются

Interface segregation principle

```
public interface Door
{
    void Lock();
    void Unlock();
    bool IsDoorOpen();
}
```

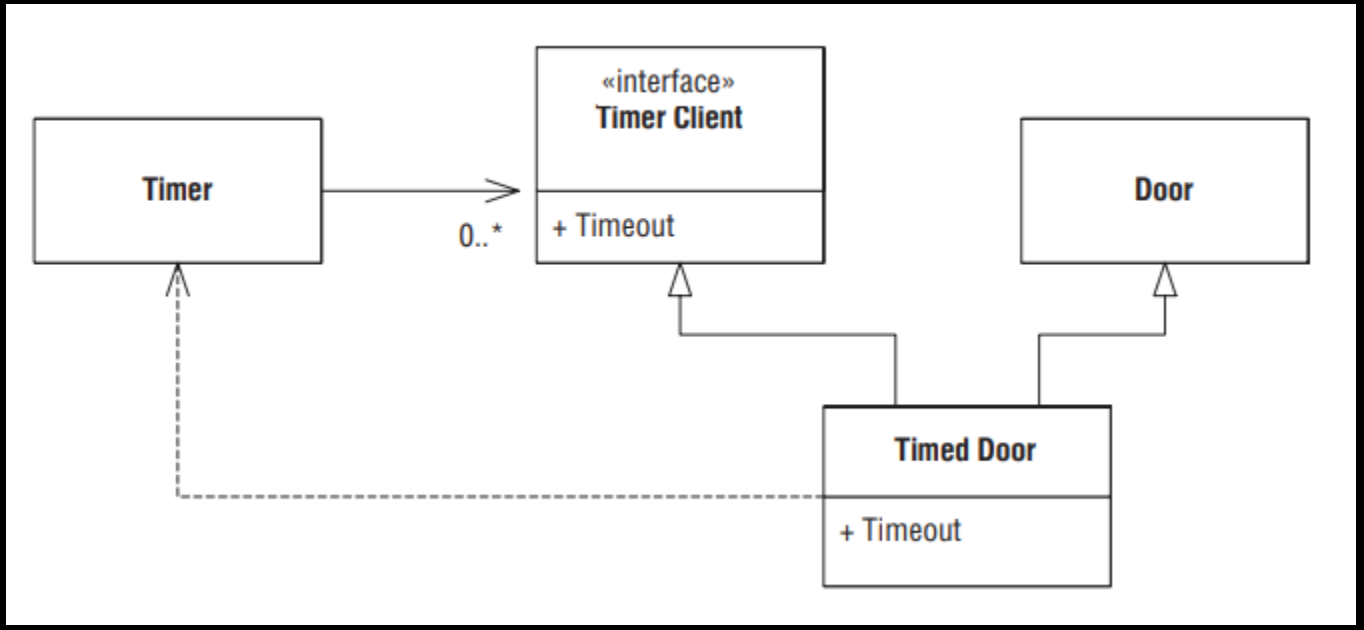
Interface segregation principle

```
public class Timer
{
    public void Register(int timeout, TimerClient client)
    {/* код */}
}

public interface TimerClient
{
    void TimeOut();
}

public class Door : TimerClient
{
}

public interface TimedDoor : Door
{
    void DoorTimeOut(int timeOutId);
}
```



Interface segregation principle

```
public interface TimedDoor : Door, TimerClient
{
}
```



Что почитать?

Р. С. Мартин

М. Мартин

Принципы, паттерны и методики гибкой разработки на языке C#