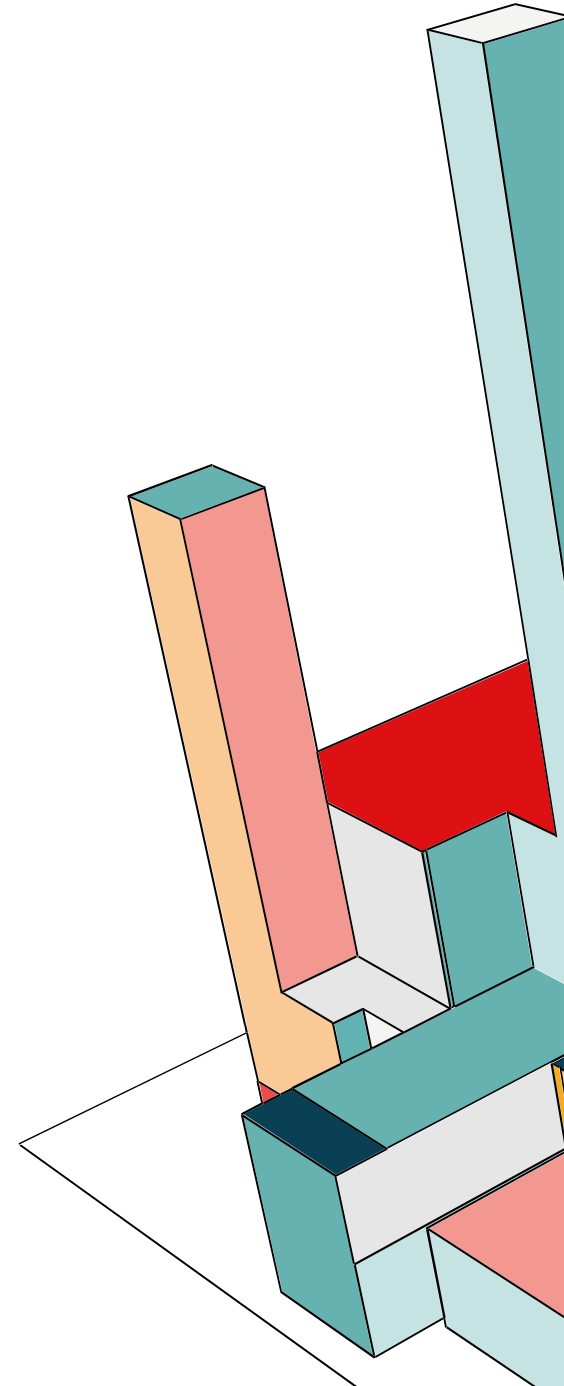




# **К**ОНСТРУИРОВАНИЕ **П**РОГРАММНОГО **О**БЕСПЕЧЕНИЯ

# ПЛАН ЛЕКЦИИ

- О курсе
- Парадигмы программирования
- Вспомним ООП. Отношения между классами и объектами
- Интерфейсы vs Абстрактные классы
- Принципы дизайна: SOLID vs GRASP, а также *KISS, DRY, YAGNI, BDUF, APO и бритва Оккама*
- Душные Итоговые советы и рекомендации



# О КУРСЕ



После  
**ASSEMBLER**

Изучающий  
**КПО**



# О СЛОЖНОСТЯХ КУРСА

|                                  |   |
|----------------------------------|---|
| Принципы проектирования          | SOLID: пять принципов ООП   |
|                                  | Принципы n-tier приложений  |
|                                  | DAL: построение слоя доступа к данным                             |
|                                  | Инверсия управления. IoC-контейнер                                |
| Архитектурные шаблоны и паттерны | MV-паттерны: MVC, MVP   |
|                                  | MV-паттерны: MVVM   |
|                                  | GoF: порождающие паттерны   |
|                                  | GoF: структурные паттерны   |
|                                  | GoF: поведенческие паттерны                                       |
| Клиент-серверная разработка      | Создание REST сервисов на WebAPI.                                 |
|                                  | Создание сервиса gRPC.  |
|                                  | Организация потоковой передачи данных.                            |
|                                  | Доступ к данным в стиле CRUD.                                     |
|                                  | Паттерн DTO   |
| DevOps                           | Автоматизация сборки проектов с Github Actions.                   |
|                                  | Инструменты проверки качества проекта                             |
|                                  | Основы контейнеризации проектов.<br>Публикация проектов в Docker. |

- Что ни модуль, то отдельная дисциплина
- Много фундаментальных знаний
- Нужно показать разработку ПО как цельный процесс, а времени хватает только на учебные примеры
- Показать не только связать с ранее пройденным материалом, но и мотивировать к дальнейшему изучению каждого модуля как отдельной дисциплины



# ГЛАВНЫЕ ПРИНЦИПЫ ПРОВЕДЕНИЯ ЛЕКЦИЙ И СЕМИНАРОВ

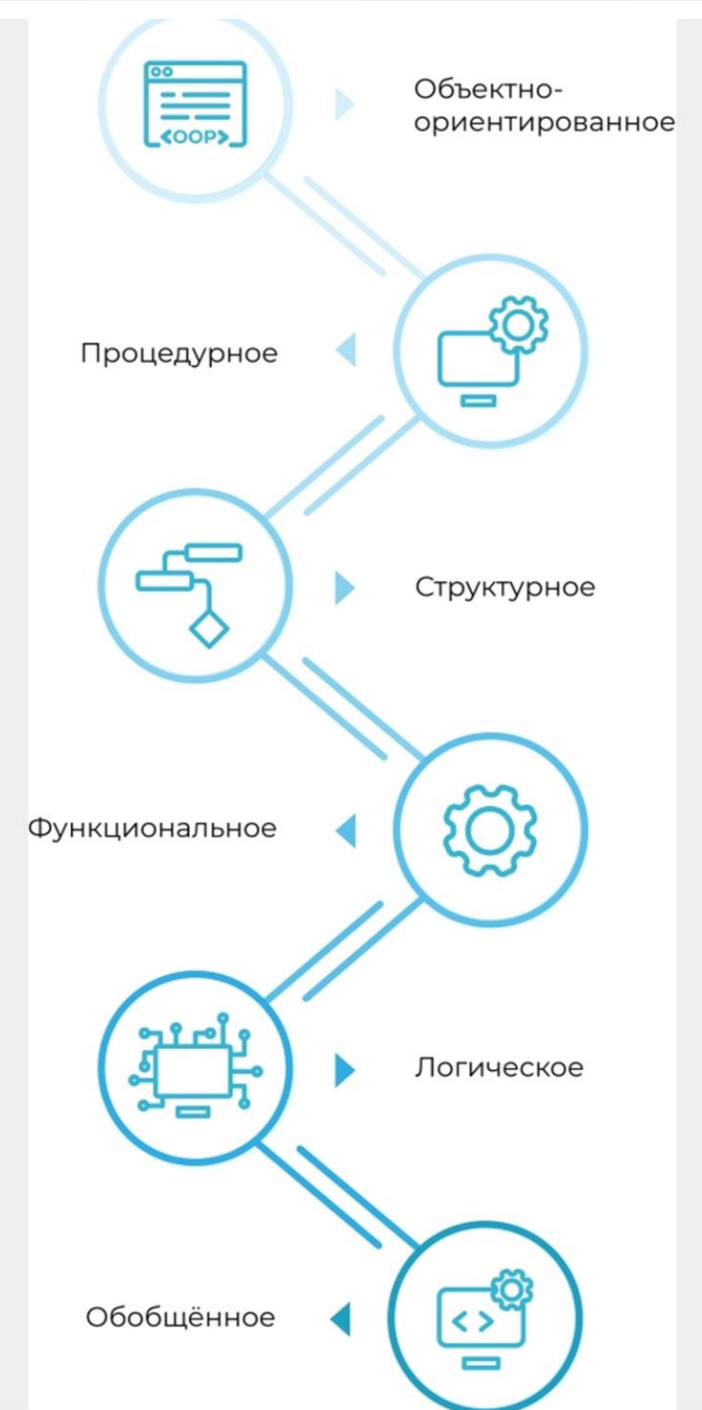
Не должно быть так, что на практических занятиях студенты получали ответы на вопросы, которые у них еще не возникли.

К этим вопросам вас нужно подвести.

$$\begin{aligned} \text{Оценка автоматом} = & ( \\ & 0.15 * \text{семинары} + \\ & 0.15 * \text{мини д. з} + \\ & 0.6 * \text{д. з.} \\ & ) / 0.9 \end{aligned}$$



# ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

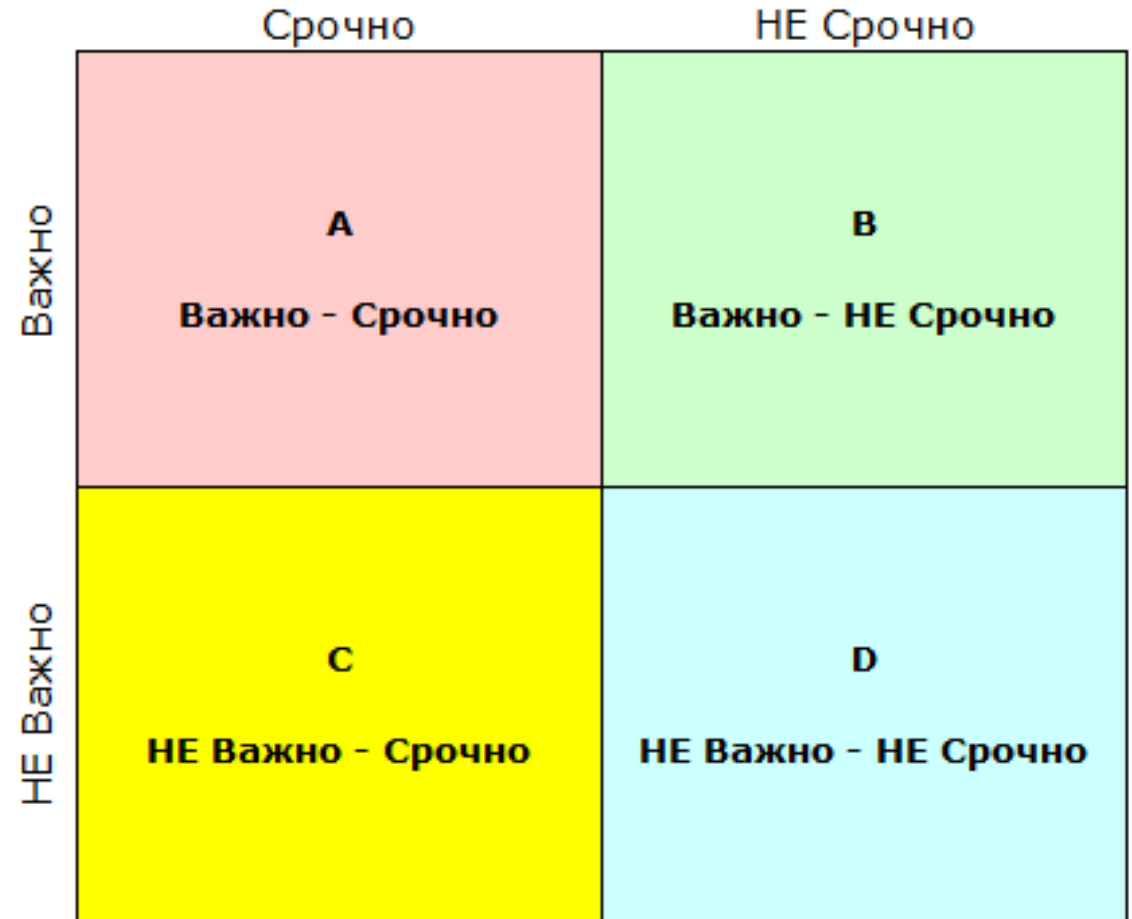


# РАССКАЗ О ДВУХ ЦЕННОСТЯХ

## Что важнее – работа ПО или его архитектура?

Цель программной архитектуры – **минимизировать человеческие ресурсы**, необходимые для построения и поддержки требуемой системы.

Если усилия поддержки ПО невелики и **остаются таковыми в течение всего жизненного цикла** - дизайн хороший.



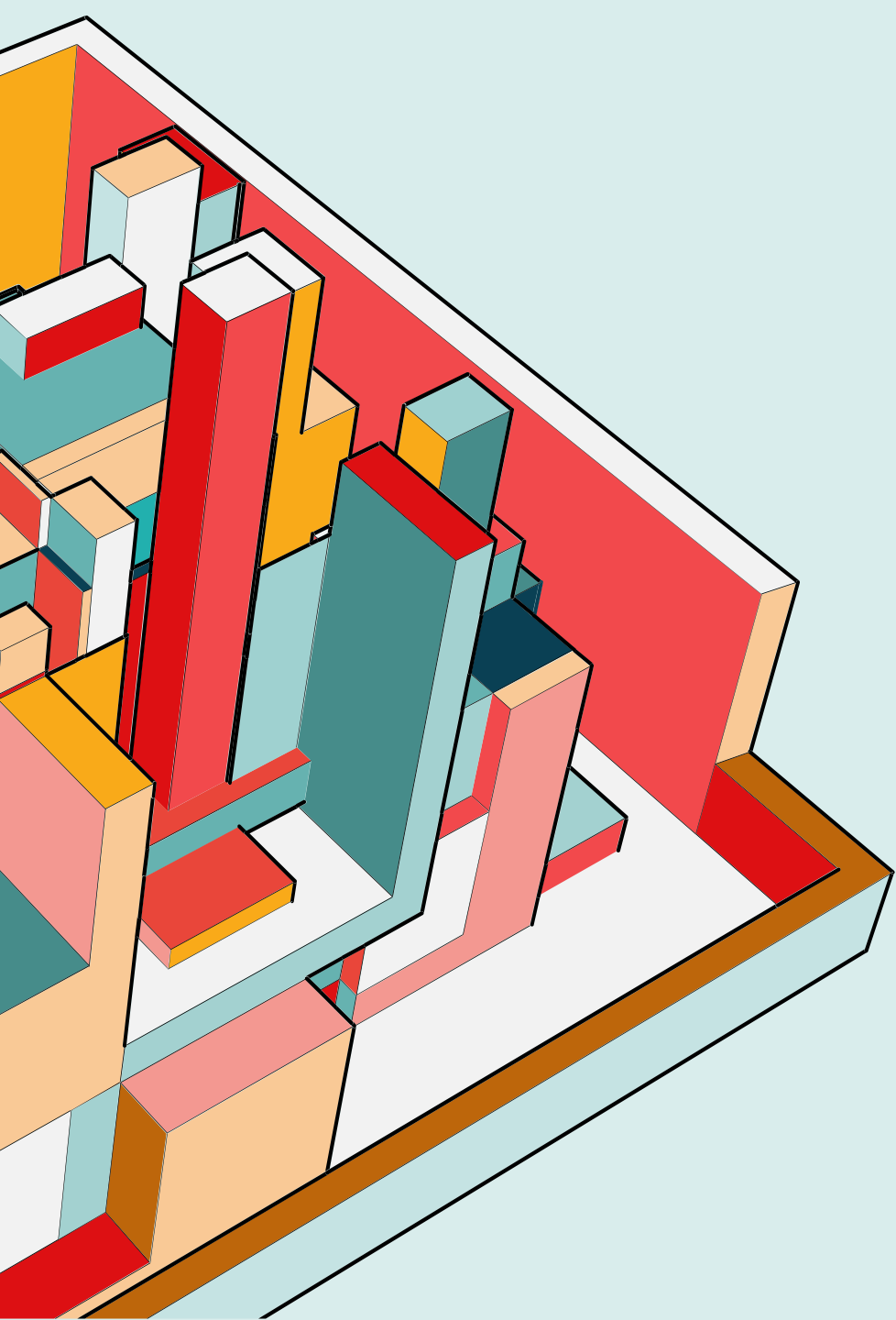
# ПАРАДИГМЫ УБИРАЮТ ВОЗМОЖНОСТИ

| Парадигма                                 | Ключевая идея   | Идеолог           | Год «открытия» |
|---|---|-------------------|----------------|
| Структурное программирование              | goto – зло, программы должны строиться из трёх базовых структур, что позволяет делать функциональную декомпозицию | Дейкстра          | 1968           |
| Объектно-ориентированное программирование | Полиморфизм – позволяет сделать модуль независимым от конкретной реализации.                                      | Алан Кэй*         | 1966           |
| Функциональное программирование           | Если переменная проинициализирована, её значение должно быть неизменным.  | Чёрч,<br>МакКарти | 1936<br>1958   |

Каждая из трёх парадигм ограничивает нас в чём-то:

- Структурное отнимает у нас goto
- ООП навязывает нам инверсию зависимостей
- ФП запрещает изменять переменные





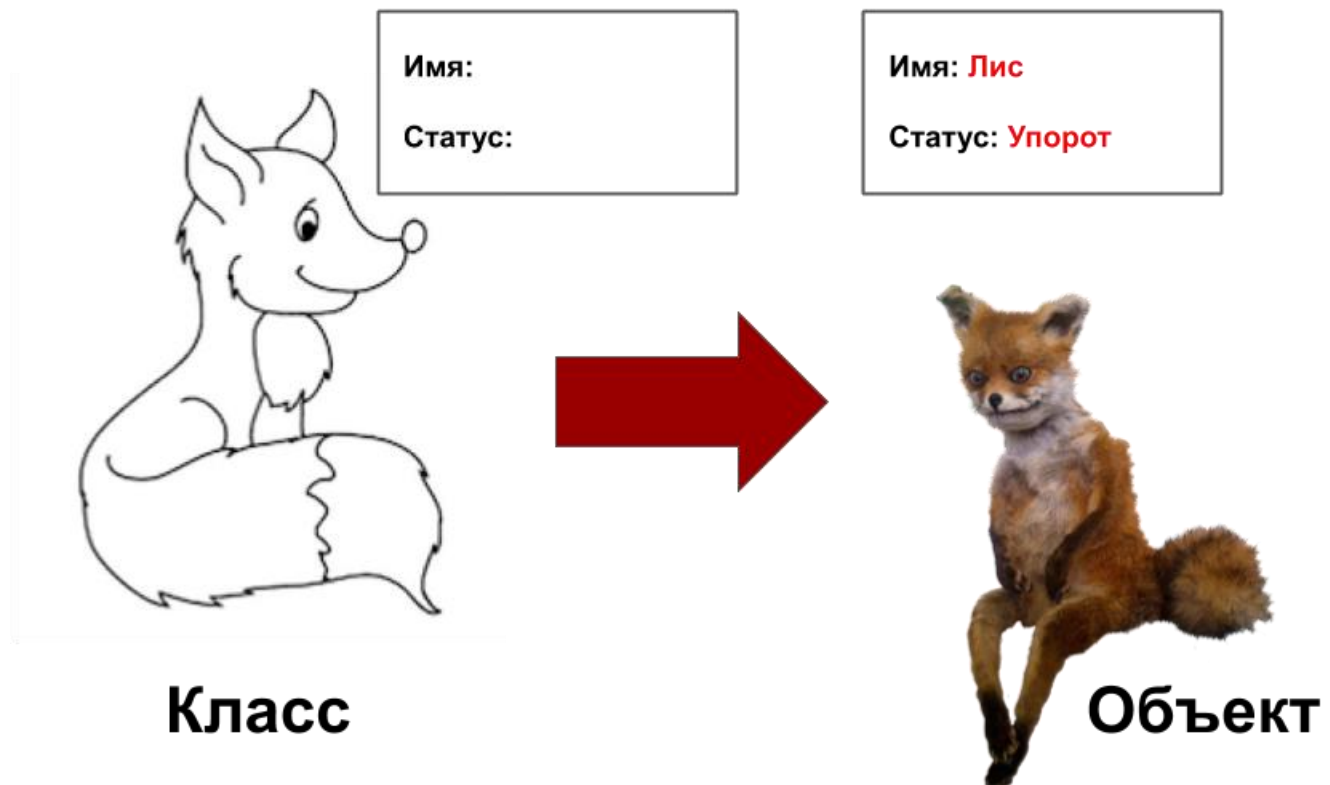
**ВСПОМНИТЬ ВСЁ  
ООП**




**Теперь я знаю**

**объектно-ориентированное  
программирование**

# КЛАСС ОПИСЫВАЕТ ТО, КАКИМ ОБЪЕКТ МОЖЕТ БЫТЬ



# **ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ И ОБЪЕКТАМИ**

A black and white portrait of actor Jason Statham. He is shown from the chest up, wearing a dark suit, white shirt, and dark tie. He has a serious expression and is looking slightly to the right of the camera. The background is dark and out of focus.

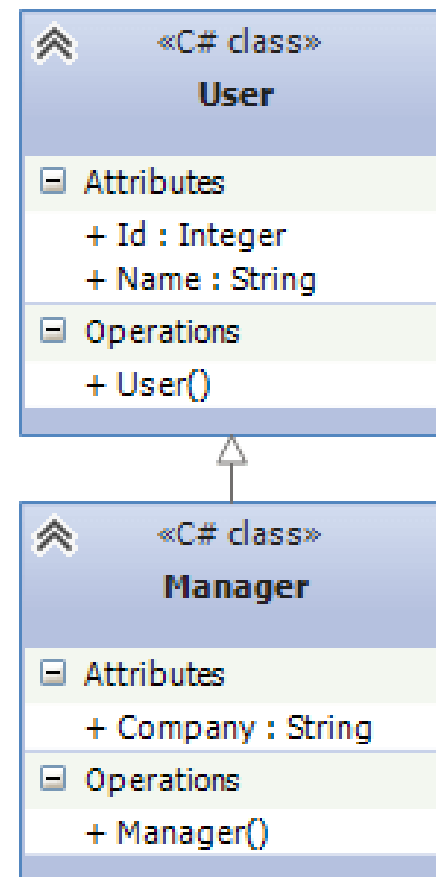
**Не важно, какие у тебя  
отношения с классом.  
Главное – отношения классов.**

**[c] Jason Statham**

# НАСЛЕДОВАНИЕ

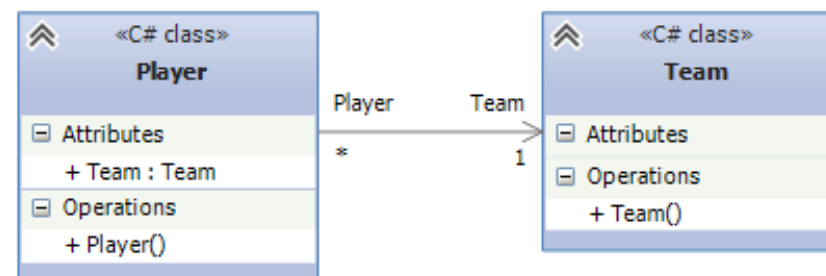
```
1 class User
2 {
3     public int Id { get; set; }
4     public string Name { get; set; }
5 }
6
7 class Manager : User
8 {
9     public string Company { get; set; }
10 }
```

Мой нарколог говорил, что первый шаг — это признать, что у тебя зависимость от наследования.



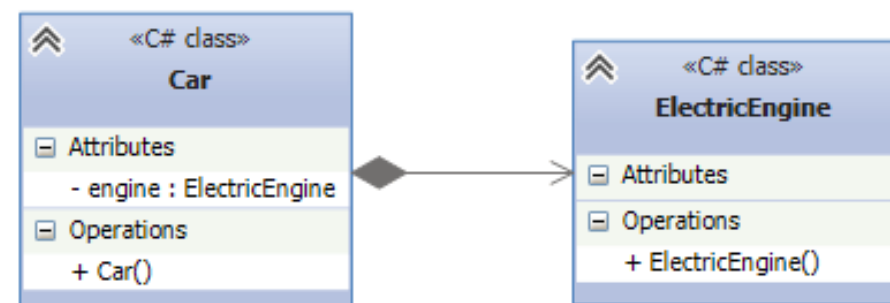
# АССОЦИАЦИЯ

```
1 class Team
2 {
3
4 }
5 class Player
6 {
7     public Team Team { get; set; }
8 }
```



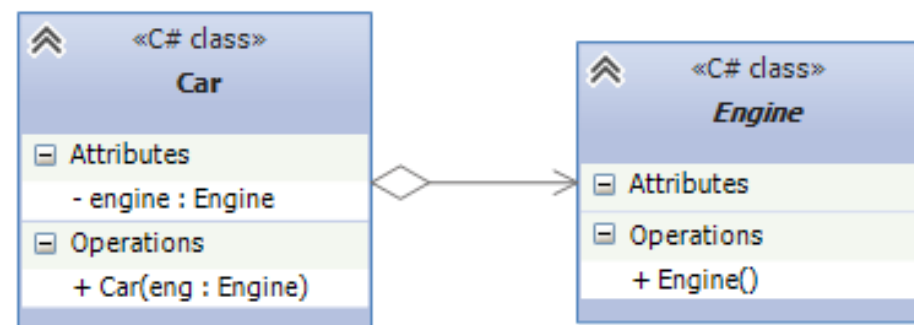
# КОМПОЗИЦИЯ

```
1 public class ElectricEngine
2 { }
3
4 public class Car
5 {
6     ElectricEngine engine;
7     public Car()
8     {
9         engine = new ElectricEngine();
10    }
11 }
```



# АГРЕГАЦИЯ

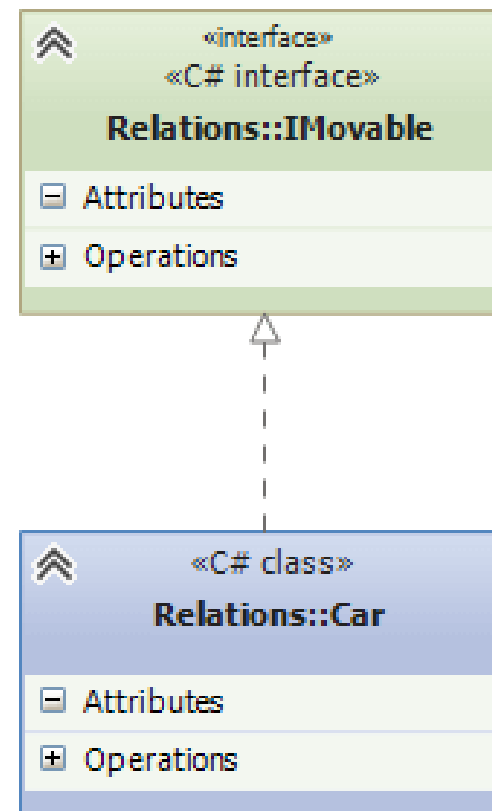
```
1 public abstract class Engine
2 { }
3
4 public class Car
5 {
6     Engine engine;
7     public Car(Engine eng)
8     {
9         engine = eng;
10    }
11 }
```






# РЕАЛИЗАЦИЯ

```
1 public interface IMovable
2 {
3     void Move();
4 }
5 public class Car : IMovable
6 {
7     public void Move()
8     {
9         Console.WriteLine("Машина едет");
10    }
11 }
```



# ИНТЕРФЕЙСЫ ИЛИ АБСТРАКТНЫЕ КЛАССЫ



Я ОРАКУЛ, Я ОТВЕЧУ НА ЛЮБОЙ ВОПРОС



Наследование или реализация



НА ЛЮБОЙ, КРОМЕ ЭТОГО

# ПРОГРАММИСТ – НЕ ОБЕЗЪЯНА

```
1 public abstract class Vehicle
2 {
3     public abstract void Move();
4 }
5
6 public class Car : Vehicle
7 {
8     public override void Move()
9     {
10         Console.WriteLine("Машина едет");
11     }
12 }
13
14 public class Bus : Vehicle
15 {
16     public override void Move()
17     {
18         Console.WriteLine("Автобус едет");
19     }
20 }
21
22 public class Tram : Vehicle
23 {
24     public override void Move()
25     {
26         Console.WriteLine("Трамвай едет");
27     }
28 }
```

```
1 public interface IMovable
2 {
3     void Move();
4 }
5
6 public abstract class Vehicle : IMovable
7 {
8     public abstract void Move();
9 }
10
11 public class Car : Vehicle
12 {
13     public override void Move() => Console.WriteLine("Машина едет");
14 }
15
16 public class Bus : Vehicle
17 {
18     public override void Move() => Console.WriteLine("Автобус едет");
19 }
20
21 public class Hourse : IMovable
22 {
23     public void Move() => Console.WriteLine("Лошадь скачет");
24 }
25
26 public class Aircraft : IMovable
27 {
28     public void Move() => Console.WriteLine("Самолет летит");
29 }
```

# КОГДА СЛЕДУЕТ ИСПОЛЬЗОВАТЬ

## Абстрактные классы:

- Определить общий функционал для родственных объектов
- Проектируем большую функциональную единицу, которая содержит много базового функционала

## Интерфейсы:

- Если нам надо определить функционал для группы разрозненных объектов, которые могут быть никак не связаны между собой.
- Если мы проектируем небольшой функциональный тип

Если классы относятся к единой системе классификации, то выбирается абстрактный класс. Иначе выбирается интерфейс

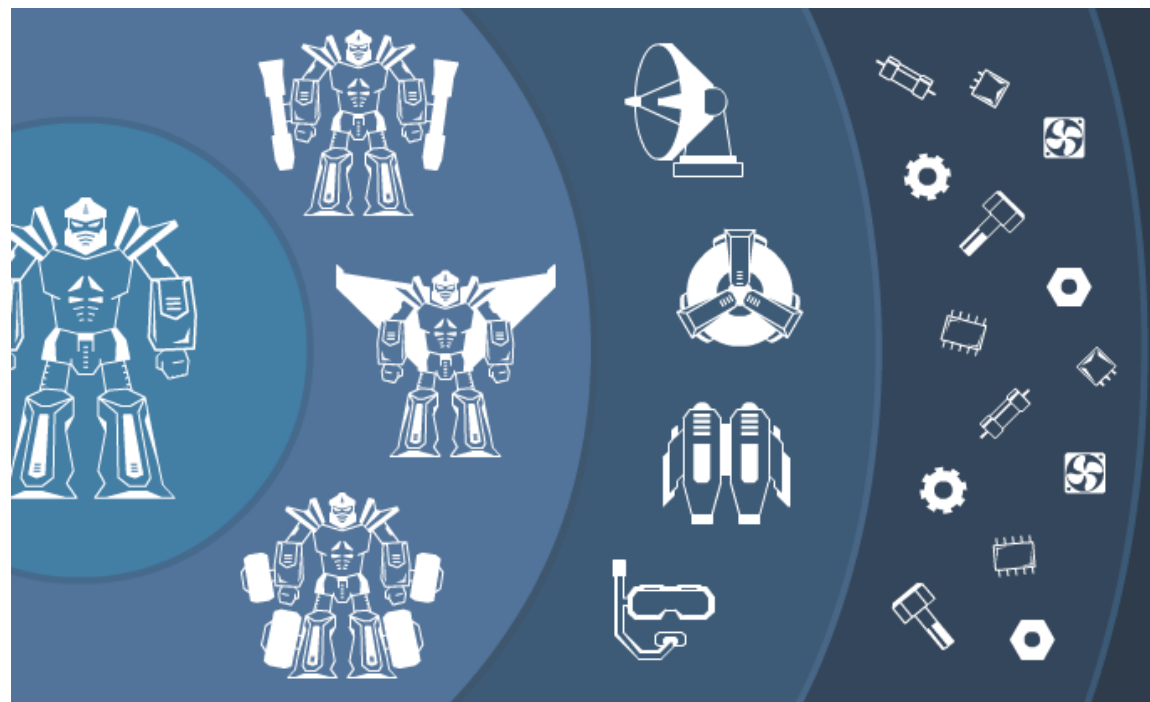
# УРОВНИ АБСТРАКЦИИ

**Первый слой.** В игре есть один класс трансформера, все свойства и поведение описаны в нем.

**Второй уровень.** В игре есть базовый трансформер с основными способностями и классы трансформеров со своей специализацией. Тем самым игроку предоставляется возможность выбора, а разработчикам упрощается добавление новых классов.

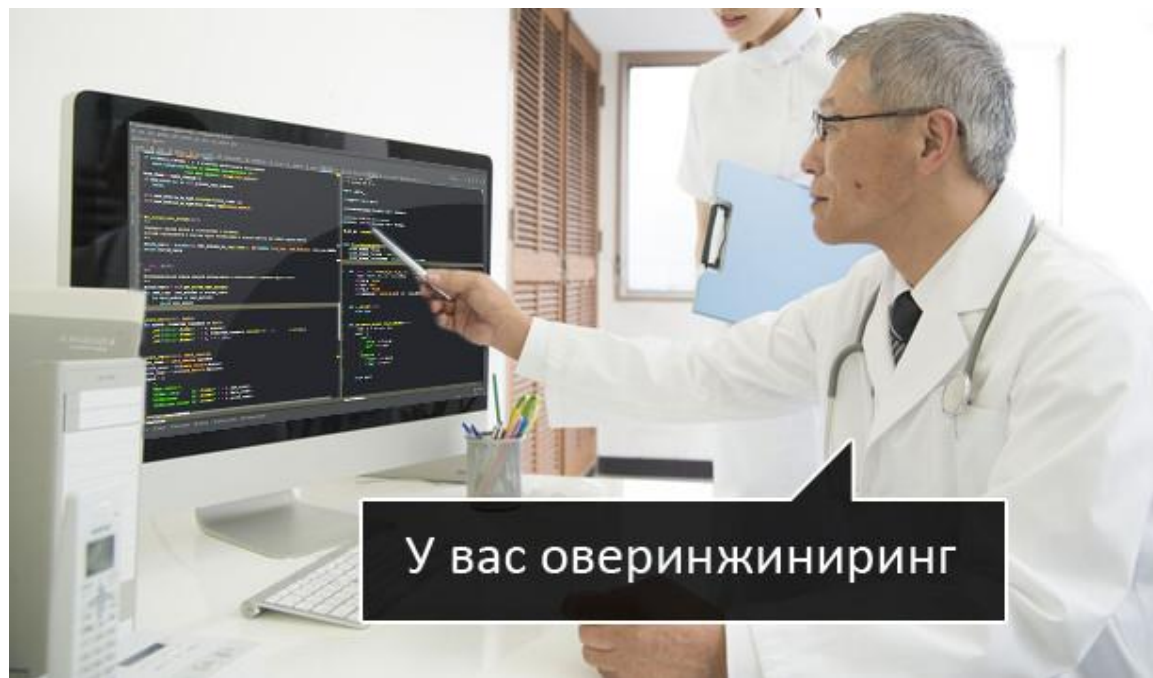
**Третий уровень.** Помимо классификации трансформеров вводится агрегация с помощью системы слотов и компонентов. Теперь часть поведения будет определяться тем, какой стаф игрок установил в своего трансформера.

**Четвертый уровень.** В компоненты можно тоже включить собственную агрегацию, предоставляющую возможность выбора материалов и деталей, из которого собираются эти компоненты.

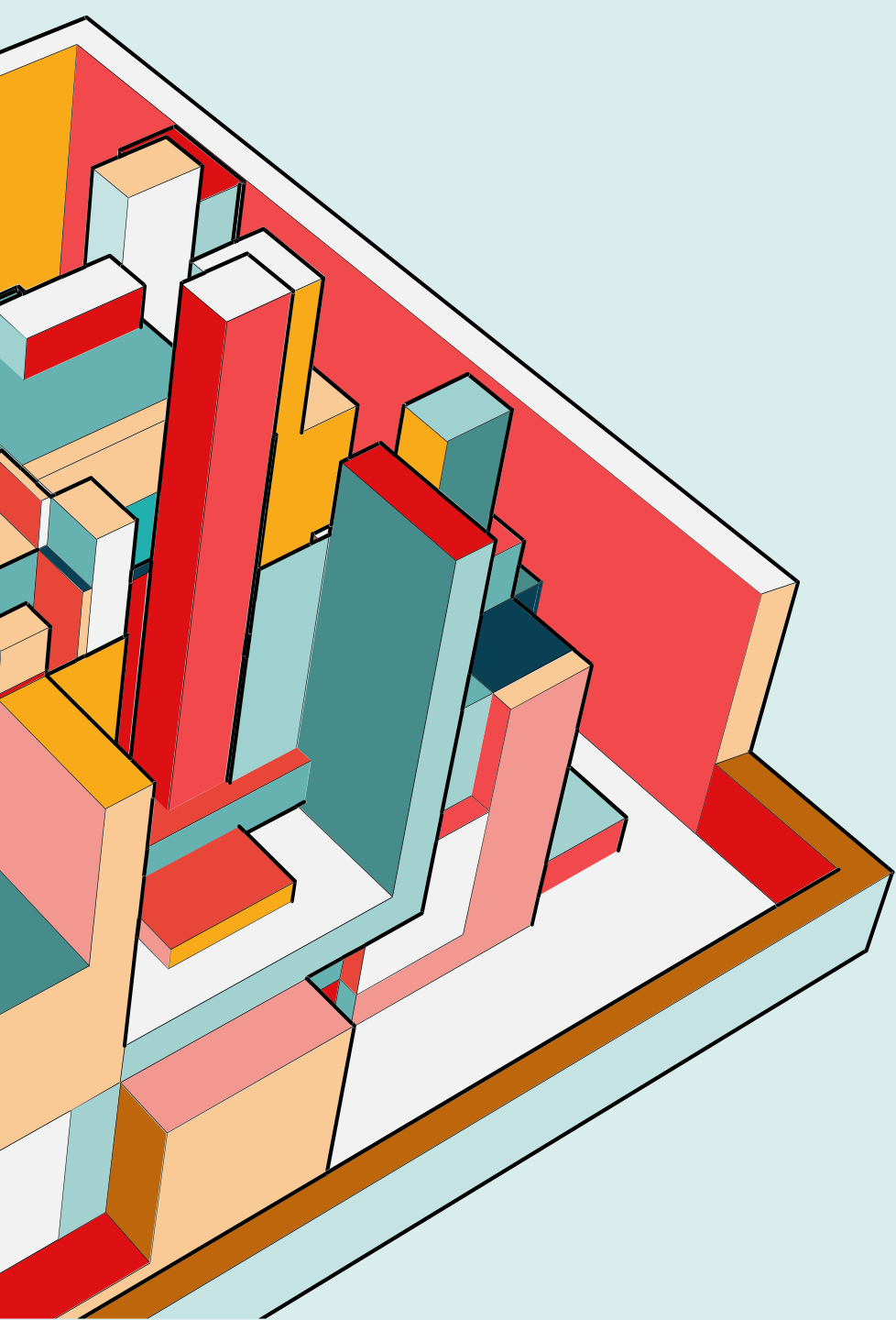


# ВЫБОР УРОВНЯ АБСТРАКЦИИ

1. Абстракции недостаточно. Расширения проекта будут упираться в архитектурные ограничения.
2. Уровень абстракции слишком высок. Оверинжиниринг в виде сложной архитектуры, которую трудно поддерживать, и излишней гибкости, которая никогда в этом проекте не пригодится.



Любую архитектурную проблему можно решить добавлением дополнительного слоя абстракции, кроме проблемы большого количества абстракций.



# ПРИНЦИПЫ ДИЗАЙНА

SOLID

GRASP

KISS, DRY, YAGNI, BDUF,  
APO и бритва Оккама





| Время | Принципы | Имена         |
|-------|----------|---------------|
| 1997  | GRASP    | Крэг Ларман   |
| 2002  | SOLID    | Роберт Мартин |

**Принцип проектирования** –  
методологическое правило, которое выражает  
общий взгляд на разработку ПО



Lisp, Smalltalk

Принципы GRASP

Соавтор Scrum

Чистый код

Чистая архитектура

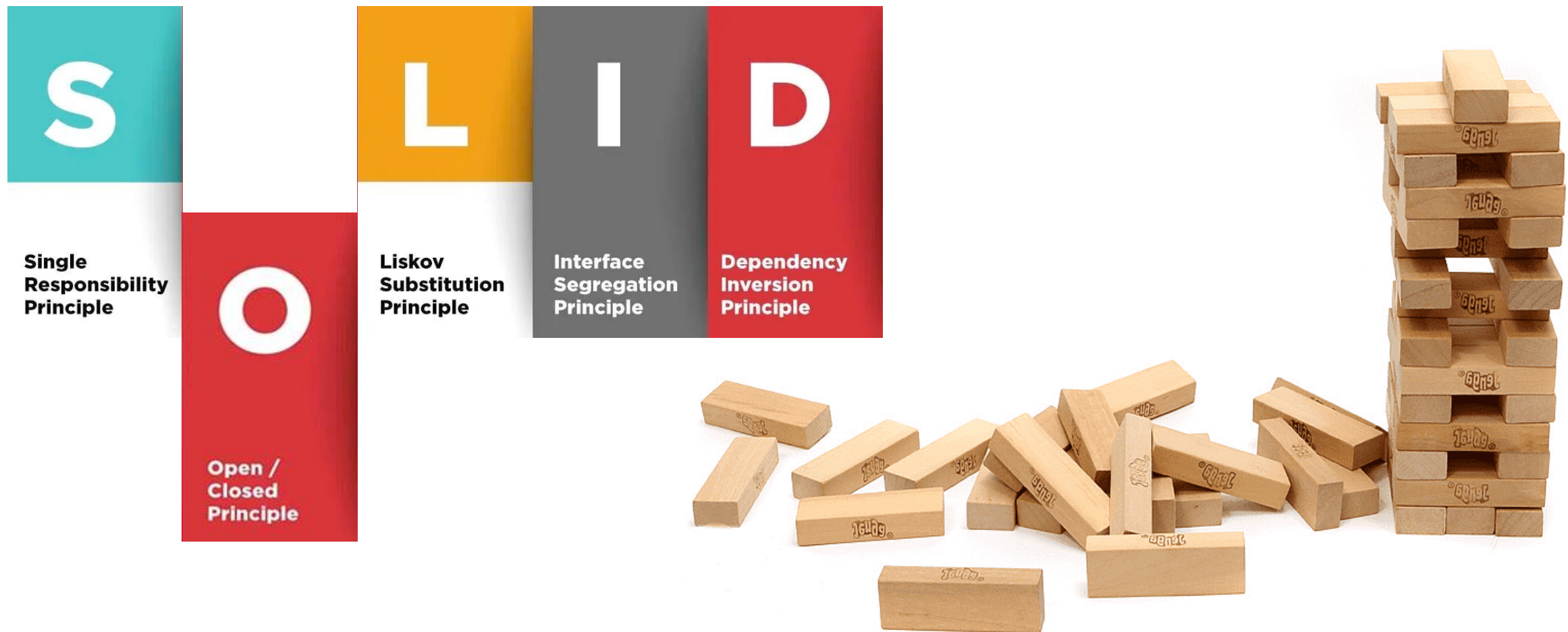
Гибкая разработка



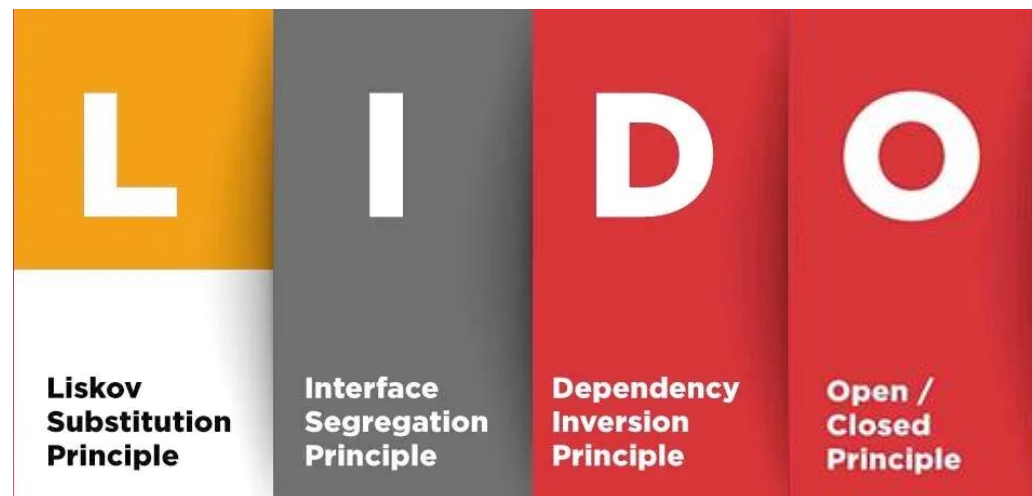
# ПРОГРАММИРОВАНИЕ



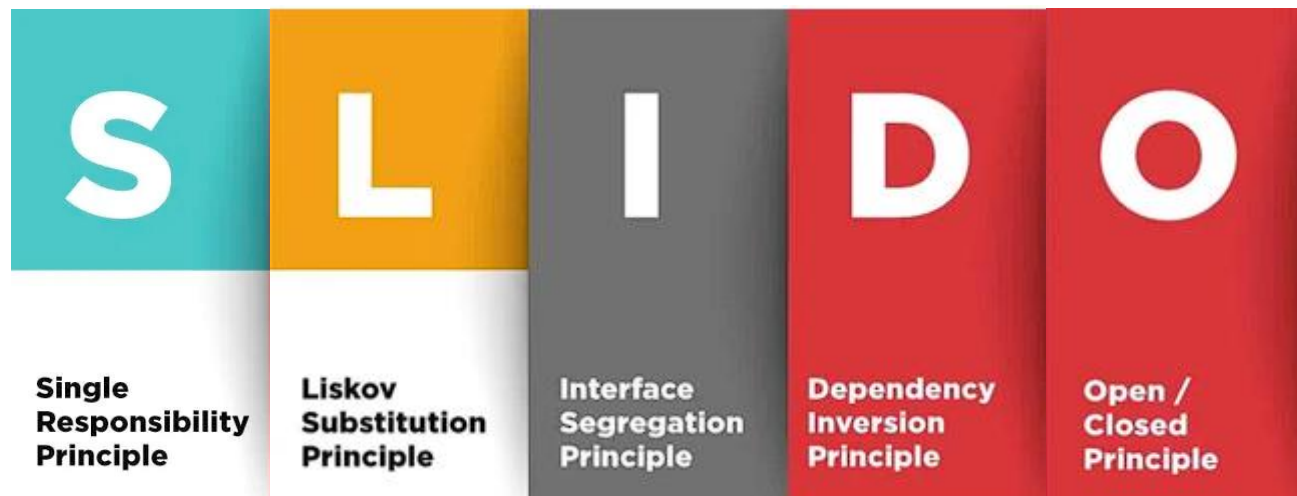
# ПРОГРАММИРОВАНИЕ — ЭТО



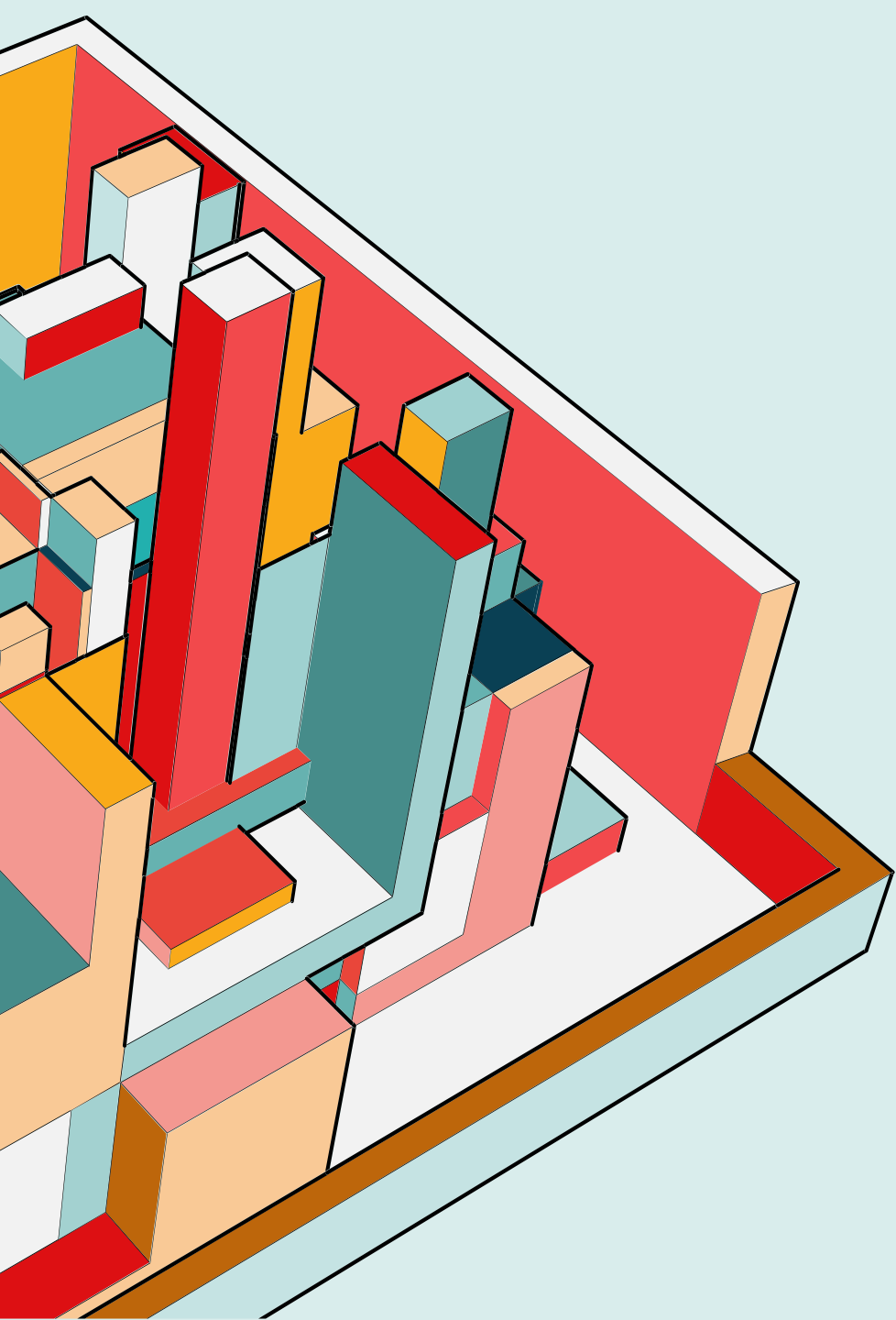
# ПРОГРАММИРОВАНИЕ — ЭТО НЕ



# ПРОГРАММИРОВАНИЕ — ЭТО НЕ ДЖЕНГА



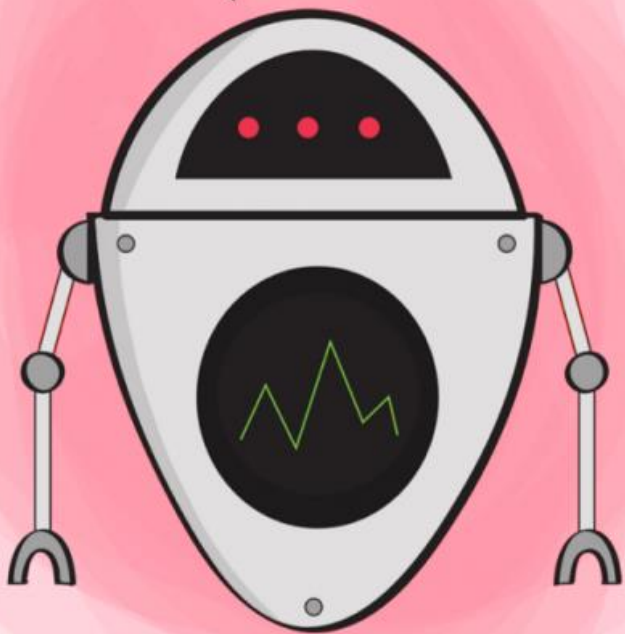




# **SINGLE- RESPONSIBILITY PRINCIPLE**

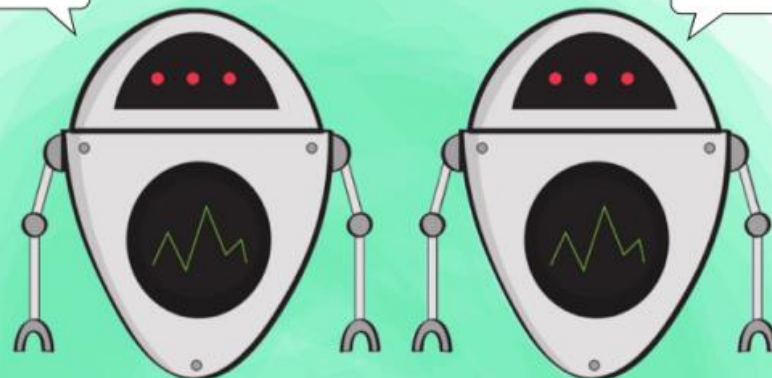
Класс должен иметь одну и только одну причину измениться

Я повар,  
садовник,  
маляр  
и водитель.



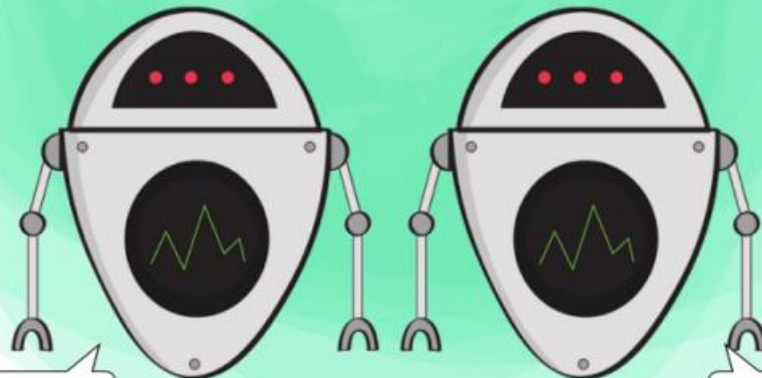
Я повар.

Я садовник.



Я маляр.

Я водитель.



```
class Report
{
    public string Text { get; set; }

    public void GoToFirstPage()
    {
        Console.WriteLine("Go to the first page");
    }

    public void GoToLastPage()
    {
        Console.WriteLine("Go to the last page");
    }

    public void GoToPage(int pageNumber)
    {
        Console.WriteLine("Go to page {0}", pageNumber);
    }

    public void Print()
    {
        Console.WriteLine("Print report");
        Console.WriteLine(Text);
    }
}
```

```
class Report
```

```
{
```

```
    public string Text { get; set; }
```

```
    public void GoToFirstPage()...
```

```
    public void GoToLastPage()...
```

```
    public void GoToPage(int pageNumber)...
```

```
    public void Print()...
```

```
    public void PrintToPDF()...
```

```
    public void PrintToPrinter()...
```

```
}
```



```
class Report
```

```
{
```

```
    public string Text { get; set; }
```

```
    public void GoToFirstPage()...
```

```
    public void GoToLastPage()...
```

```
    public void GoToPage(int pageNumber)...
```

```
    public void Print()...
```

```
    public void PrintToPDF()...
```

```
    public void PrintToPrinter()...
```

```
}
```

```
class Report
{
    public string Text { get; set; }

    public void GoToFirstPage()...

    public void GoToLastPage()...

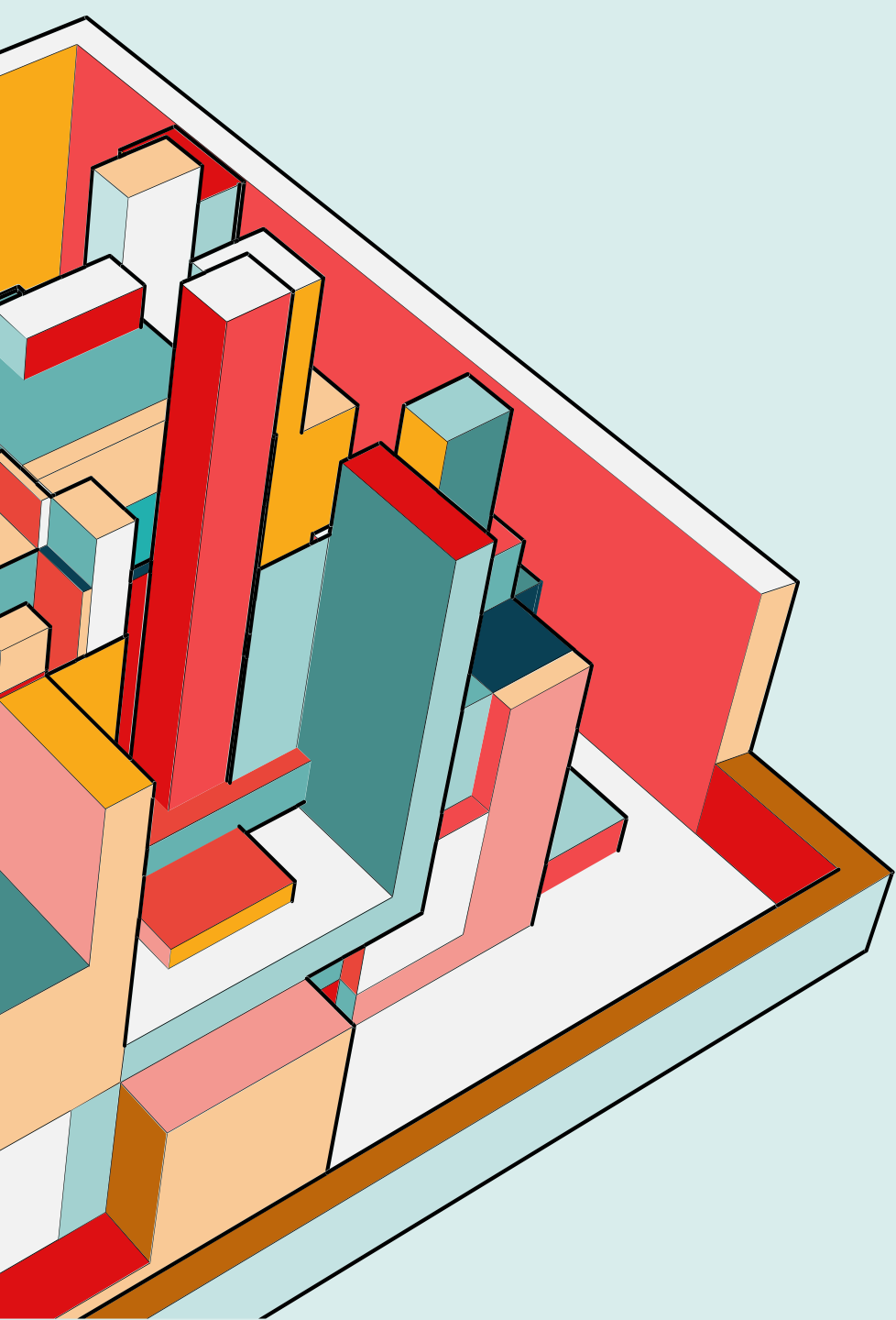
    public void GoToPage(int pageNumber)...
```

```
class Printer
{
    public void Print(Report report)...
```

public void PrintToPDF(Report report)...

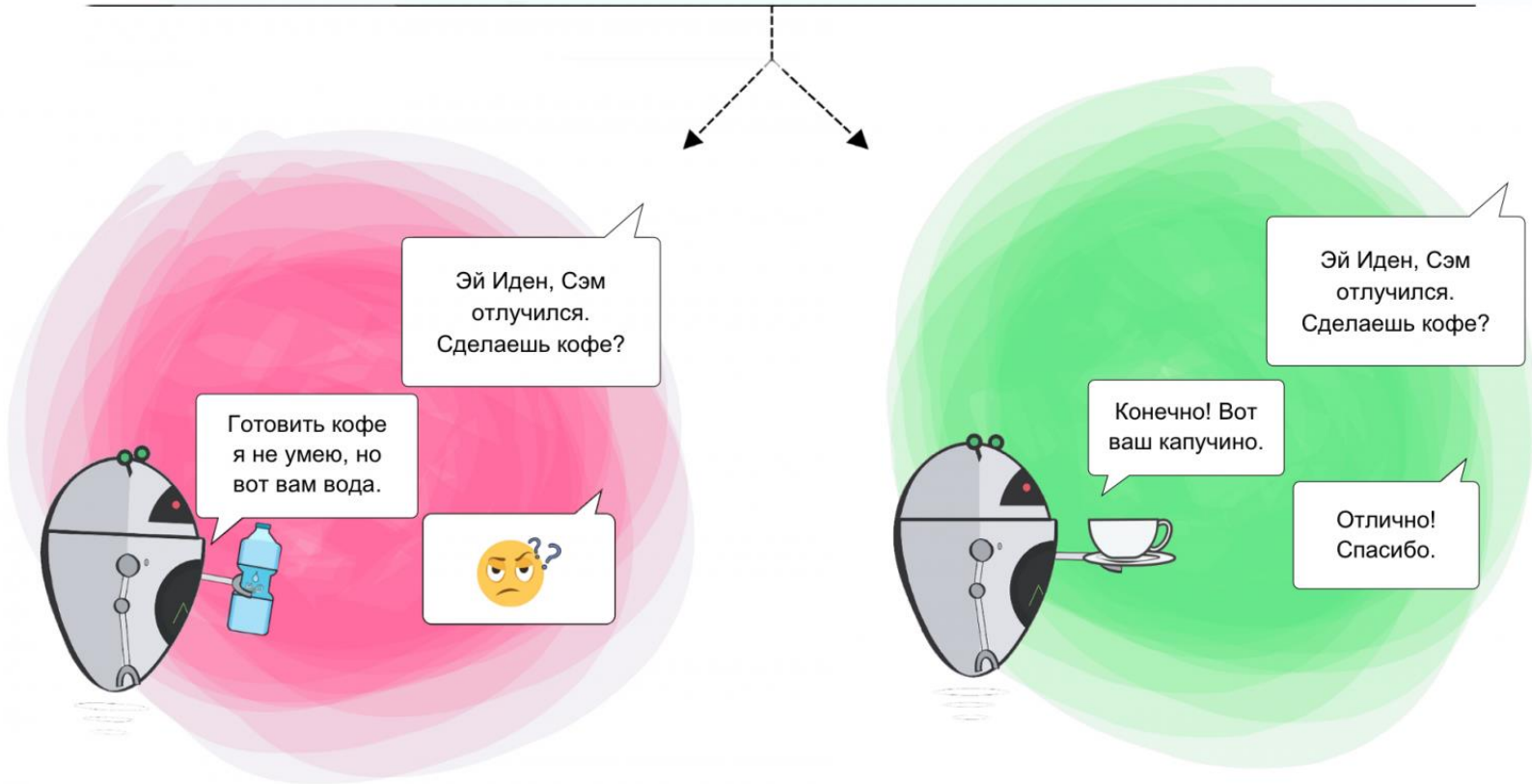
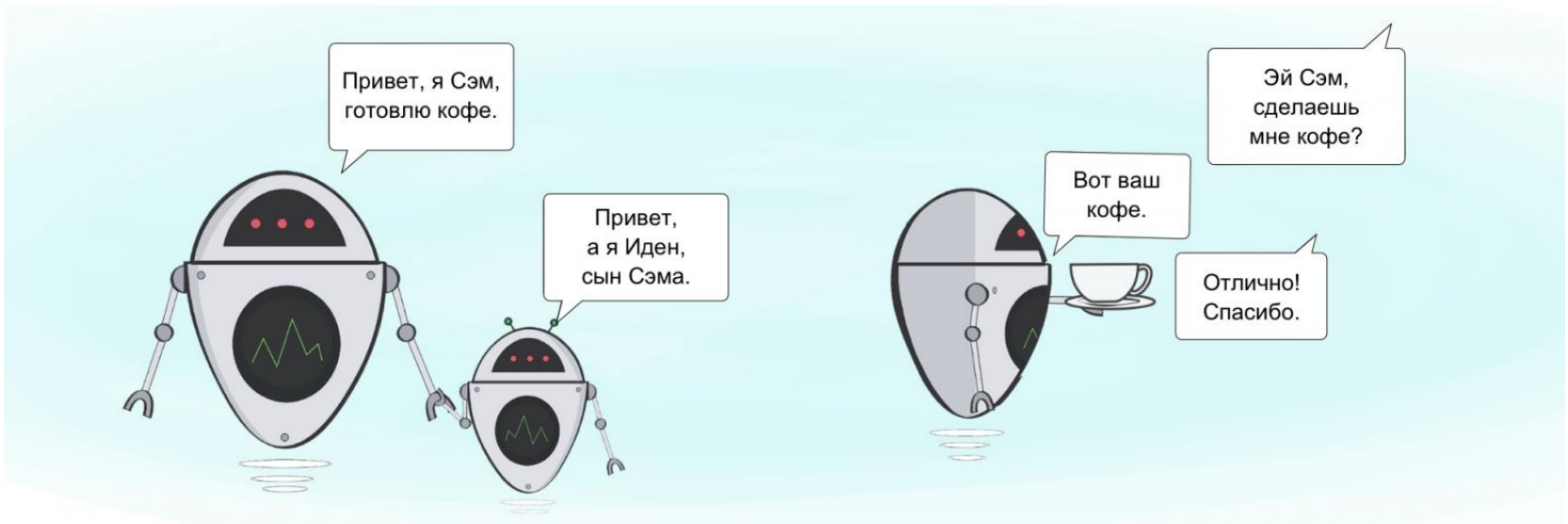
public void PrintToPrinter(Report report)...

```
}
```



# LSKOV SUBSTITUTION PRINCIPLE

Вы должны иметь возможность использовать производный класс вместо родительского класса и вести себя с ним таким же образом без внесения изменений



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

    public int GetArea()
    {
        return Width * Height;
    }
}
```

```
class Square : Rectangle
{
    public override int Width
    {
        get
        {
            return base.Width;
        }

        set
        {
            base.Width = value;
            base.Height = value;
        }
    }

    public override int Height...
```

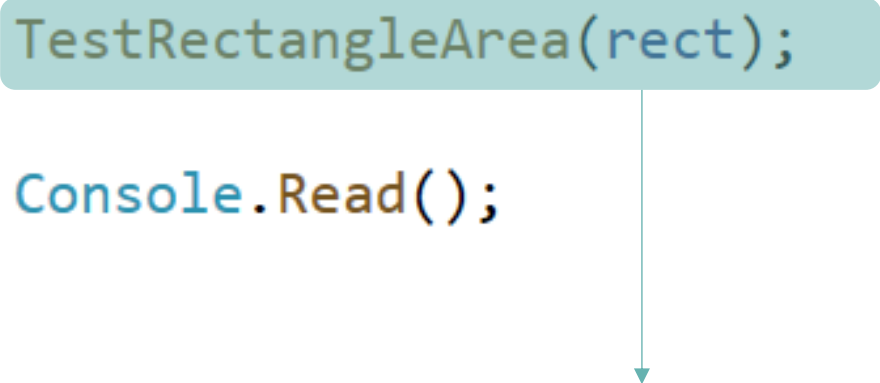
```
class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }

    public int GetArea()
    {
        return Width * Height;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Rectangle rect = new Square();
        TestRectangleArea(rect);

        Console.Read();
    }

    public static void TestRectangleArea(Rectangle rect)
    {
        rect.Height = 5;
        rect.Width = 10;
        if (rect.GetArea() != 50)
            throw new Exception("Incorrect area!");
    }
}
```

A light blue rectangular box highlights the line `TestRectangleArea(rect);` in the `Main` method. A vertical arrow points from the bottom of this box down to the parameter `Rectangle rect` in the `TestRectangleArea` method signature.

```
static void Main(string[] args)
{
    Rectangle rect = new Square();
    TestRectangleArea(rect);

    Console.Read();
}

public static void TestRectangleArea(Rectangle rect)
{
    if (rect is Square)
    {
        rect.Height = 5;
        if (rect.GetArea() != 25)
            throw new Exception("Incorrect area!");
    }
    else if (rect is Rectangle)
    {
        rect.Height = 5;
        rect.Width = 10;
        if (rect.GetArea() != 50)
            throw new Exception("Incorrect area!");
    }
}
```

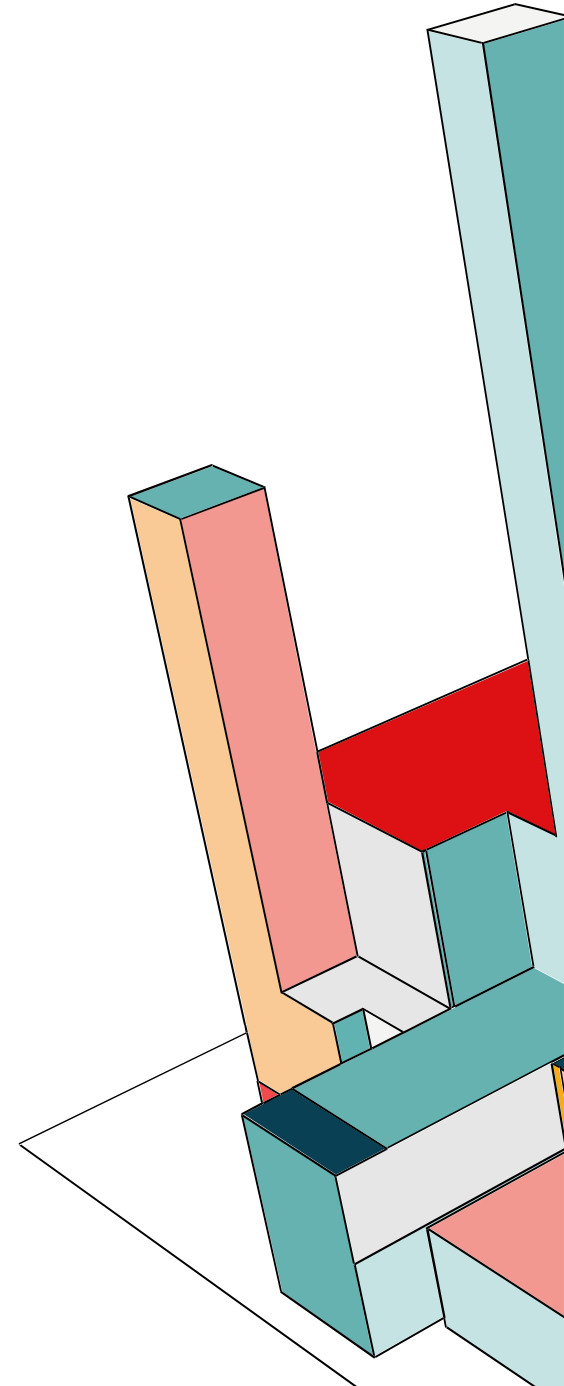


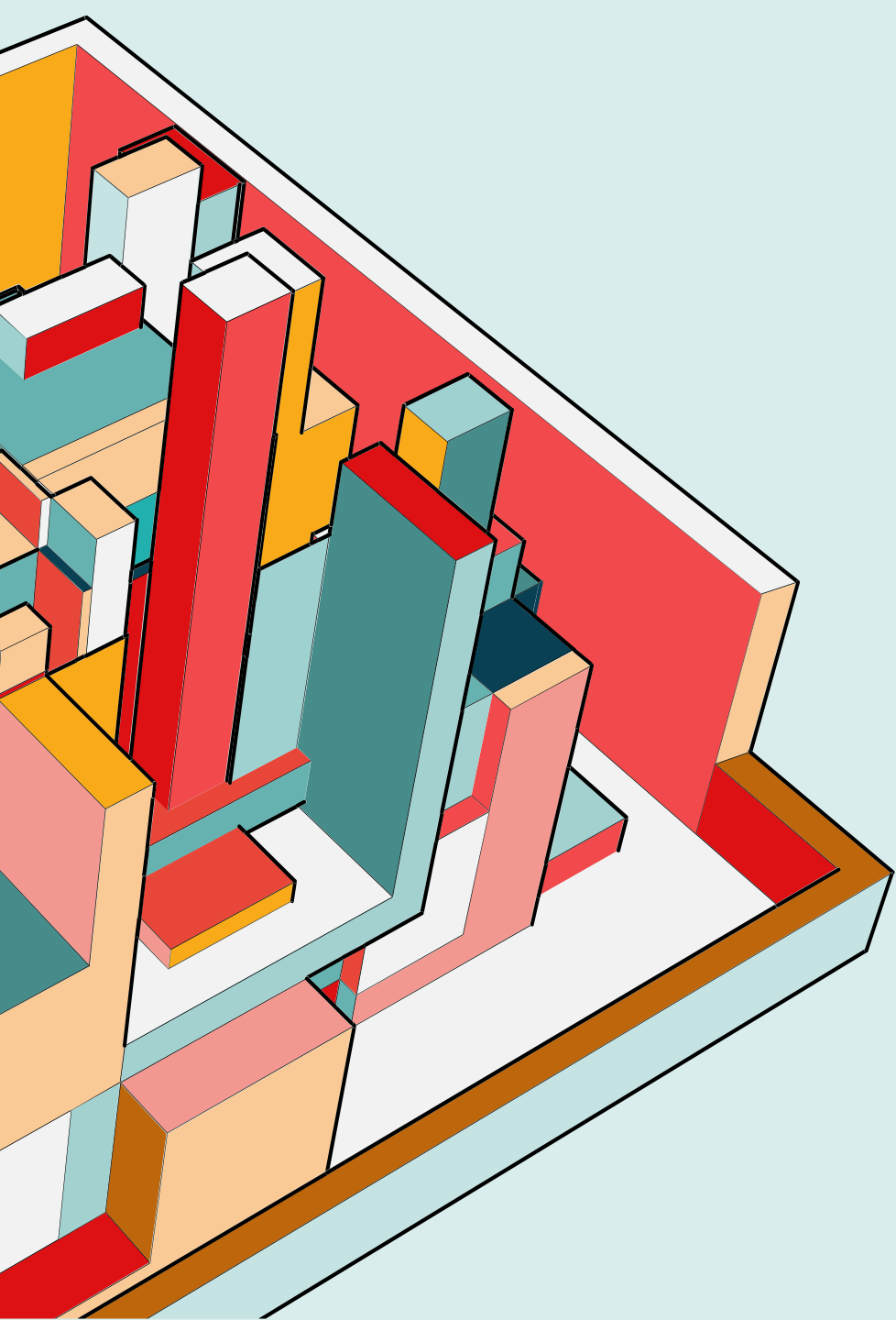
# ОШИБКИ В НАСЛЕДОВАНИИ

Невозможно установить правильность модели,  
рассматриваемой изолированно

Как предусмотреть требования клиентов?

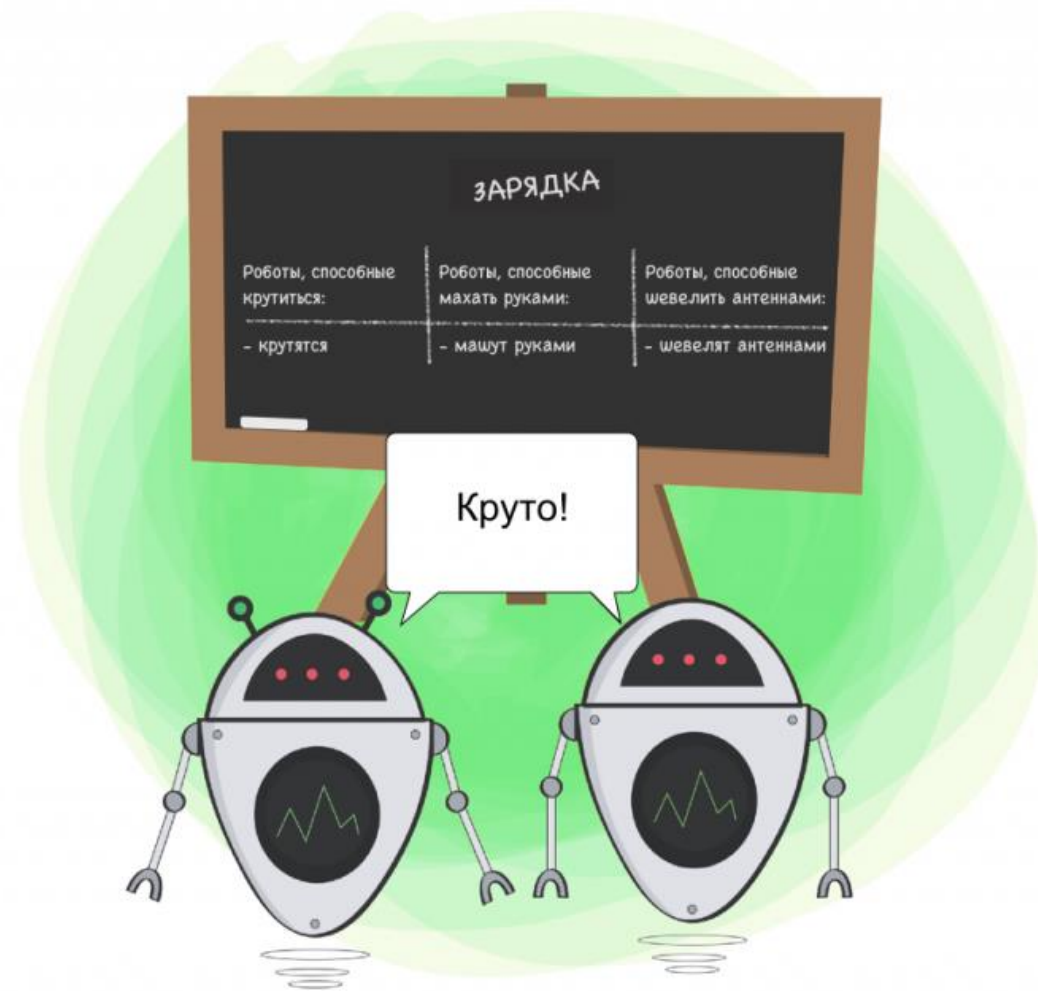
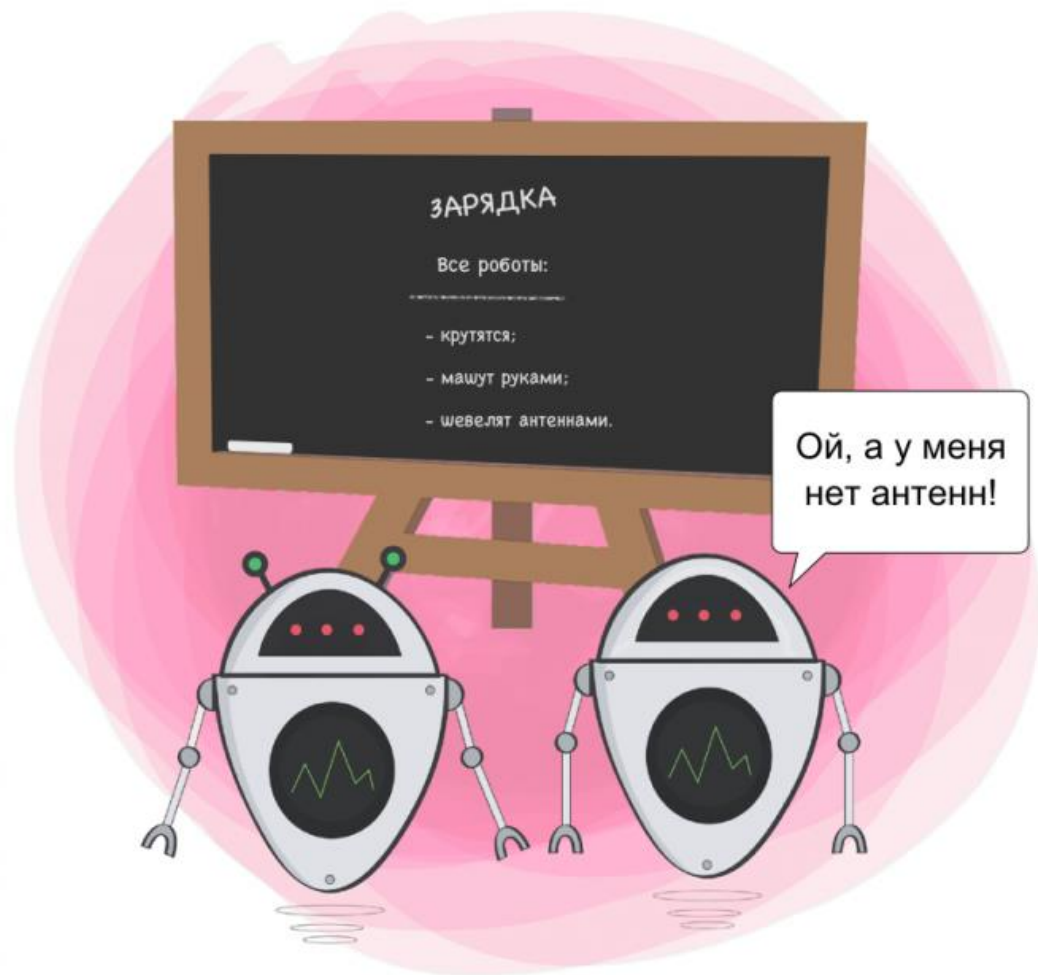
- Экстрасенсорные способности Опыт
- Программирование через тестирование
- Программирование по контракту





# INTERFACE SEGREGATION PRINCIPLE

Слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе



```
interface IMessage
{
    void Send();
    string Text { get; set; }
    string Subject { get; set; }
    string ToAddress { get; set; }
    string FromAddress { get; set; }
}

class EmailMessage : IMessage
{
    public string Subject { get; set; }
    public string Text { get; set; }
    public string FromAddress { get; set; }
    public string ToAddress { get; set; }

    public void Send()
    {
        Console.WriteLine("Send Email: {0}", Text);
    }
}
```

```
class TextMessage : IMessage
{
    public string Text { get; set; }
    public string FromAddress { get; set; }
    public string ToAddress { get; set; }

    public string Subject
    {
        get
        {
            throw new NotImplementedException();
        }

        set
        {
            throw new NotImplementedException();
        }
    }

    public void Send()
    {
        Console.WriteLine("Send Sms: {0}", Text);
    }
}
```

```
interface IMessage
```

```
{  
    void Send();  
    string Text { get; set; }  
    string Subject { get; set; }  
    string ToAddress { get; set; }  
    string FromAddress { get; set; }  
}
```



```
interface IMessage
```

```
{  
    void Send();  
    string Text { get; set; }  
    string ToAddress { get; set; }  
    string Subject { get; set; }  
    string FromAddress { get; set; }  
    byte[] Voice { get; set; }  
}
```

```
class VoiceMessage : IMessage
{
    public string ToAddress { get; set; }
    public string FromAddress { get; set; }
    public byte[] Voice { get; set; }

    public string Text { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
    public string Subject { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }

    public void Send() { Console.WriteLine("Send voice"); }
}
```

```
class EmailMessage : IMessage
{
    public string Subject { get; set; }
    public string Text { get; set; }
    public string FromAddress { get; set; }
    public string ToAddress { get; set; }

    public byte[] Voice { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }

    public void Send() { Console.WriteLine("Send Email: {0}", Text); }
}
```

```
interface IMessage
{
    void Send();
    string Text { get; set; }
    string ToAddress { get; set; }
    string Subject { get; set; }
    string FromAddress { get; set; }

    byte[] Voice { get; set; }
}
```



```
interface IMessage
{
    void Send();
    string ToAddress { get; set; }
    string FromAddress { get; set; }
}

interface IVoiceMessage : IMessage
{
    byte[] Voice { get; set; }
}

interface ITextMessage : IMessage
{
    string Text { get; set; }
}

interface IEmailMessage : ITextMessage
{
    string Subject { get; set; }
}
```



```
class VoiceMessage : IVoiceMessage
{
    public string ToAddress { get; set; }
    public string FromAddress { get; set; }

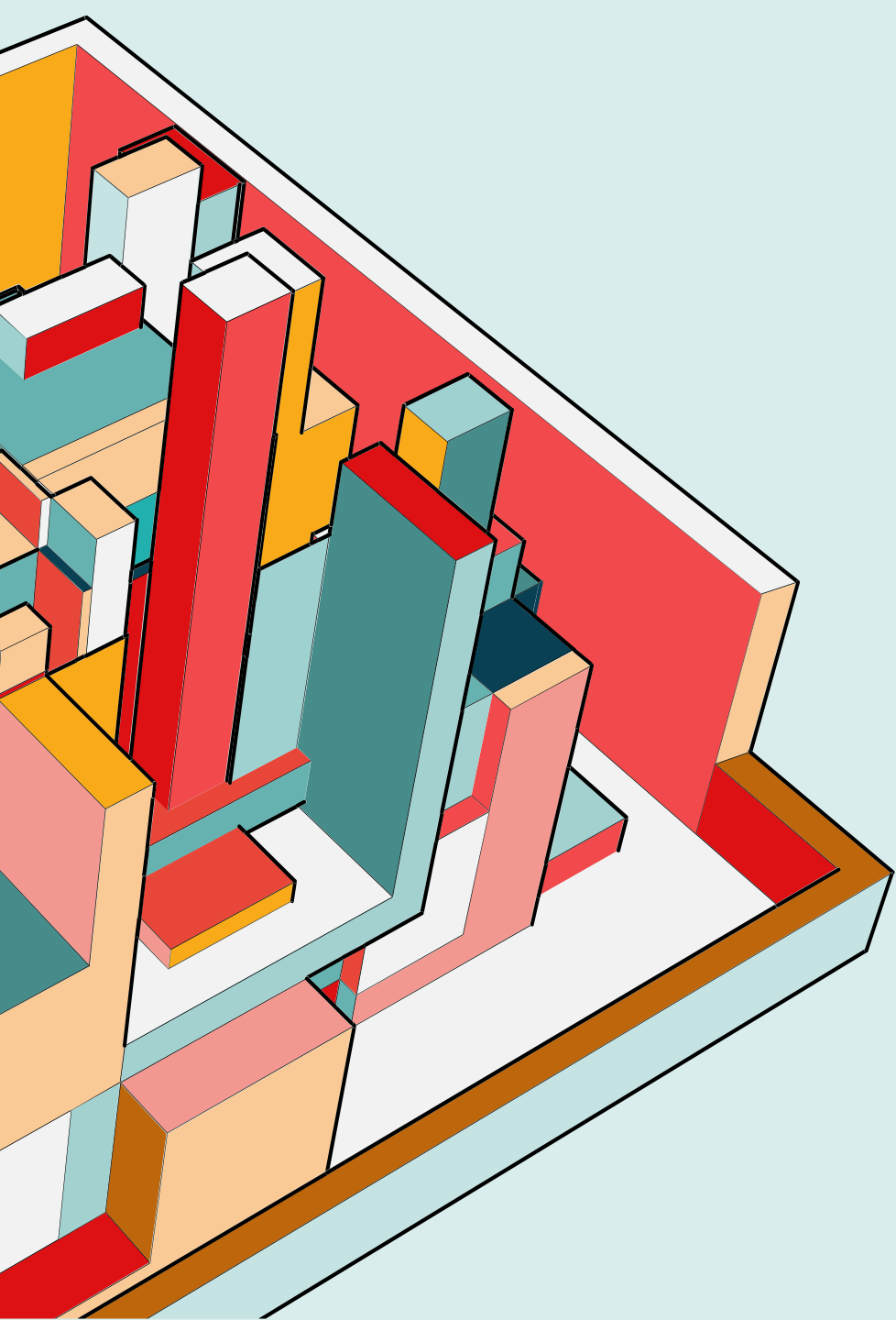
    public byte[] Voice { get; set; }
    public void Send()
}

class EmailMessage : IEmailMessage
{
    public string Text { get; set; }
    public string Subject { get; set; }
    public string FromAddress { get; set; }
    public string ToAddress { get; set; }

    public void Send()
}

class TextMessage : ITextMessage
{
    public string Text { get; set; }
    public string FromAddress { get; set; }
    public string ToAddress { get; set; }

    public void Send()
}
```

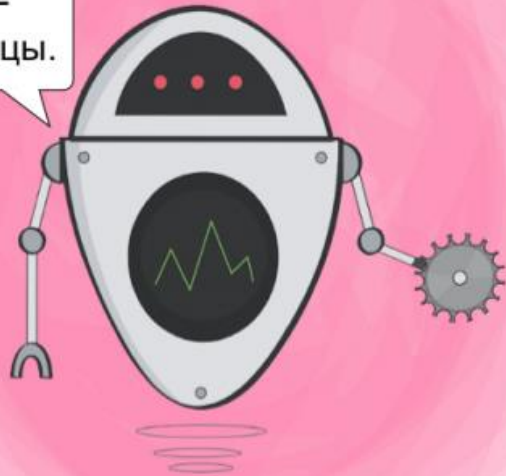


# DEPENDENCY INVERSION PRINCIPLE

Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Я режу пиццу  
своей рукой-  
ножом для пиццы.



Я режу пиццу  
любым доступным  
инструментом.



```
class Book
{
    public string Text { get; set; }
    public ConsolePrinter Printer { get; set; }

    public void Print()
    {
        Printer.Print(Text);
    }
}
```

```
class ConsolePrinter
{
    public void Print(string text)
    {
        Console.WriteLine(text);
    }
}
```

```
class Book
{
    public string Text { get; set; }
    public HtmlPrinter Printer { get; set; }

    public void Print()
    {
        Printer.Print(Text);
    }
}
```

```
class ConsolePrinter
{
    public void Print(string text)...
```

```
class HtmlPrinter
{
    public void Print(string text)...
```

```
interface IPrinter
{
    void Print(string text);
}

class Book
{
    public string Text { get; set; }
    public IPrinter Printer { get; set; }

    public Book(IPrinter printer) { this.Printer = printer; }

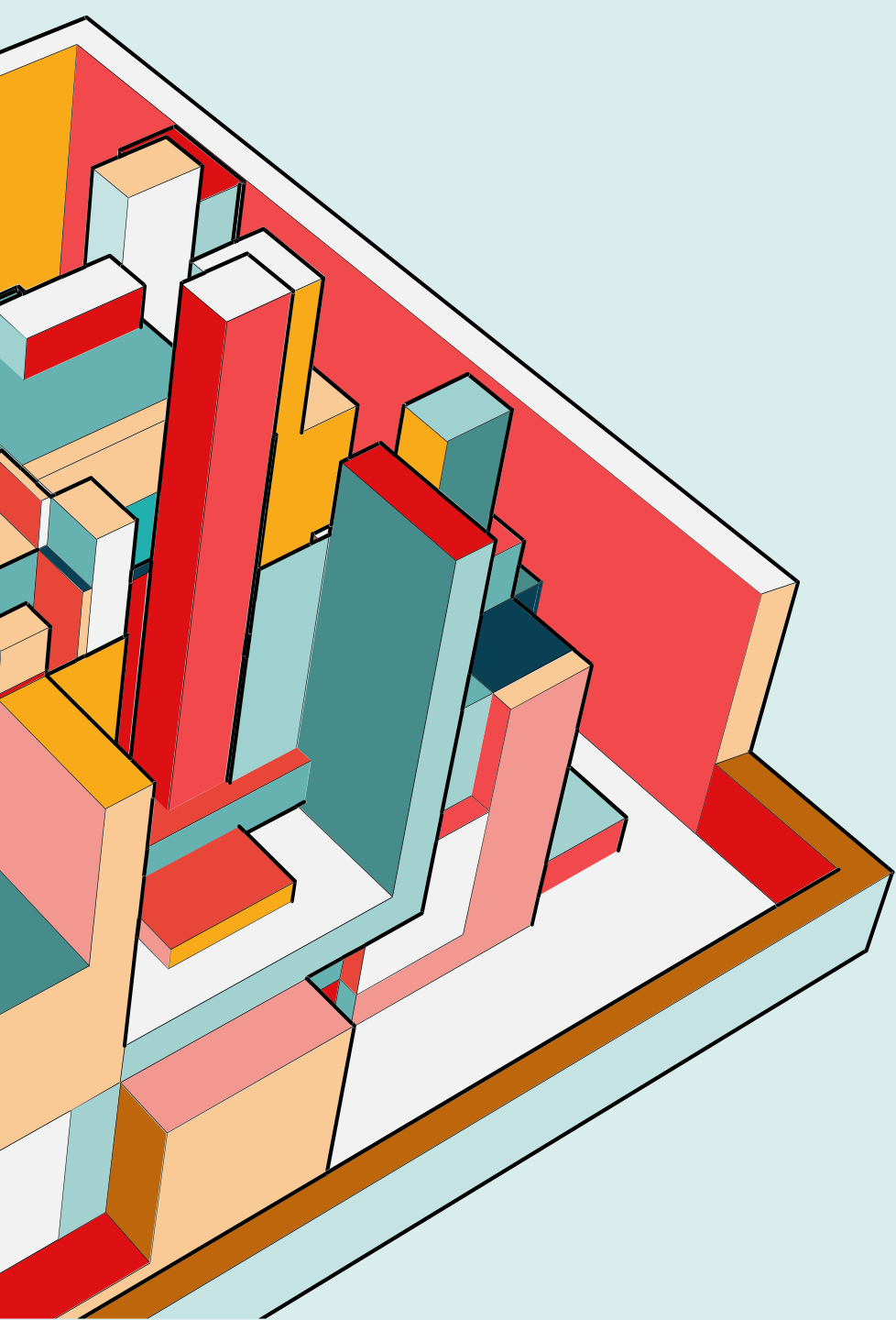
    public void Print() { Printer.Print(Text); }
}

class ConsolePrinter : IPrinter
{
    public void Print(string text)...
```

```
class HtmlPrinter : IPrinter
{
    public void Print(string text)...
```

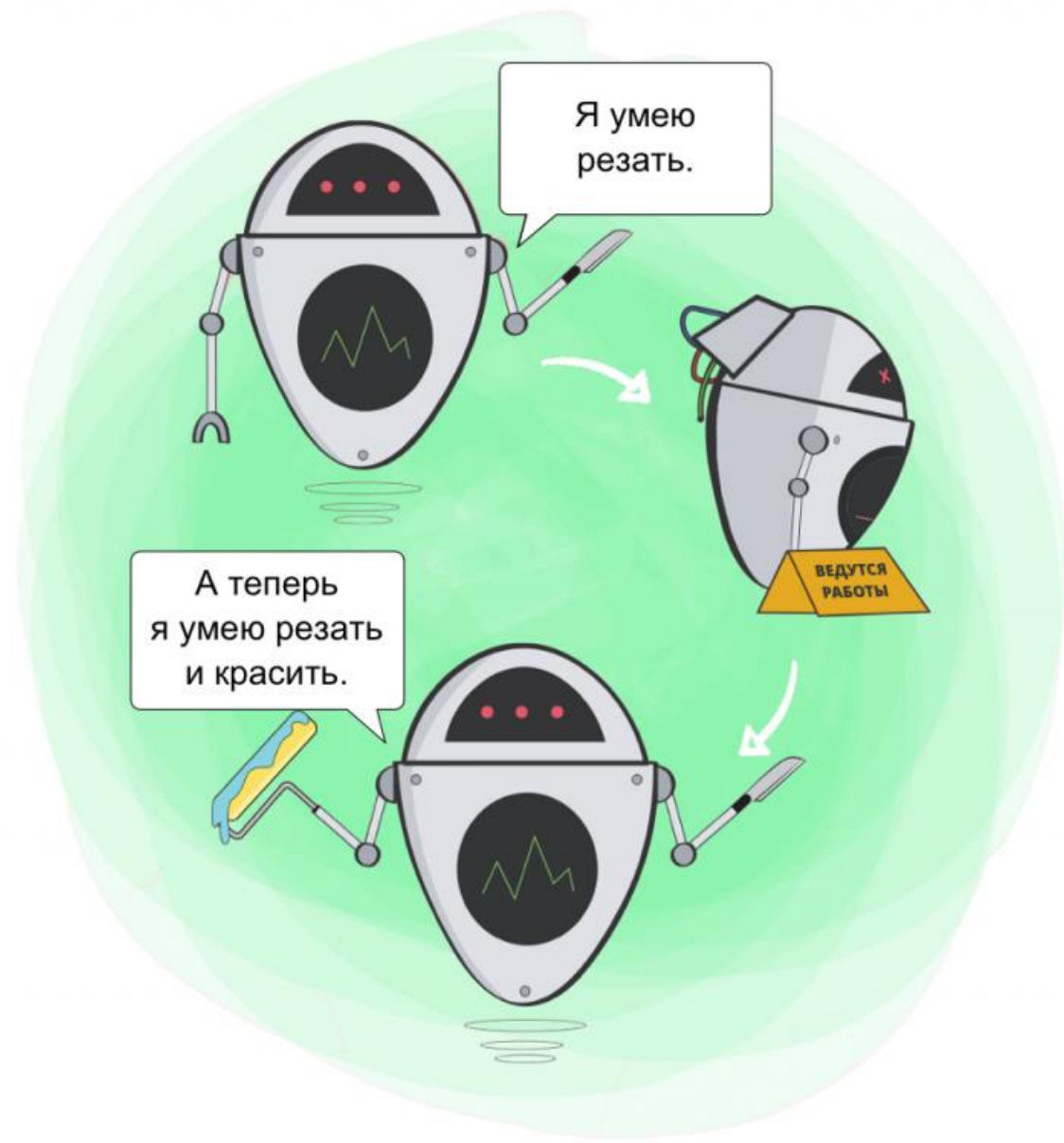
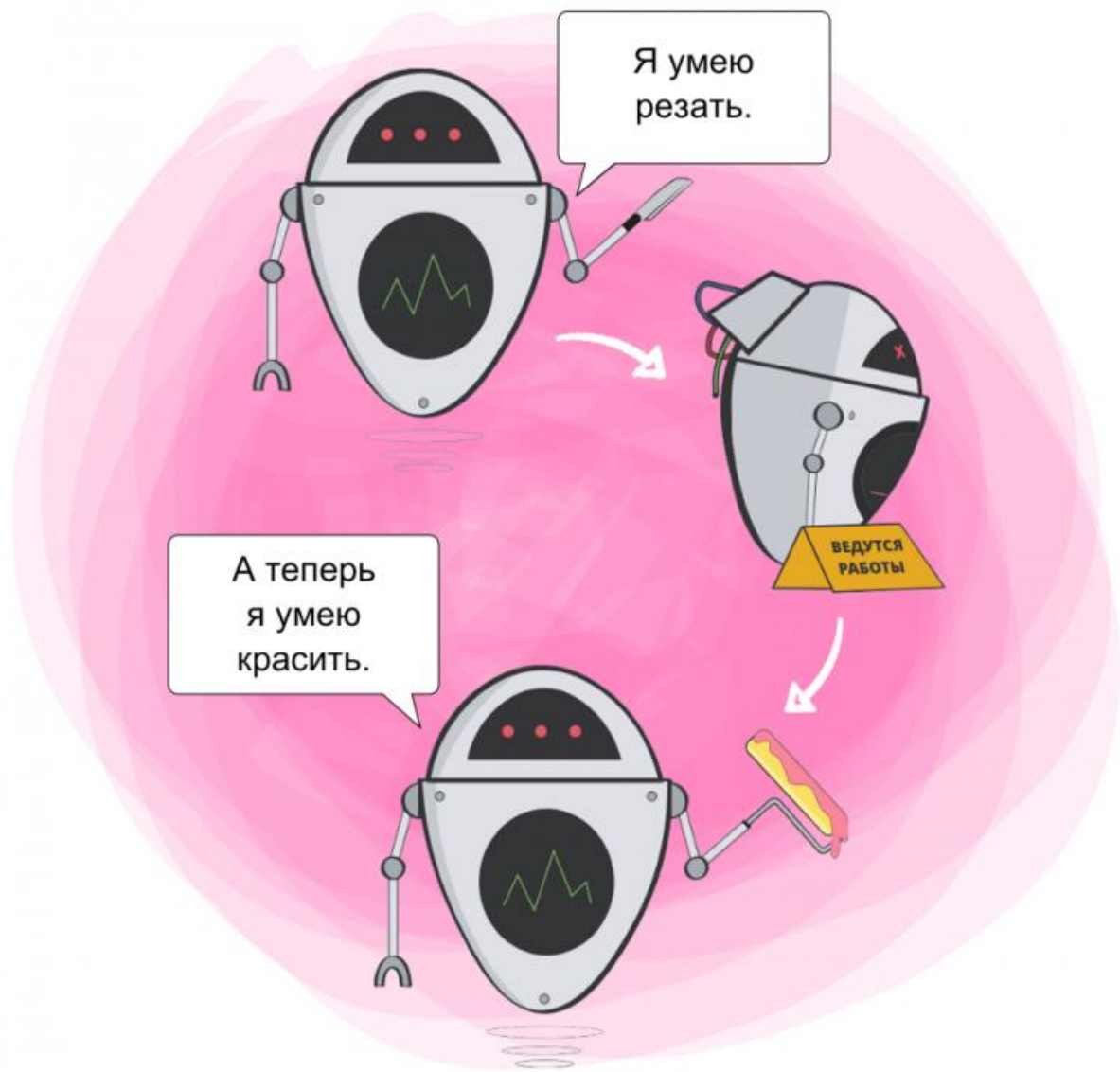
```
static void Main(string[] args)
{
    Book book = new Book(new ConsolePrinter());
    book.Print();

    book.Printer = new HtmlPrinter();
    book.Print();
}
```



# OPEN-CLOSED PRINCIPLE

Объекты должны быть открыты  
для расширения, но закрыты  
для модификации





```
class Cookbook
{
    public string Name { get; set; }

    public void CookDinner()
    {
        Console.WriteLine("Peeling potatoes");
        Console.WriteLine("Cooking potatoes");
        Console.WriteLine("Potatoes are ready");
    }
}
```

```
class Cookbook
{
    public string Name { get; set; }

    public void CookDinner()
    {
        Console.WriteLine("Peeling potatoes");
        Console.WriteLine("Cooking potatoes");
        Console.WriteLine("Potatoes are ready");
    }

    public void CookSalad()
    {
        Console.WriteLine("Preparing vegetables");
        Console.WriteLine("Cut vegetables");
        Console.WriteLine("Vegetables are ready");
    }
}
```

# КАК СОБЛЮДАЕТСЯ ОСР



Разделять модули, которые  
изменяются по разным причинам



Выстраивать зависимости таким  
образом, чтобы не было зависимостей  
от наиболее изменяемых модулей

```
interface IMeal
{
    void Make();
}
```

```
class PotatoMeal : IMeal
{
    public void Make()...
```

```
class SaladMeal : IMeal
{
    public void Make()...
```

```
public void Make()
{
    Console.WriteLine("Peeling potatoes");
    Console.WriteLine("Cooking potatoes");
    Console.WriteLine("Potatoes are ready");
}
```

```
class Cookbook
{
    public string Name { get; set; }

    public void CookDinner(IMeal meal)
    {
        meal.Make();
    }
}
```

```

abstract class MealBase
{
    public void Make()
    {
        Prepare();
        Cook();
        FinalSteps();
    }
    protected abstract void Prepare();
    protected abstract void Cook();
    protected abstract void FinalSteps();
}

```

```

class PotatoMeal : MealBase
{
    protected override void Cook()
    {
        Console.WriteLine("Cooking potatoes");
    }

    protected override void FinalSteps()
    {
        Console.WriteLine("Potatoes are ready");
    }

    protected override void Prepare()
    {
        Console.WriteLine("Peeling potatoes");
    }
}

class SaladMeal : MealBase
{
    protected override void Cook()...

    protected override void FinalSteps()...

    protected override void Prepare()...
}

```

```
class Cookbook
{
    public void MakeDinner(MealBase[] menu)
    {
        foreach (MealBase meal in menu)
            meal.Make();
    }
}
```

```
abstract class MealBase...
```

```
class PotatoMeal : MealBase
{
    protected override void Make()
    {
        Prepare();
        Cook();
        FinalSteps();
    }

    protected override void Prepare()...
```

```
abstract class MealBase
{
    public void Make()
    {
        Prepare();
        Cook();
        FinalSteps();
    }
    protected abstract void Prepare();
    protected abstract void Cook();
    protected abstract void FinalSteps();
}
```

```
class SaladMeal ...
```

## Option 1.

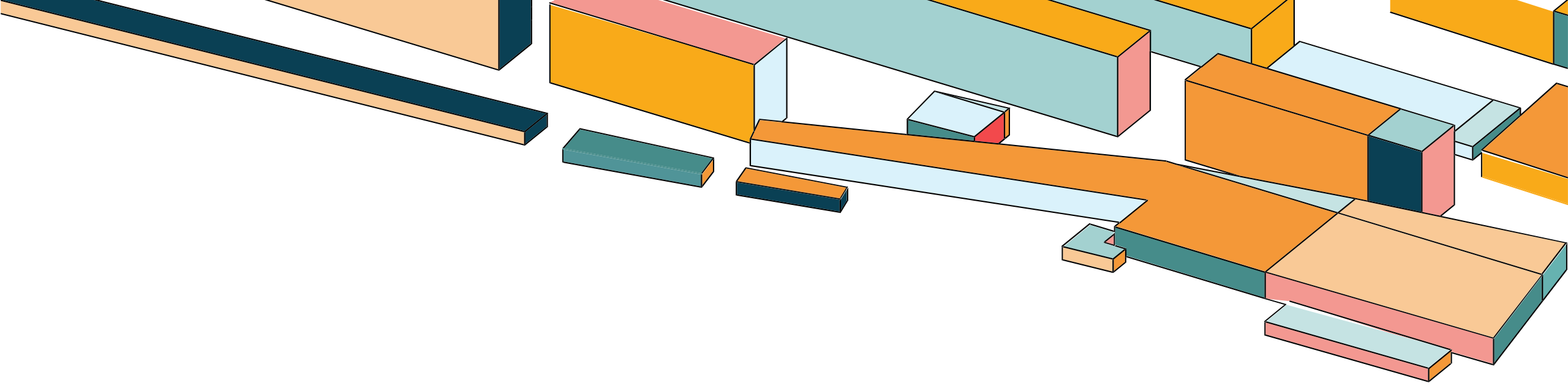
```
class Program
{
    static void Main(string[] args)
    {
        Cookbook cookbook = new Cookbook();

        cookbook.CookDinner(new PotatoMeal());
        cookbook.CookDinner(new SaladMeal());
    }
}
```

## Option 2.

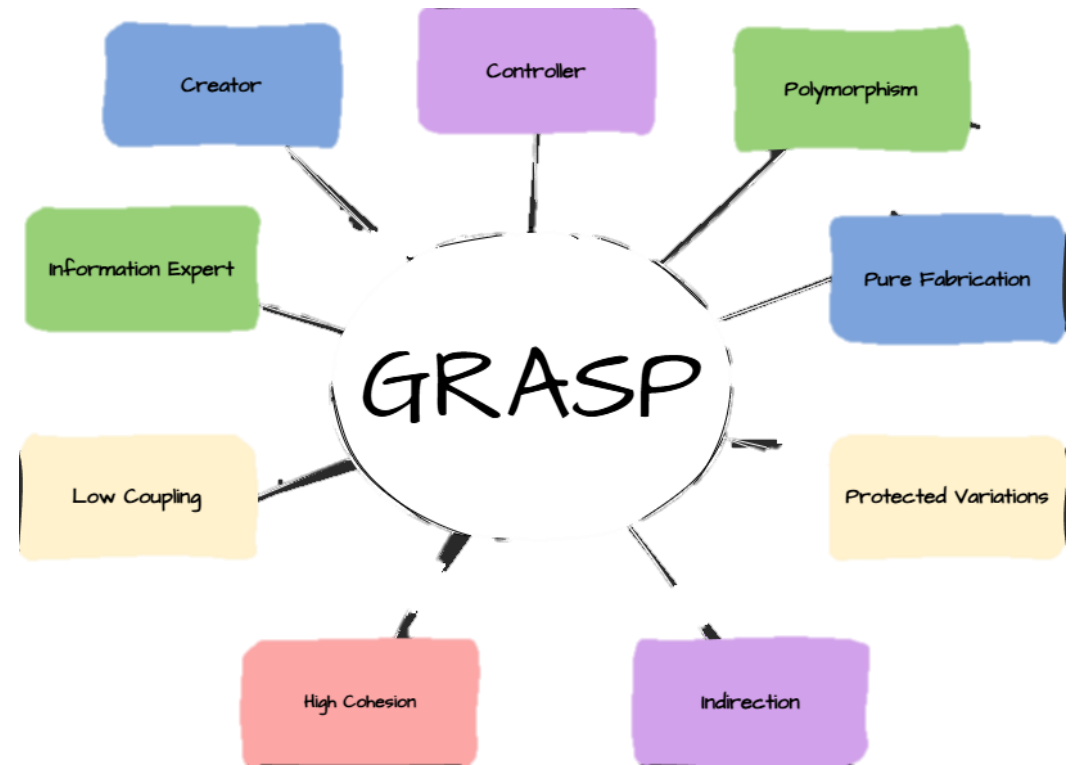
```
class Program
{
    static void Main(string[] args)
    {
        MealBase[] menu = new MealBase[]
        { new PotatoMeal(), new SaladMeal() };

        Cookbook cookbook = new Cookbook();
        cookbook.MakeDinner(menu);
    }
}
```



General Responsibility Assignment Software Patterns

# ВОЗЬМИ ОТВЕТСТВЕННОСТЬ В СВОИ РУКИ С GRASP ПРИНЦИПАМИ





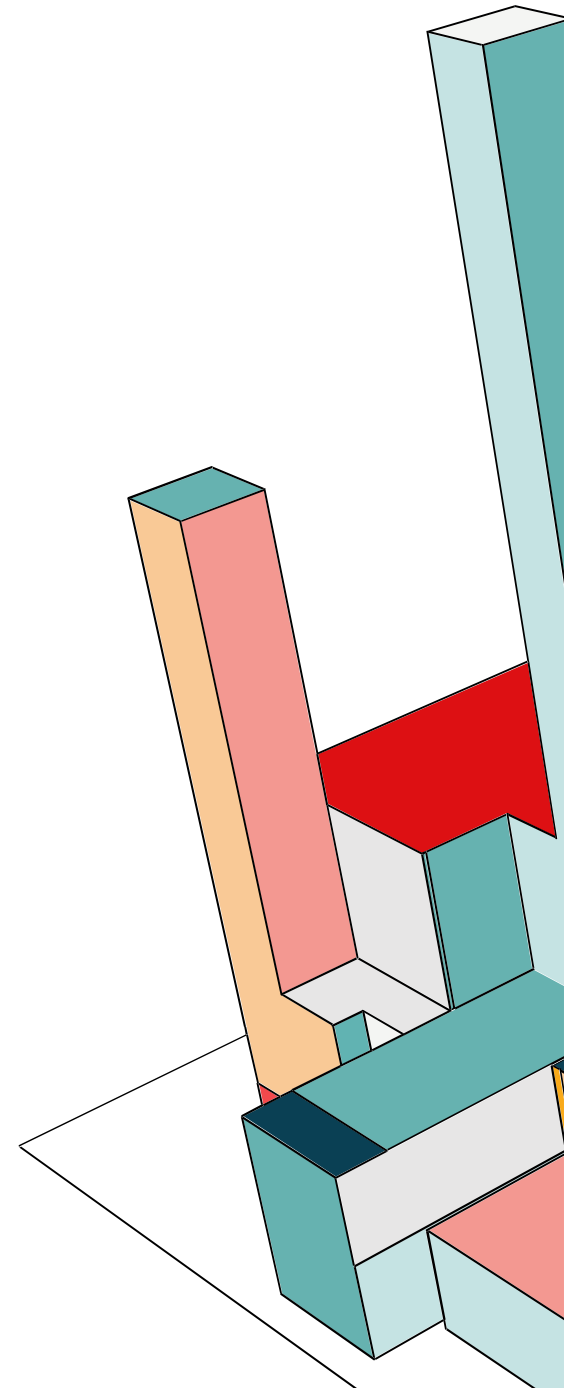
# КЛАССИФИКАЦИЯ ШАБЛОНОВ GRASP

## Основные шаблоны

- High Cohesion (Высокая сцепленность)
- Low Coupling (Низкое связывание)
- Information expert (Информационный эксперт)
- Creator (Создатель)
- Controller (Контроллер)

## Дополнительные шаблоны

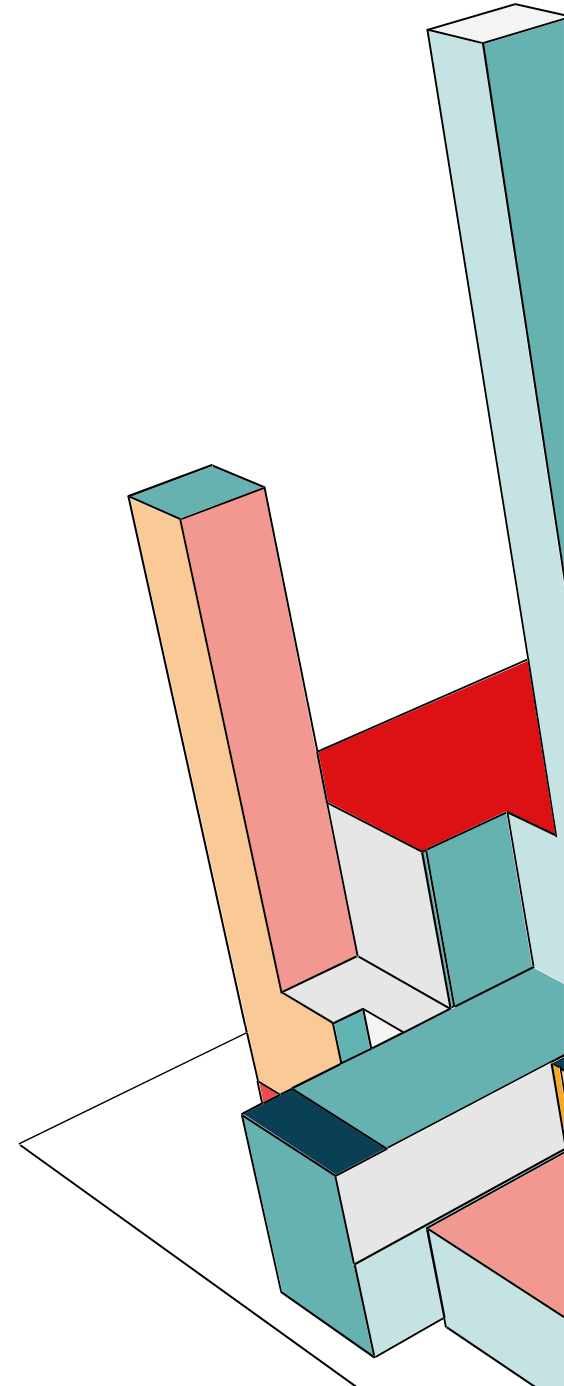
- Protected Variations (Устойчивость к изменениям)
- Polymorphism (Полиморфизм)
- Indirection (Перенаправление)
- Pure Fabrication (чистая выдумка, синтетика)



# HIGH COHESION & LOW COUPLING

"... система должна состоять из слабо связанных классов, которые должны содержать связанную бизнес — логику."

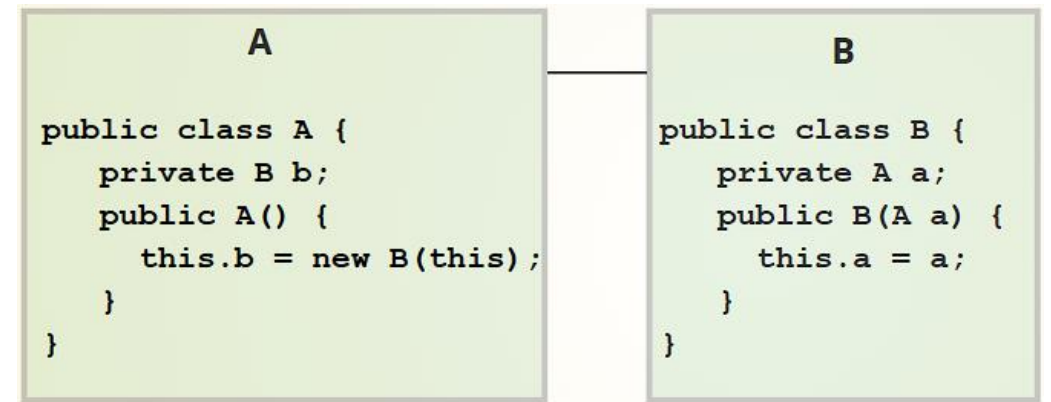
Эти два принципа рассматриваются совместно, и говорят о том, что классы целесообразно строить таким образом, чтобы они имели сильную внутреннюю связь и были слабо связаны между собой



# НИЗКОЕ СВЯЗЫВАНИЕ

Как обнулить связывание?

1. Удалить атрибуты нестандартных типов
2. Не вызывать службы классов другого типа
3. Для параметров, возвращаемых значений и локальных переменных избавиться от нестандартных типов
4. Отказаться от наследования и реализации интерфейсов

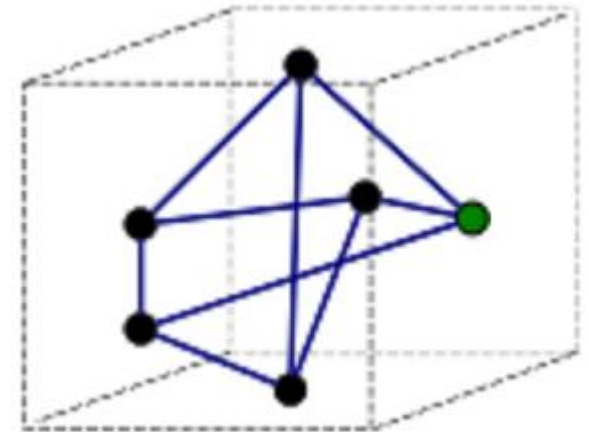


Если возвести Low Coupling в абсолют,  
то вся функциональность будет реализована  
в одном единственном классе

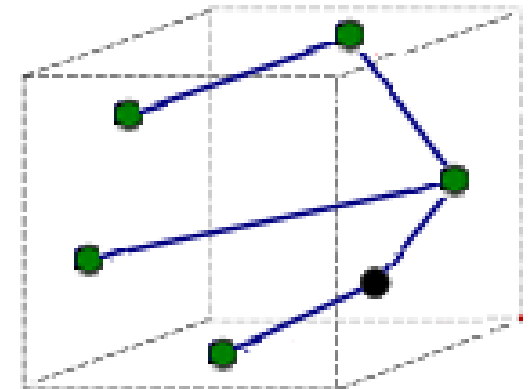
# ВЫСОКАЯ СЦЕПЛЕННОСТЬ

Классы должны содержать сцепленную бизнес — логику!

Модульность - это свойство системы, разбитой на множество модулей с **высокой** степенью связности и **слабым** зацеплением



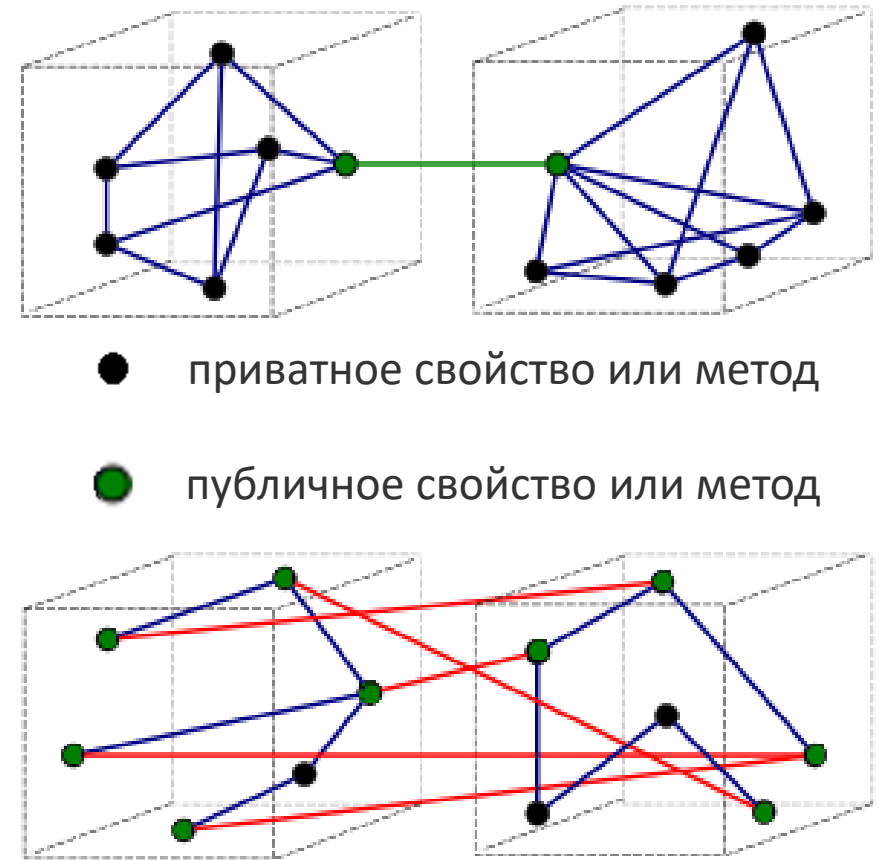
- приватное свойство или метод
- публичное свойство или метод



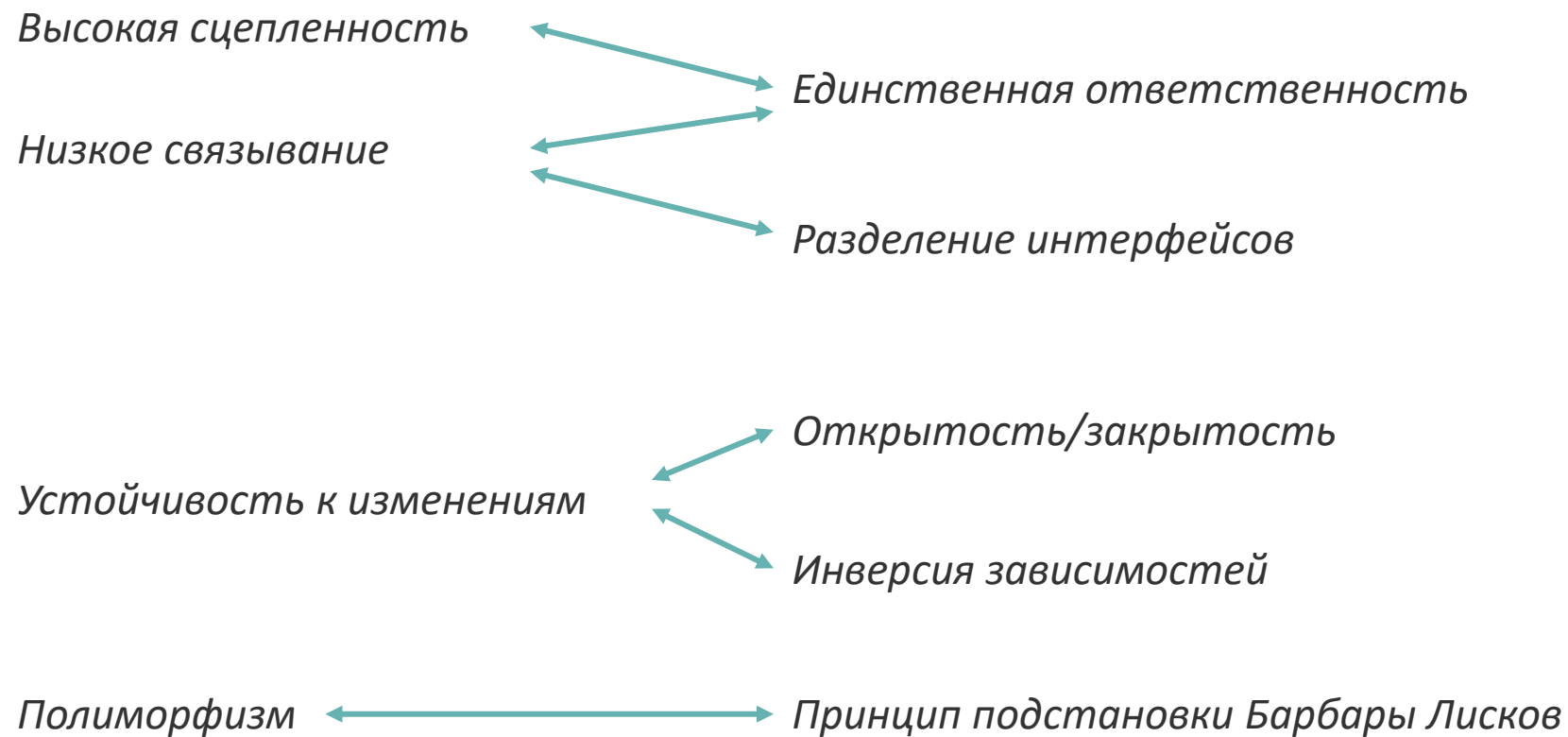
# МАКСИМИЗИРУЙТЕ СФОКУСИРОВАННОСТЬ МИНИМИЗИРУЙТЕ ЗАВИСИМОСТИ

В идеальном случае, класс имеет одну точку связывания, и пара таких классов может быть соединена единственным образом.

*Совет от Лармана: классы, которые являются достаточно общими и с высокой вероятностью будут повторно использоваться в дальнейшем, должны иметь минимальную степень зацепления с др. классами*

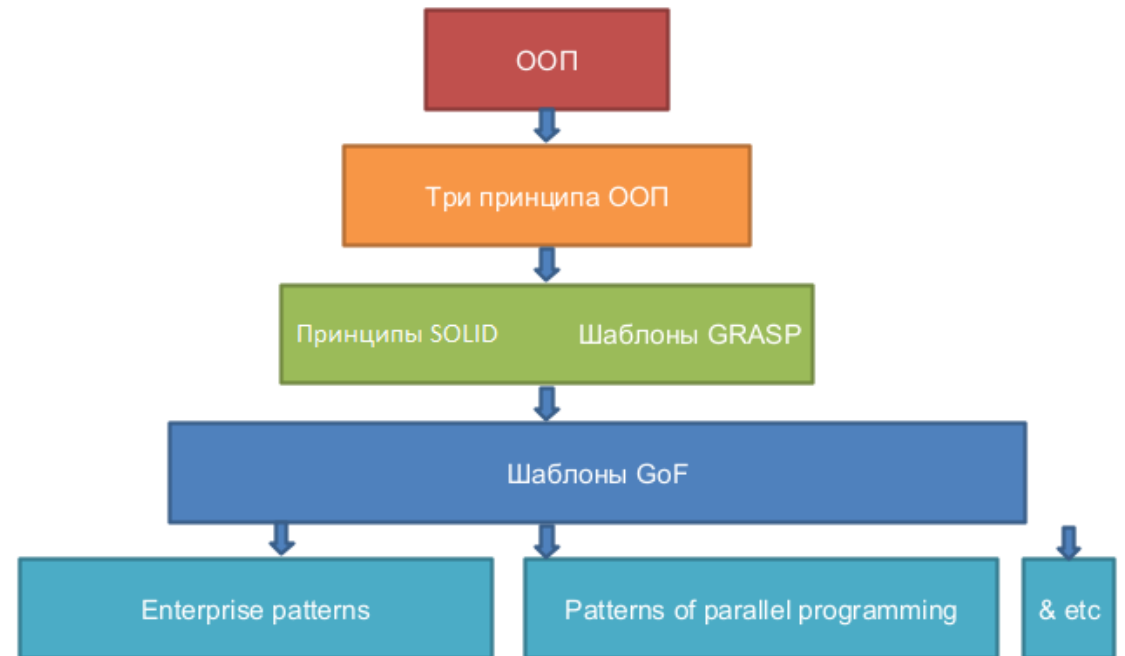


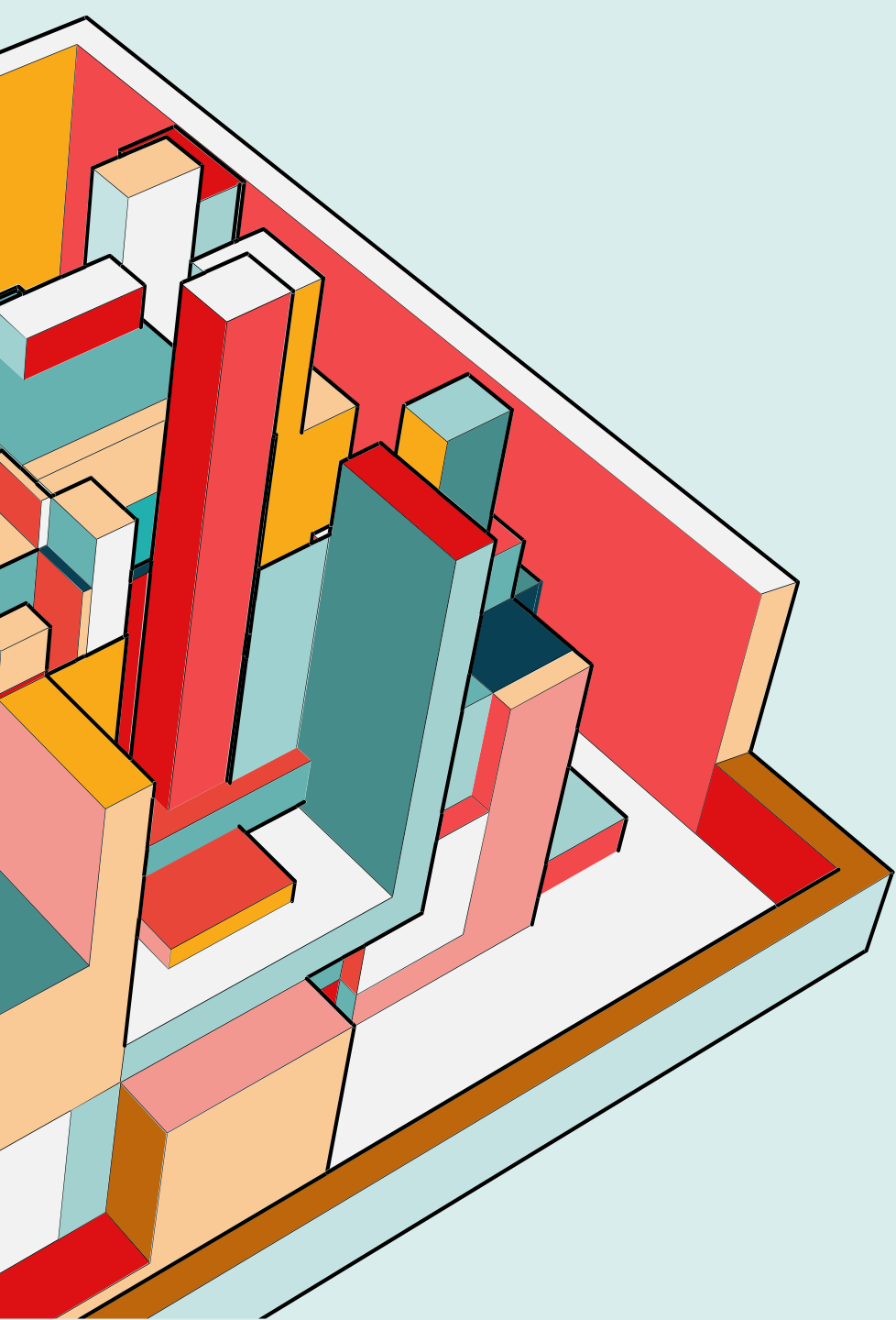
# СВЯЗЬ GRASP С ПРИНЦИПАМИ SOLID



# GRASP

- Предпочтение "простым" классам, экспертам своей ответственности
- При усложнении определенная функциональность или связь с другими классами выносятся в отдельную структуру
- Точки изменения защищаются с помощью абстрактных (полиморфных) операций (необходимо оценивать вероятность возможных модификаций)
- Для конкретной решения задачи можно "синтезировать" класс, отсутствующий в предметной области (например, Контроллер)





## ЭТО ЕЩЕ НЕ ВСЁ? ДРУГИЕ ПРИНЦИПЫ:

- ☐ **KISS**    Keep It Simple, Stupid!
- ☐ **DRY**    Don't Repeat Yourself
- ☐ **YAGNI**    You Ain't Gonna Need It



# YAGNI

You Aren't Gonna Need It /  
Вам это не понадобится

Этот принцип применим при рефакторинге.

Не бойтесь удалять лишние методы. Даже если раньше они были полезны – теперь они не нужны.

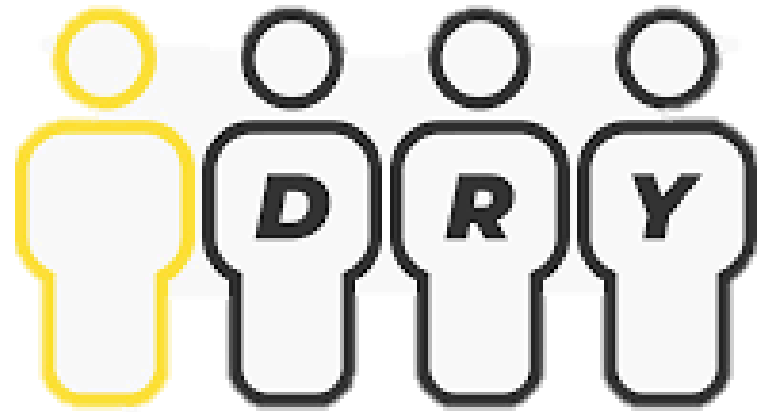


# DRY

## Don't Repeat Yourself / Не повторяйтесь

Прежде чем что-либо писать, проявите прагматизм: посмотрите. Возможно, эта функция где-то реализована.

Возможно, эта бизнес-логика существует в другом месте. Повторное использование кода — всегда разумное решение.



# KISS

## Keep It Simple, Stupid / Будь проще

Этот принцип был разработан ВМС США в 1960 году. Этот принцип гласит, что простые системы будут работать лучше и надежнее.

Применительно к разработке ПО он значит следующее – не придумывайте к задаче более сложного решения, чем ей требуется.



# BDUF

## Big Design Up Front / Глобальное проектирование прежде всего

Прежде чем переходить к реализации, убедитесь, что все хорошо продумано.

Очень распространенный контраргумент заключается в том, что стоимость решения проблем зачастую ниже стоимости времени планирования.



# APO

## Avoid Premature Optimization / Избегайте преждевременной оптимизации

Если вы следуете KISS или YAGNI, вы не попадетесь на этот крючок.

Очень простой пример – масштабирование. Вы не станете покупать 40 серверов из предположения, что ваше новое приложение станет очень популярным.

Преждевременная оптимизация - корень всех зол.



# БРИТВА ОККАМА

Уильям Оккам: Не следует  
множить сущее без  
необходимости

Не создавайте ненужных сущностей без необходимости. Будьте прагматичны — подумайте, нужны ли они, поскольку они могут в конечном итоге усложнить вашу кодовую базу.

Когда умножил сущее  
без необходимости:

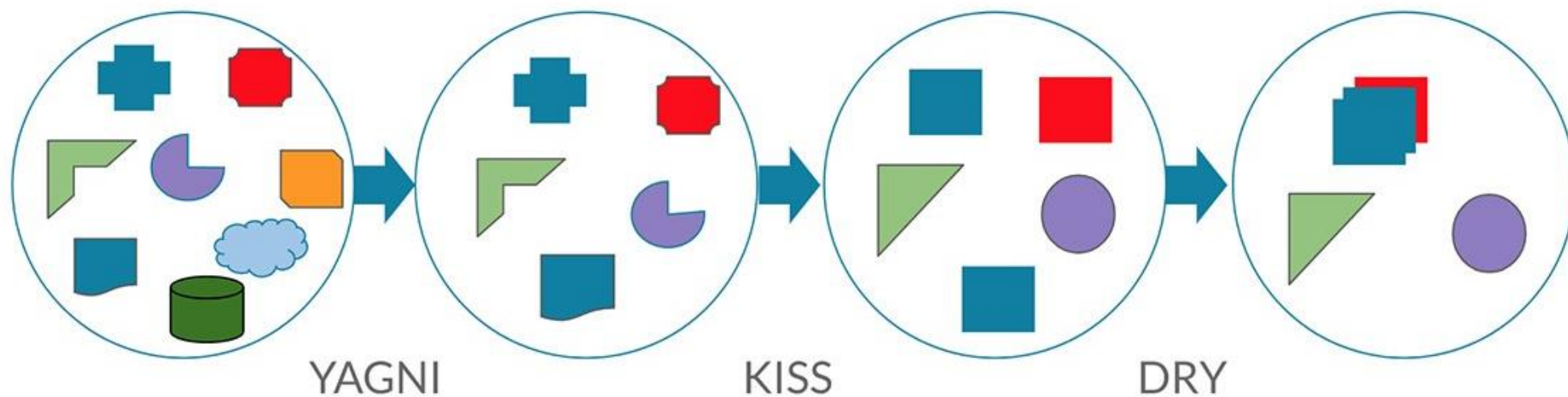
Ну ты меня совсем  
не уважаешь, что ли? ((



# ЖИЗНЬ СЛОЖНЕЕ...

Бывает, что SOLID противоречит ягни, драй и кису.

Иногда они успешно сосуществуют в некоторых местах. Но есть достаточно случаев когда надо выбирать. Лично я агитирую за выбор соблюдения SOLID.



# СПАСИБО!

Брита Тамм

502-555-0152

[brita@firstupconsultants.com](mailto:brita@firstupconsultants.com)

[www.firstupconsultants.com](http://www.firstupconsultants.com)

