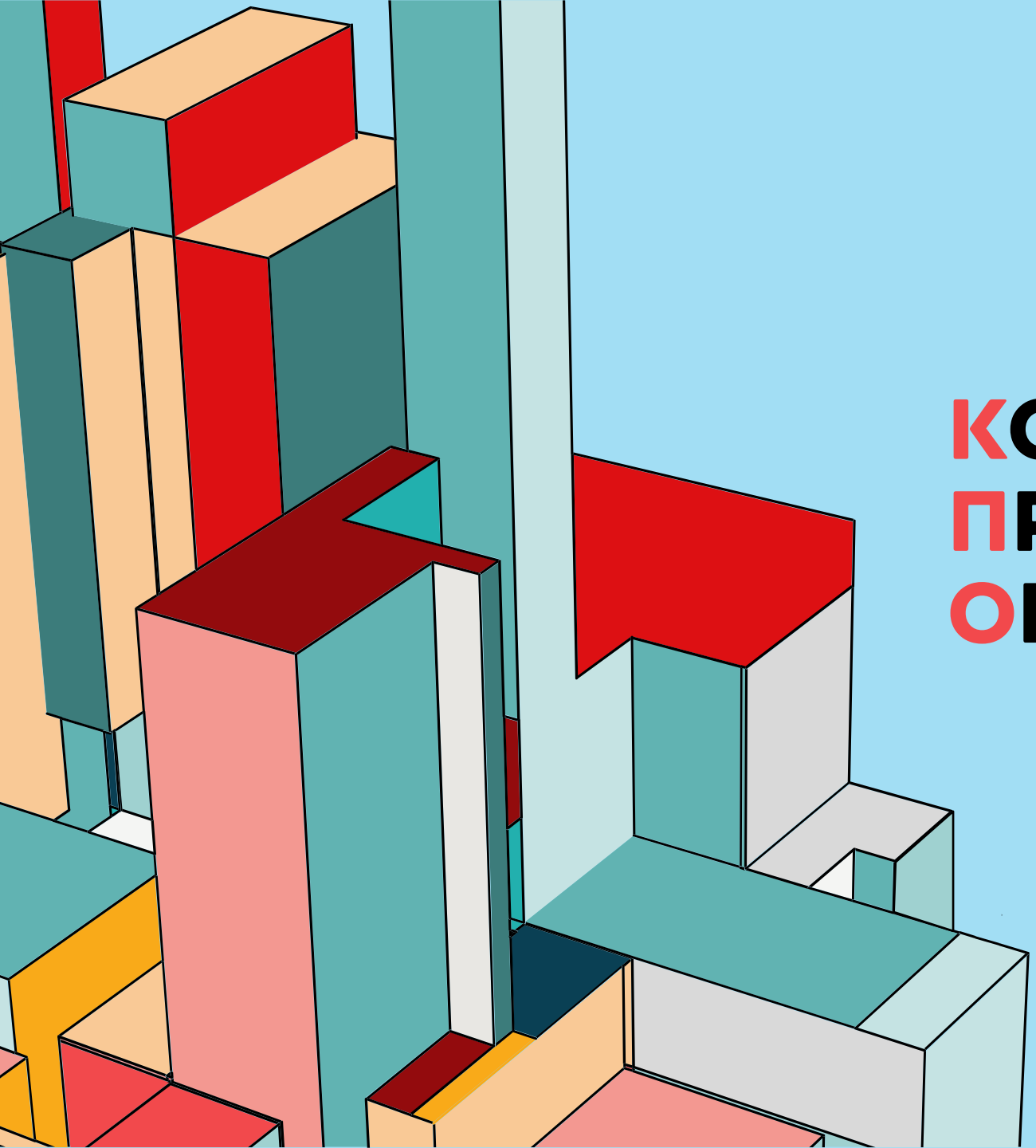
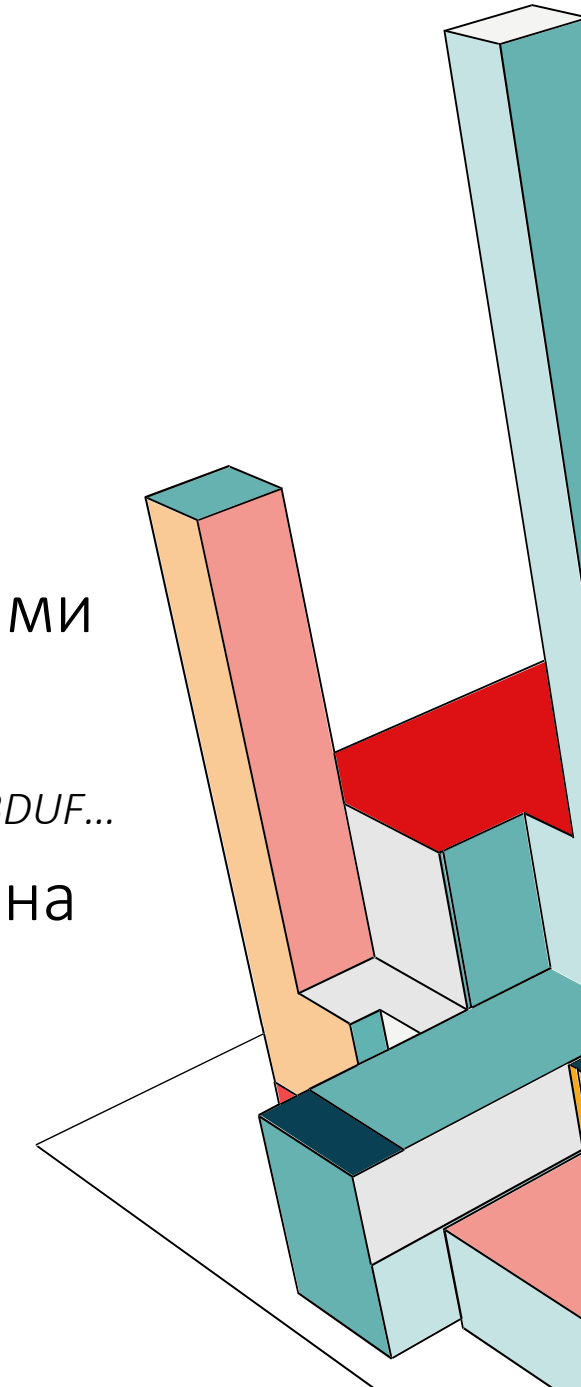


КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

План лекции

- О курсе
- Парадигмы программирования
- Вспомним ООП: отношения между классами и объектами
- Интерфейсы vs Абстрактные классы
- Принципы дизайна: SOLID + GRASP, а также *KISS, DRY, YAGNI, BDUF…*
- ~~Душные~~ Советы и рекомендации по принципам дизайна



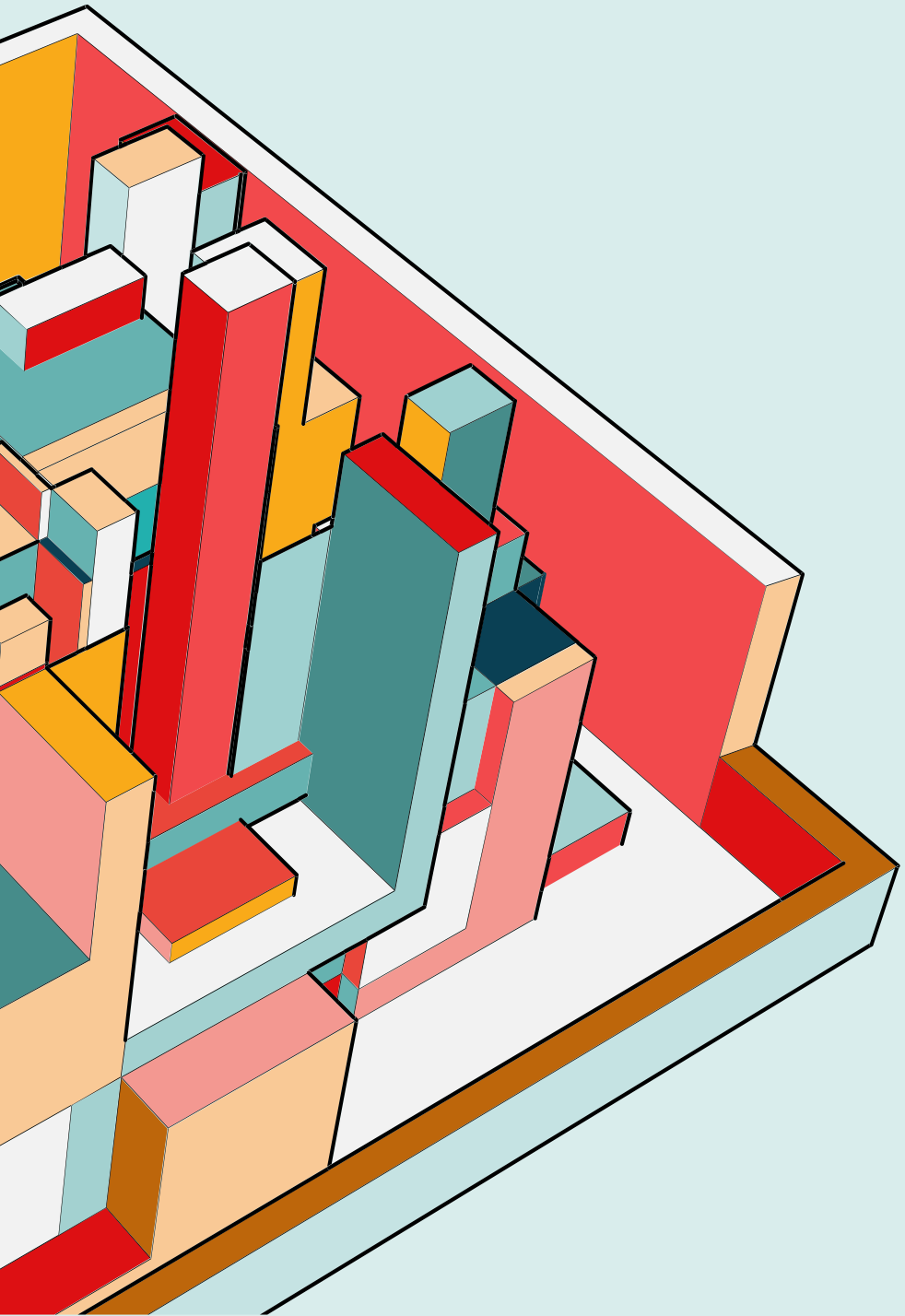
ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

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, МакКарти	1936 1958

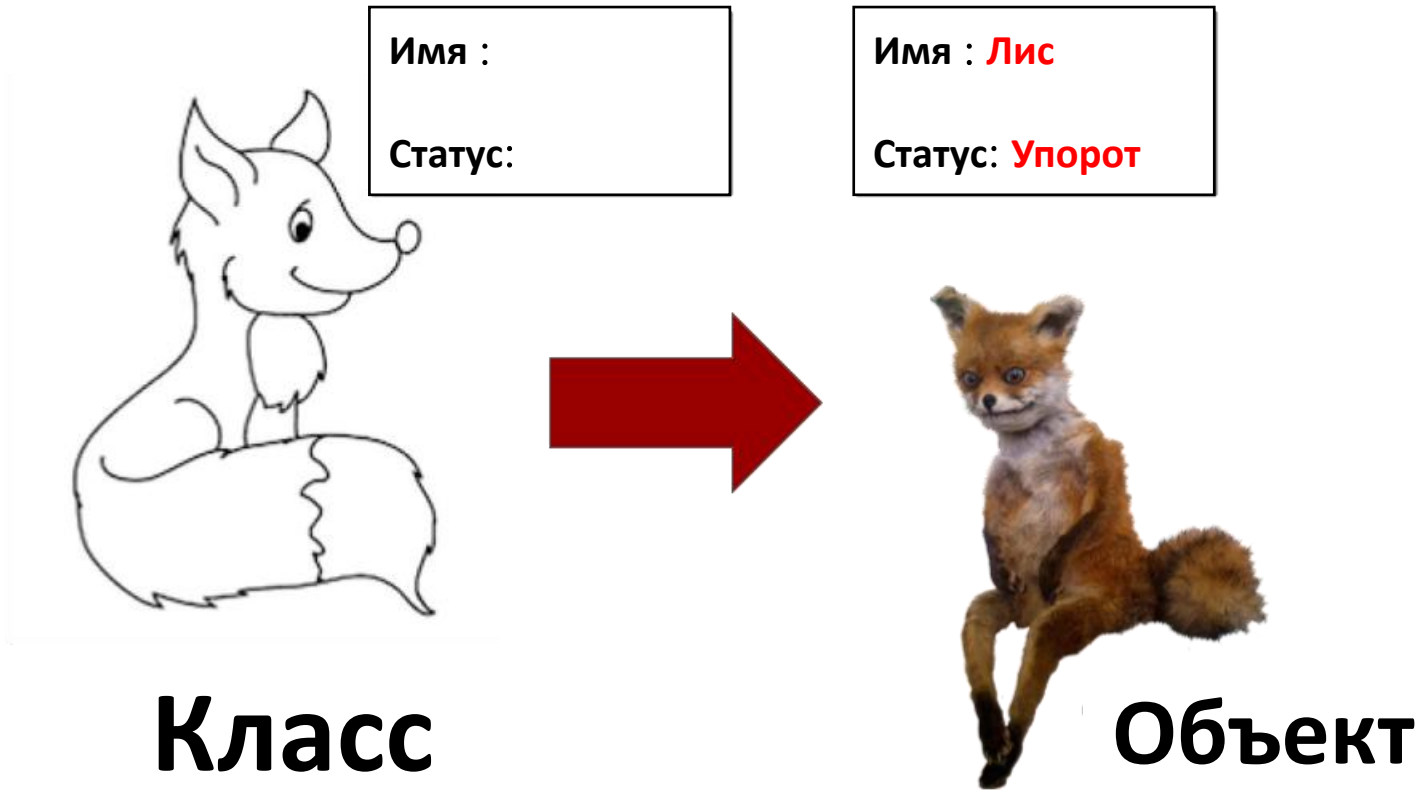
Каждая из трёх парадигм ограничивает нас в чём-то:

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



ВСПОМНИТЬ ~~ВСЁ~~ ООП

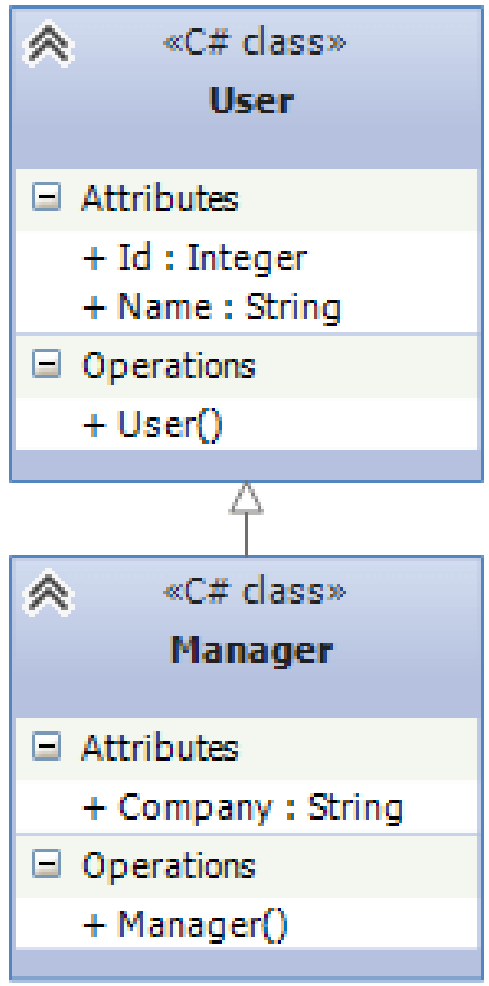
КЛАСС ОПИСЫВАЕТ ТО, КАКИМ ОБЪЕКТ МОЖЕТ БЫТЬ



ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ И ОБЪЕКТАМИ

НАСЛЕДОВАНИЕ

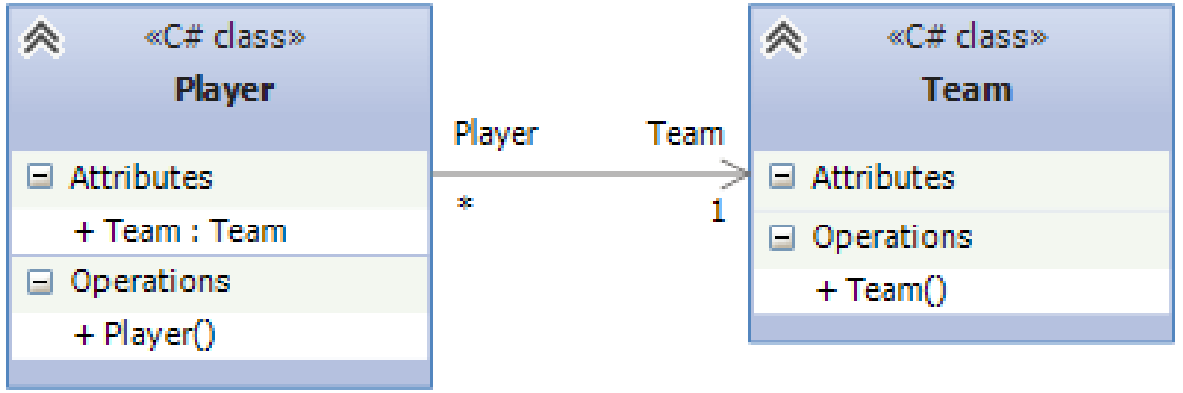
```
class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Manager : User
{
    public string Company { get; set; }
}
```



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

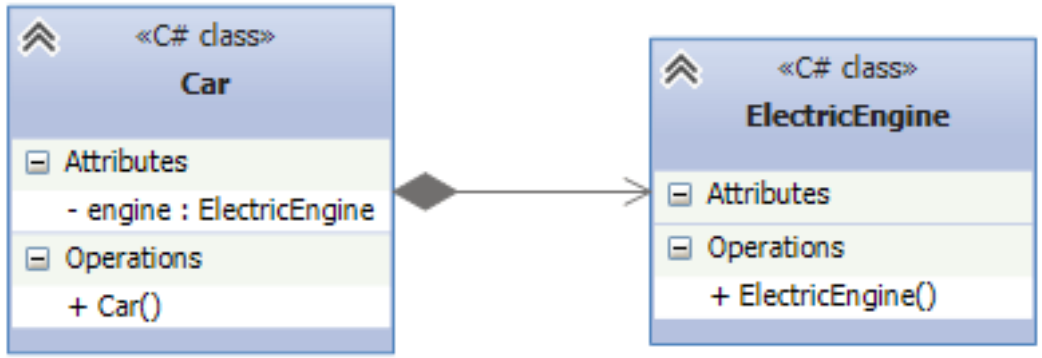
АССОЦИАЦИЯ

```
class Team
{

}
class Player
{
    public Team Team { get; set; }
}
```



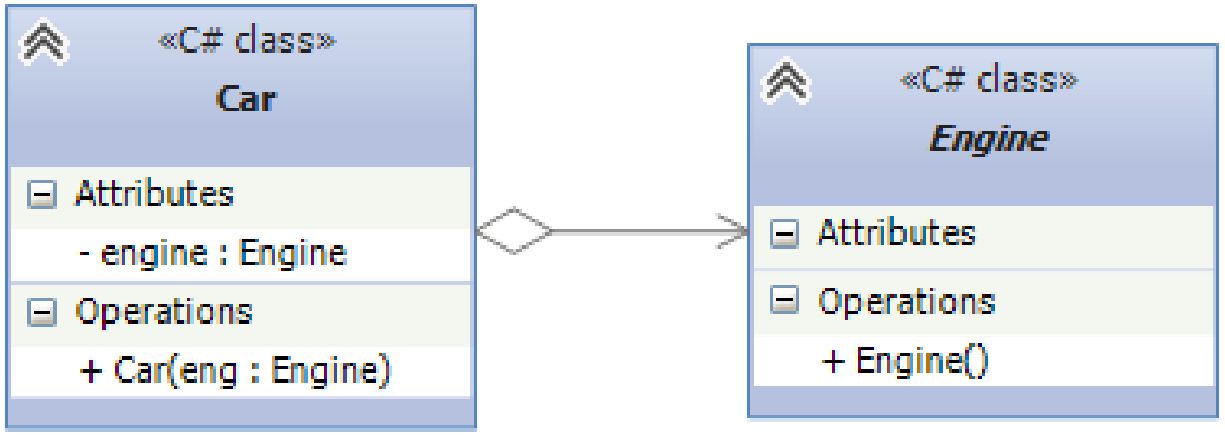
КОМПОЗИЦИЯ

```
public class ElectricEngine
{}

public class Car
{
    ElectricEngine engine;
    public Car()
    {
        engine = new ElectricEngine();
    }
}
```



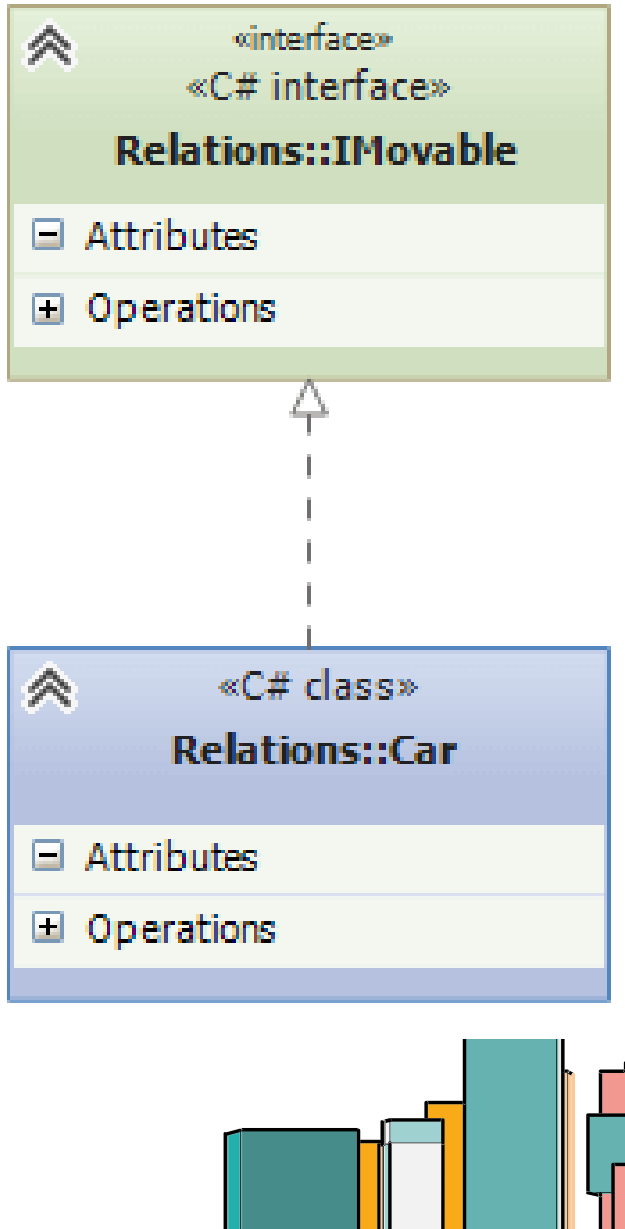
АГРЕГАЦИЯ

```
public abstract class Engine
{ }

public class Car
{
    Engine engine;
    public Car (Engine eng)
    {
        engine = eng;
    }
}
```



РЕАЛИЗАЦИЯ

```
public interface IMovable
{
    void Move();
}

public class Car : IMovable
{
    public void Move()
    {
        Console.WriteLine ("Машина едет");
    }
}
```



ИНТЕРФЕЙСЫ ИЛИ АБСТРАКТНЫЕ КЛАССЫ

ОБЕЗЬЯНА

```
public abstract class Vehicle
{
    public abstract void Move();
}

public class Car : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Машина едет");
    }
}

public class Bus : Vehicle
{
    public override void Move()
    {
        Console.WriteLine("Автобус едет");
    }
}
```

Блок класса Vehicle

Car, наследуемый от класса Vehicle

Bus наследуемый от класса Vehicle

ПРОГРАММИСТ

```
public interface IMovable
{
    void Move();
}

public abstract class Vehicle : IMovable
{
    public abstract void Move();
}

public class Car : Vehicle
{
    public override void Move() => Console.WriteLine("Машина едет");
}

public class Horse : IMovable
{
    public void Move() => Console.WriteLine("Лошадь скачет");
}
```

КОГДА СЛЕДУЕТ ИСПОЛЬЗОВАТЬ

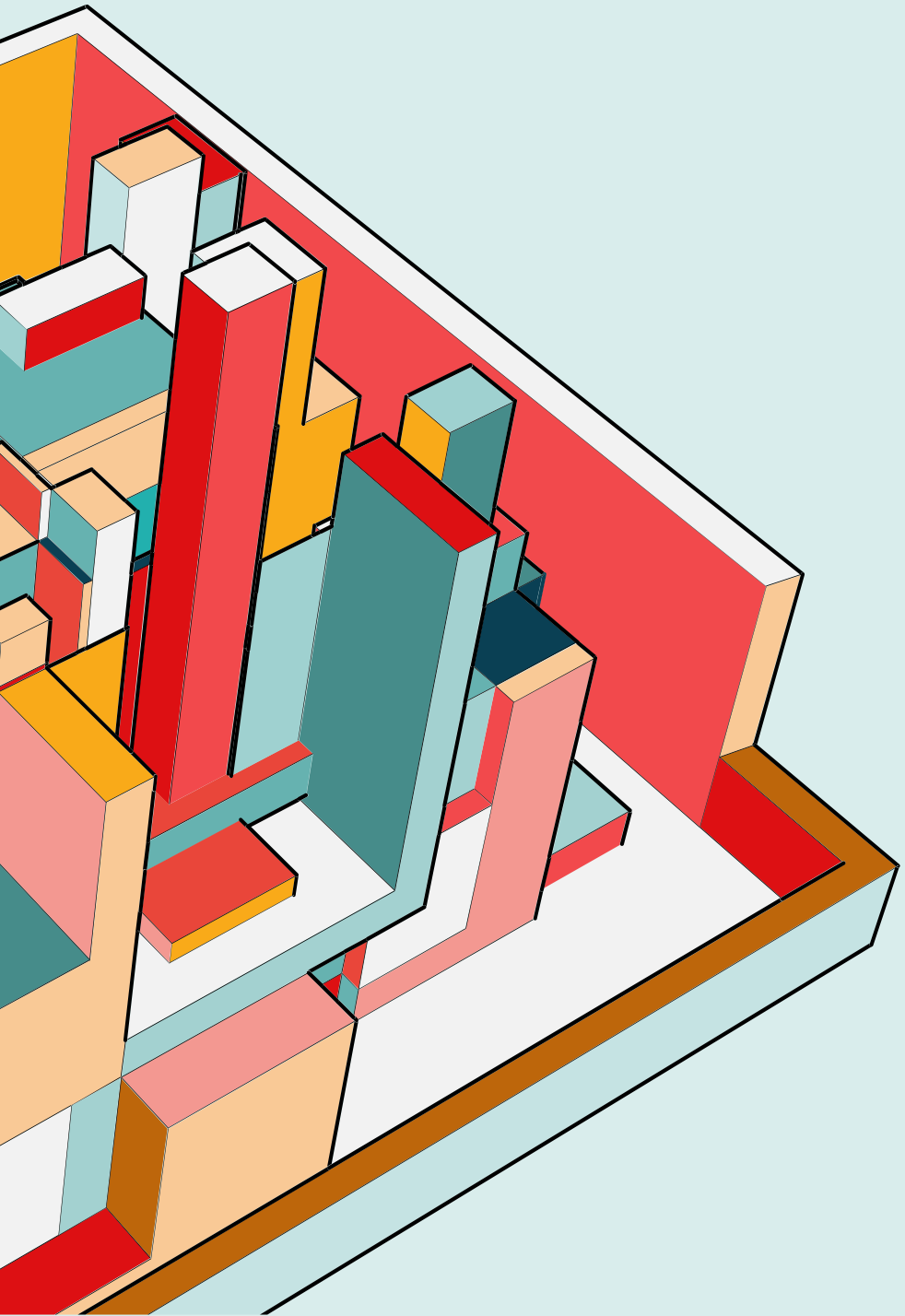
Абстрактные классы:

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

Интерфейсы:

- Определить функционал для группы разрозненных объектов, которые могут быть никак не связаны между собой.
- Проектируем небольшой функциональный тип

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



ПРИНЦИПЫ ДИЗАЙНА

SOLID

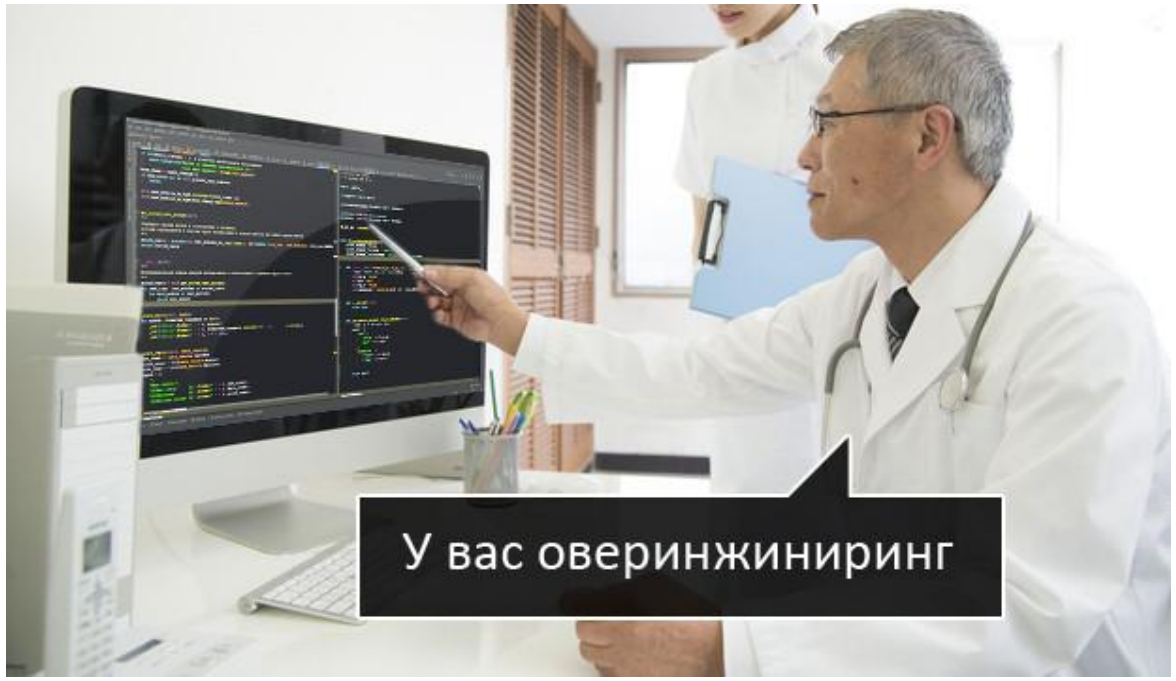
GRASP

KISS, DRY, YAGNI, BDUF,

APO и бритва Оккама

ВЫБОР УРОВНЯ АБСТРАКЦИИ

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



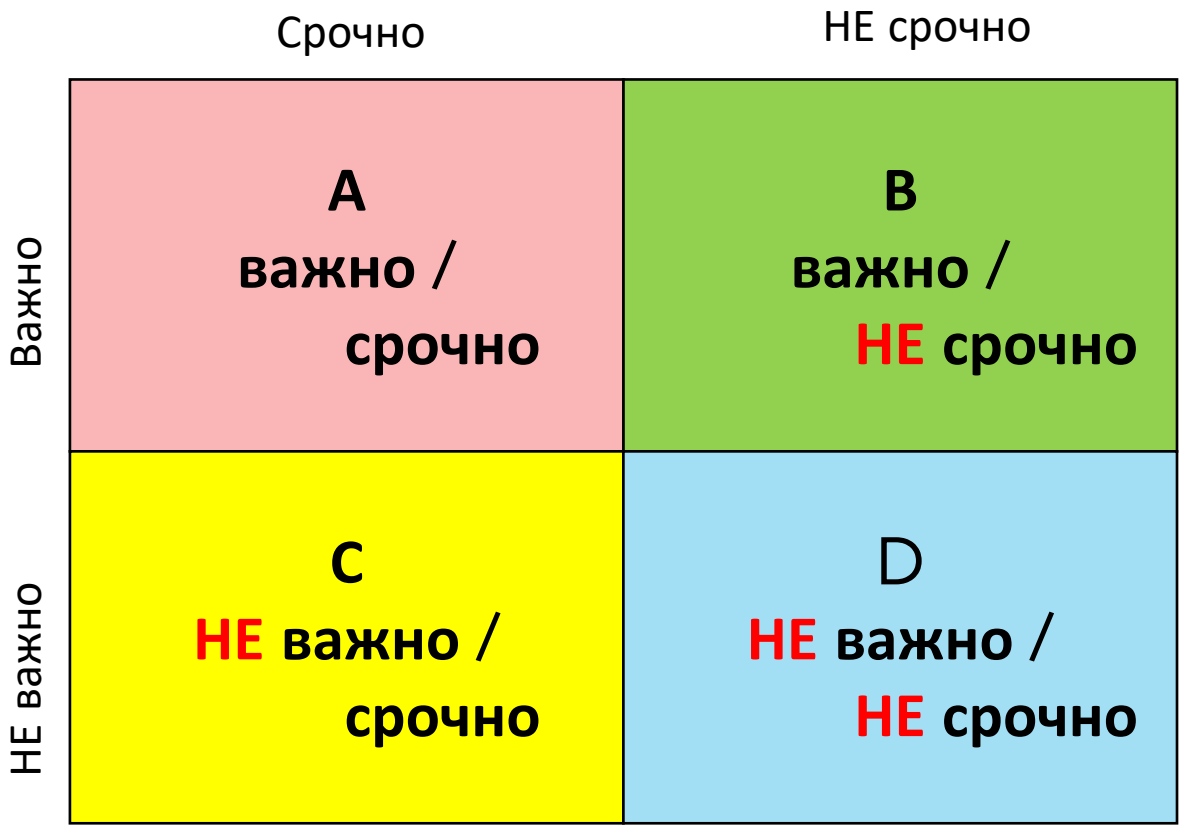
Любую архитектурную проблему можно решить добавлением дополнительного слоя абстракции,
кроме проблемы большого количества абстракций

МАТРИЦА ЭЙЗЕНХАУЭРА

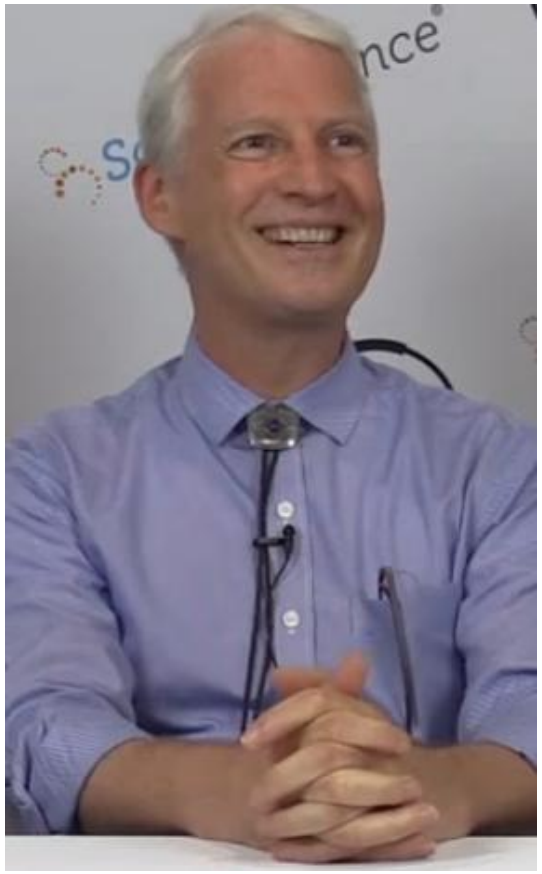
Что важнее – работающее ПО
или ПО с идеальной архитектурой?

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.



Программист должен бороться за архитектуру.
Архитектура – это его ответственность.



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

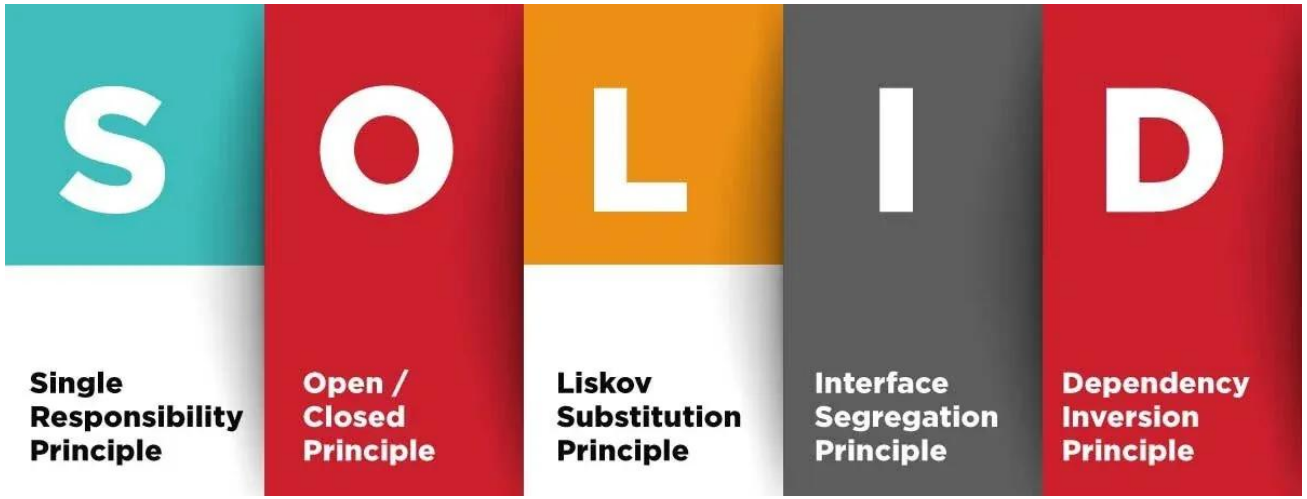
Соавтор Scrum

Чистый код

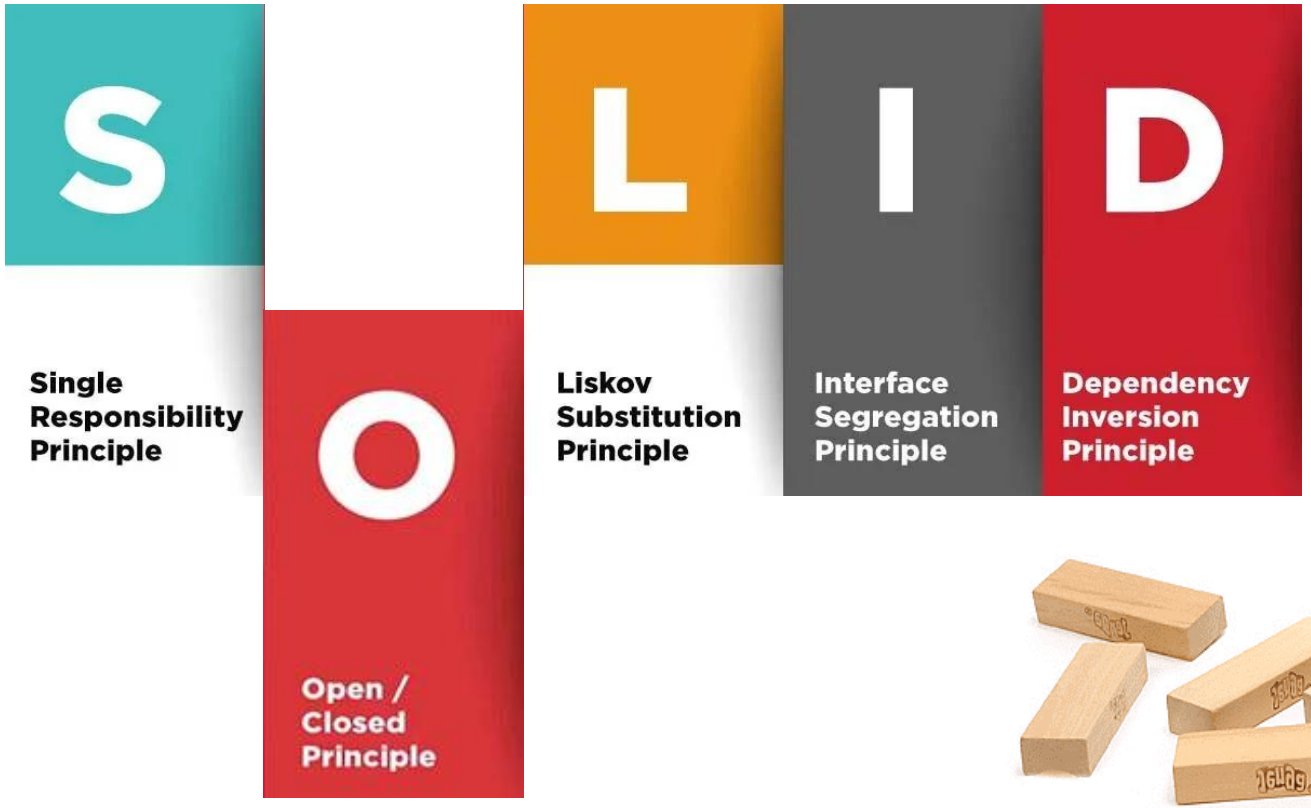
Чистая архитектура

Гибкая разработка

ПРОГРАММИРОВАНИЕ



ПРОГРАММИРОВАНИЕ — ЭТО

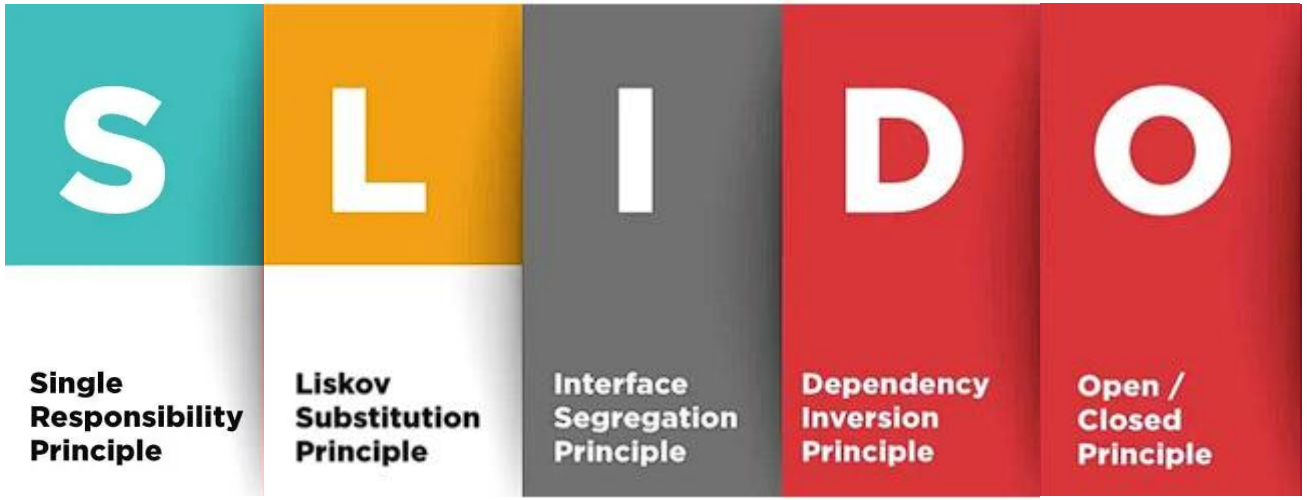


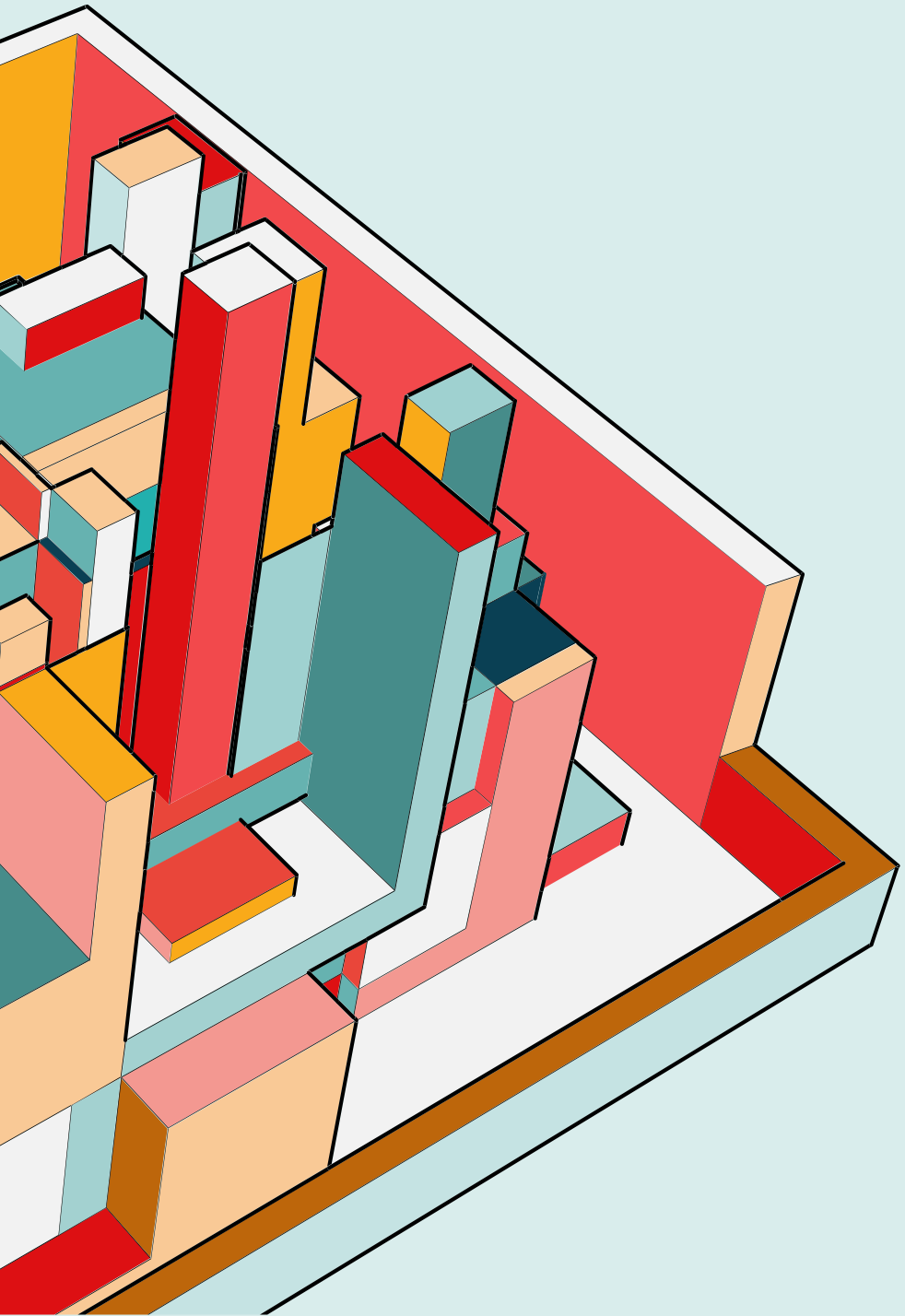
ПРОГРАММИРОВАНИЕ — ЭТО НЕ

S — Single Responsibility Principle

SLIDO — Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle, Open / Closed Principle

ПРОГРАММИРОВАНИЕ — ЭТО НЕ ДЖЕНГА





SINGLE-RESPONSIBILITY PRINCIPLE

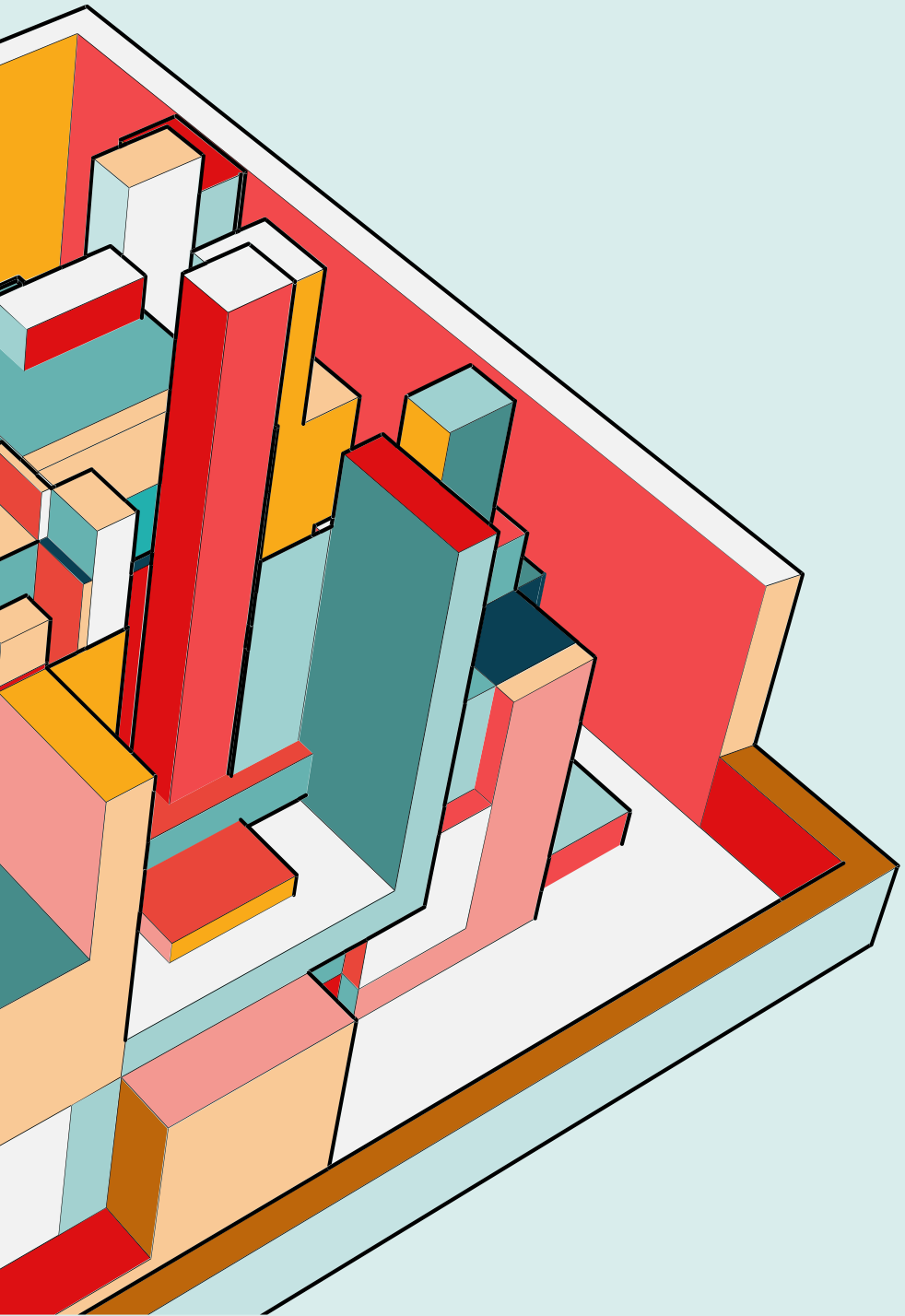
Класс должен иметь одну и только одну причину измениться

```
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()...
}
```

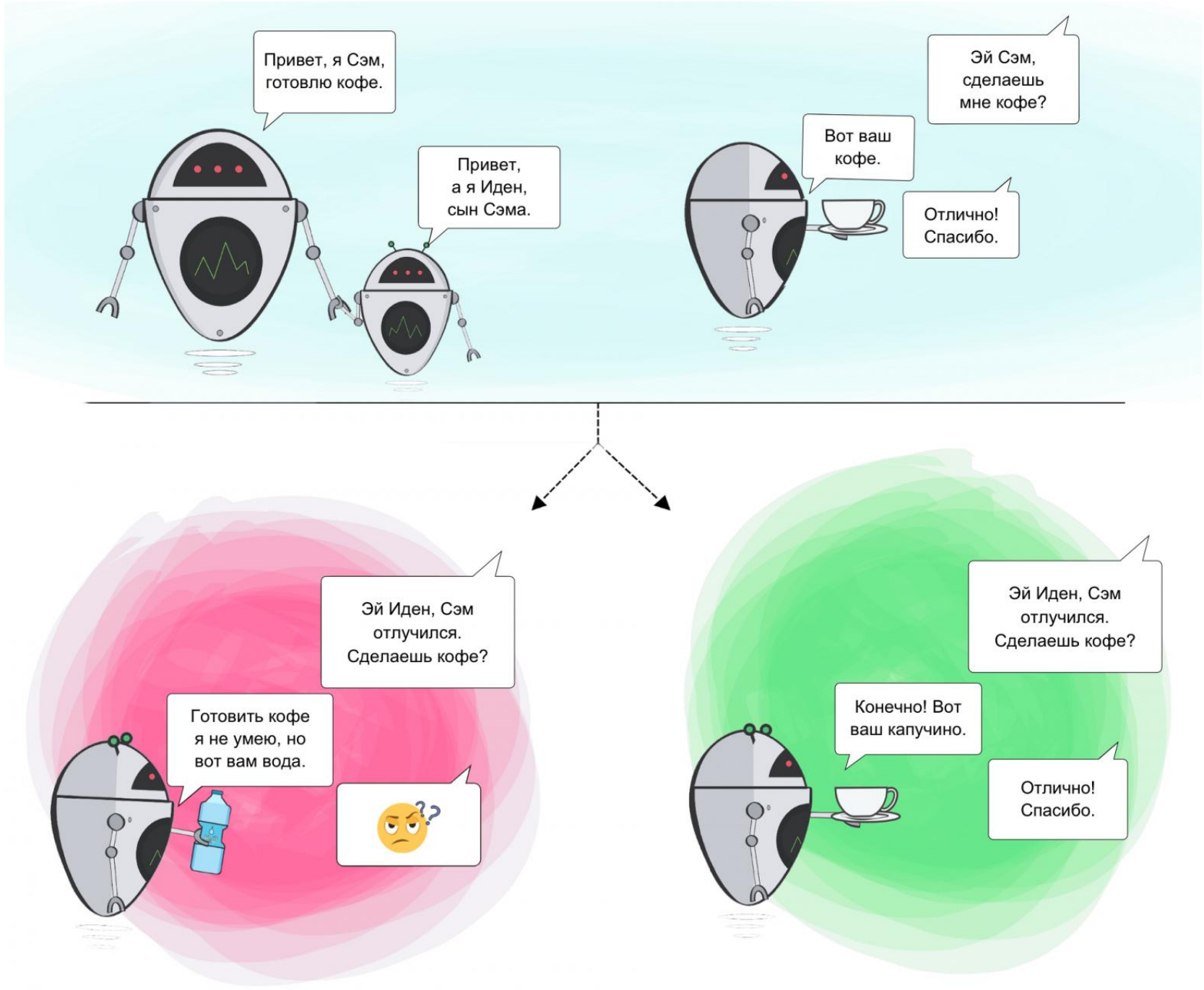
```
class Report
{
    public string Text { get; set; }

    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)...
}
```



LISKOV SUBSTITUTION PRINCIPLE

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


Привет, я Сэм, готовлю кофе.
Привет, а я Иден, сын Сэма.
Эй Сэм, сделаешь мне кофе?
Вот ваш кофе.
Отлично! Спасибо.
Эй Иден, Сэм отлучился. Сделаешь кофе?
Готовить кофе я не умею, но вот вам вода.
Эй Иден, Сэм отлучился. Сделаешь кофе?
Конечно! Вот ваш капучино.
Отлично! Спасибо.

```
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!");
    }
}
```

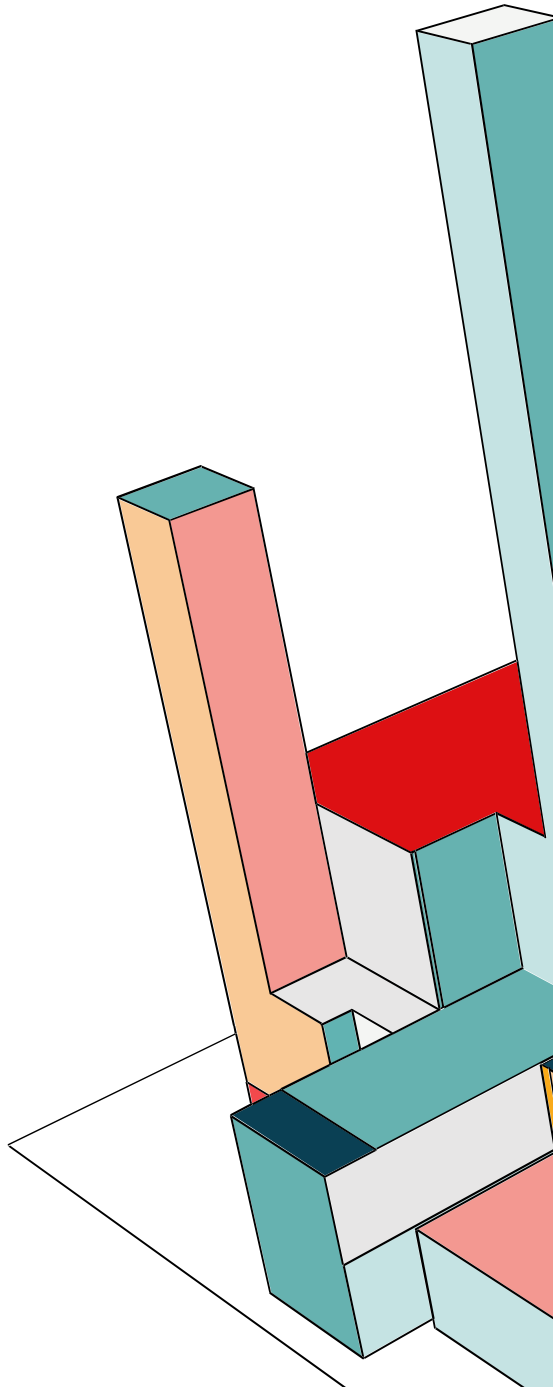
```
static void Main(string[] args)
{
    Rectangle rect = new Square();
    TestRectangleArea(rect);

    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!");
    }
}
```

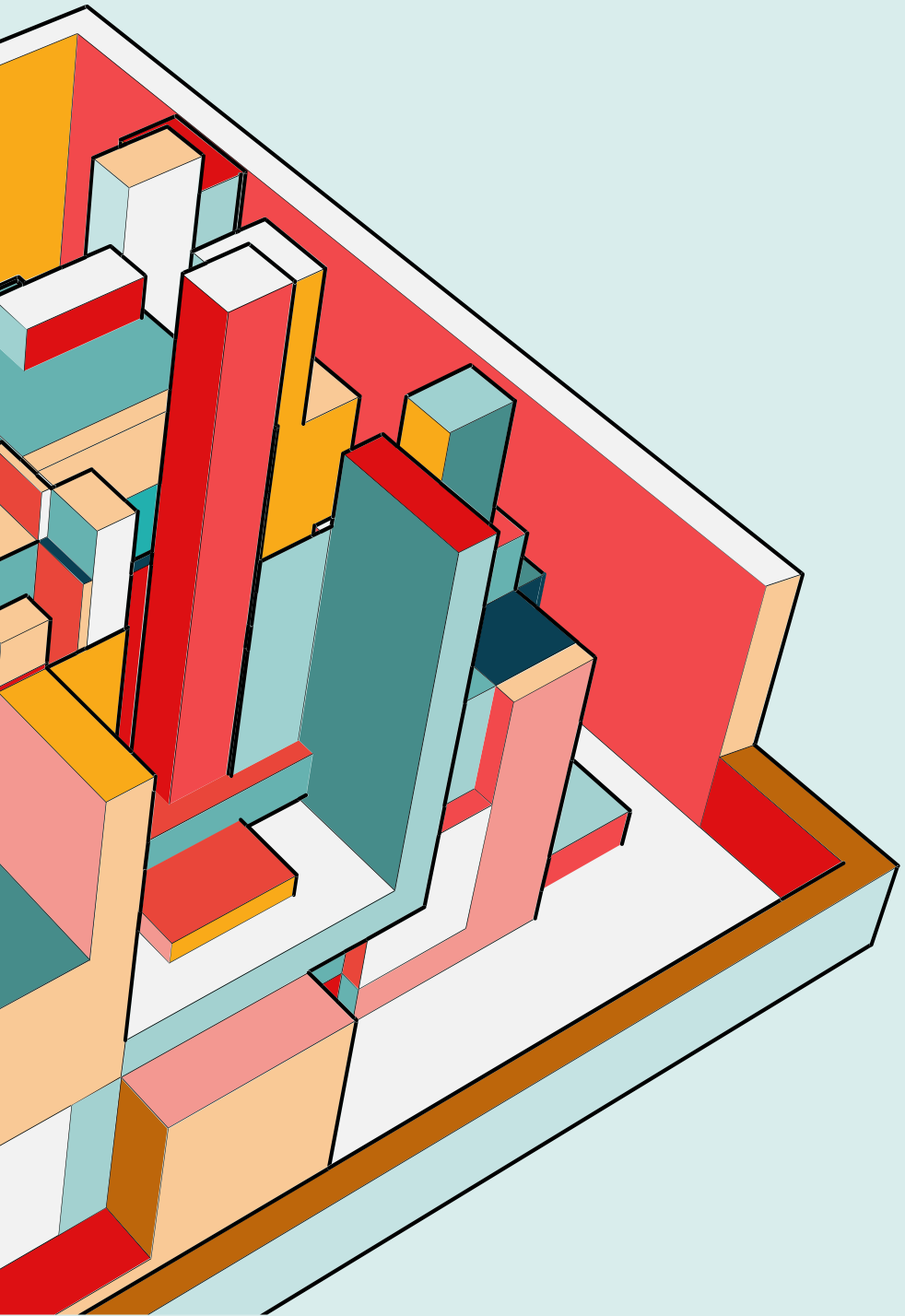
ОШИБКИ В НАСЛЕДОВАНИИ

Невозможно установить правильность модели, рассматриваемой изолированно

Как предусмотреть требования клиентов?

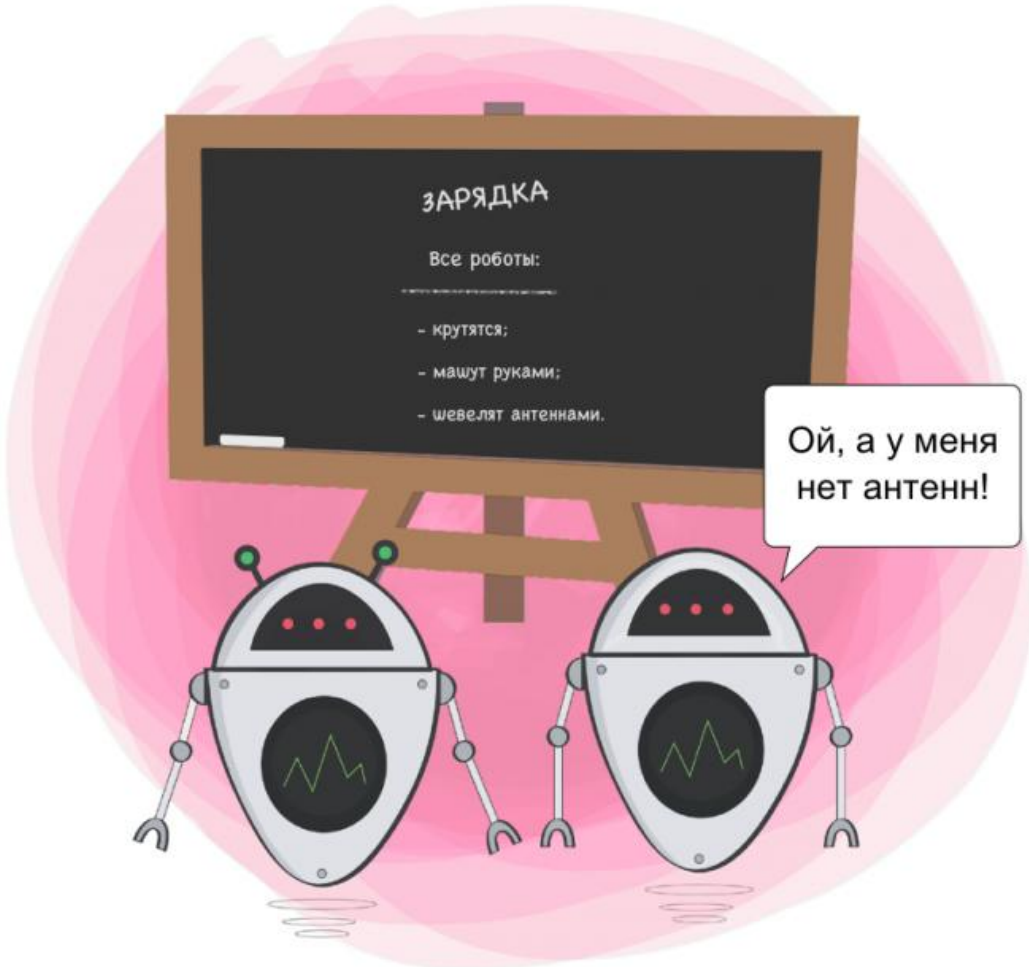
- ~~Экстрасенсорные способности~~ Опыт
- Программирование через тестирование
- Программирование по контракту

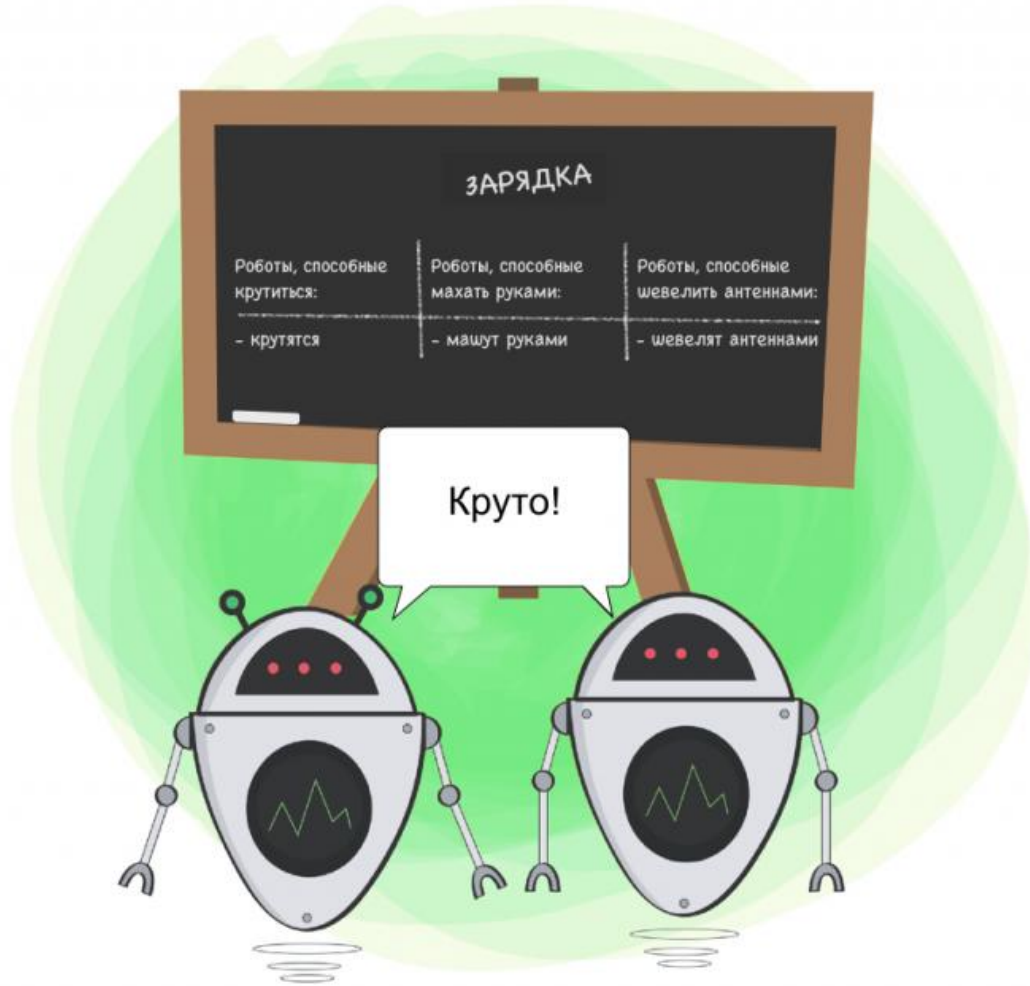




INTERFACE SEGREGATION PRINCIPLE

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


ЗАРЯДКА
Все роботы:
- крутятся;
- машут руками;
- шевелят антеннами.
Ой, а у меня нет антенн!


ЗАРЯДКА
Роботы, способные крутиться:
- крутятся
Роботы, способные махать руками:
- машут руками
Роботы, способные шевелить антеннами:
- шевелят антеннами
Круто!

```
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; }
}
```

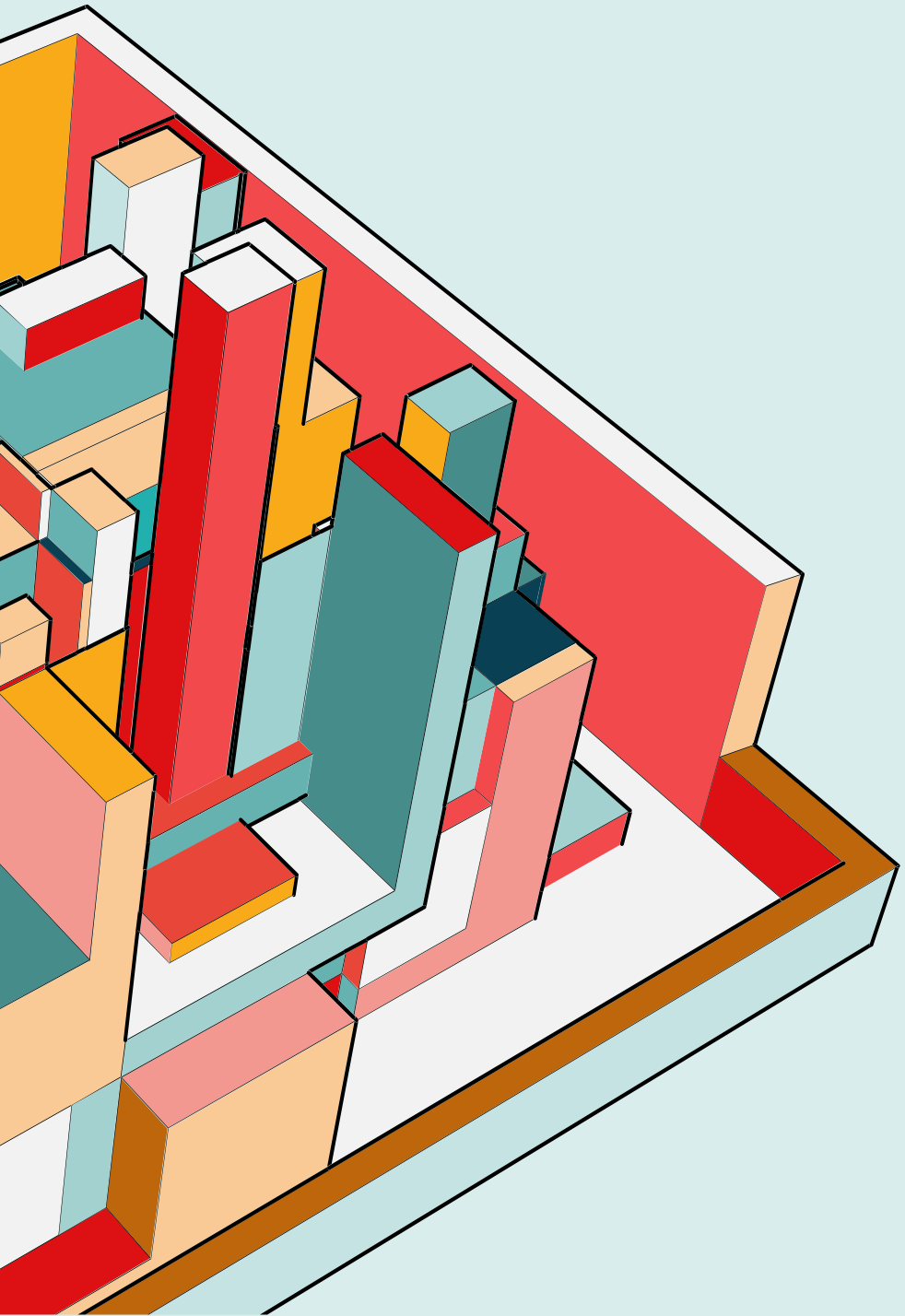
```
interface IMessage
{
    void Send();
    string Text { get; set; }
    string ToAddress { get; set; }
    string Subject { get; set; }
    string FromAddress { get; set; }

    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; }
}
```

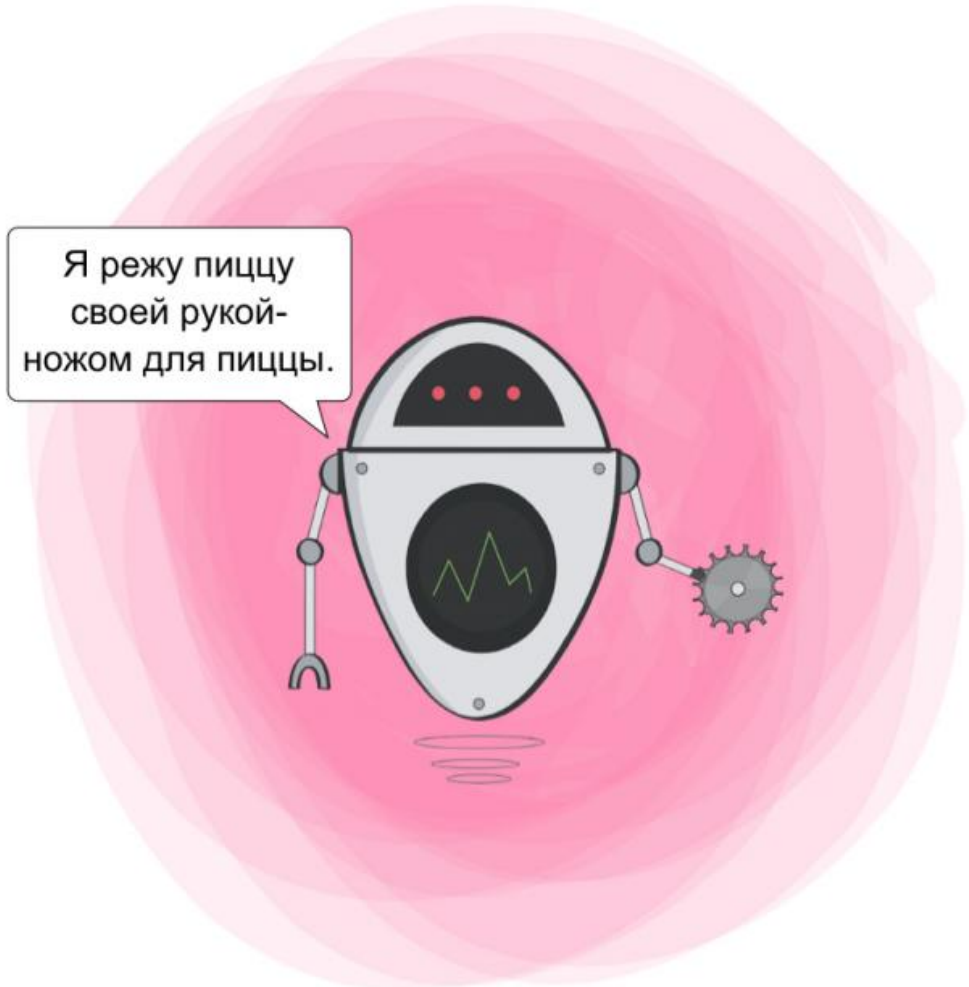
```
class VoiceMessage : IVoiceMessage
{
    public string ToAddress { get; set; }
    public string FromAddress { get; set; }

    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.

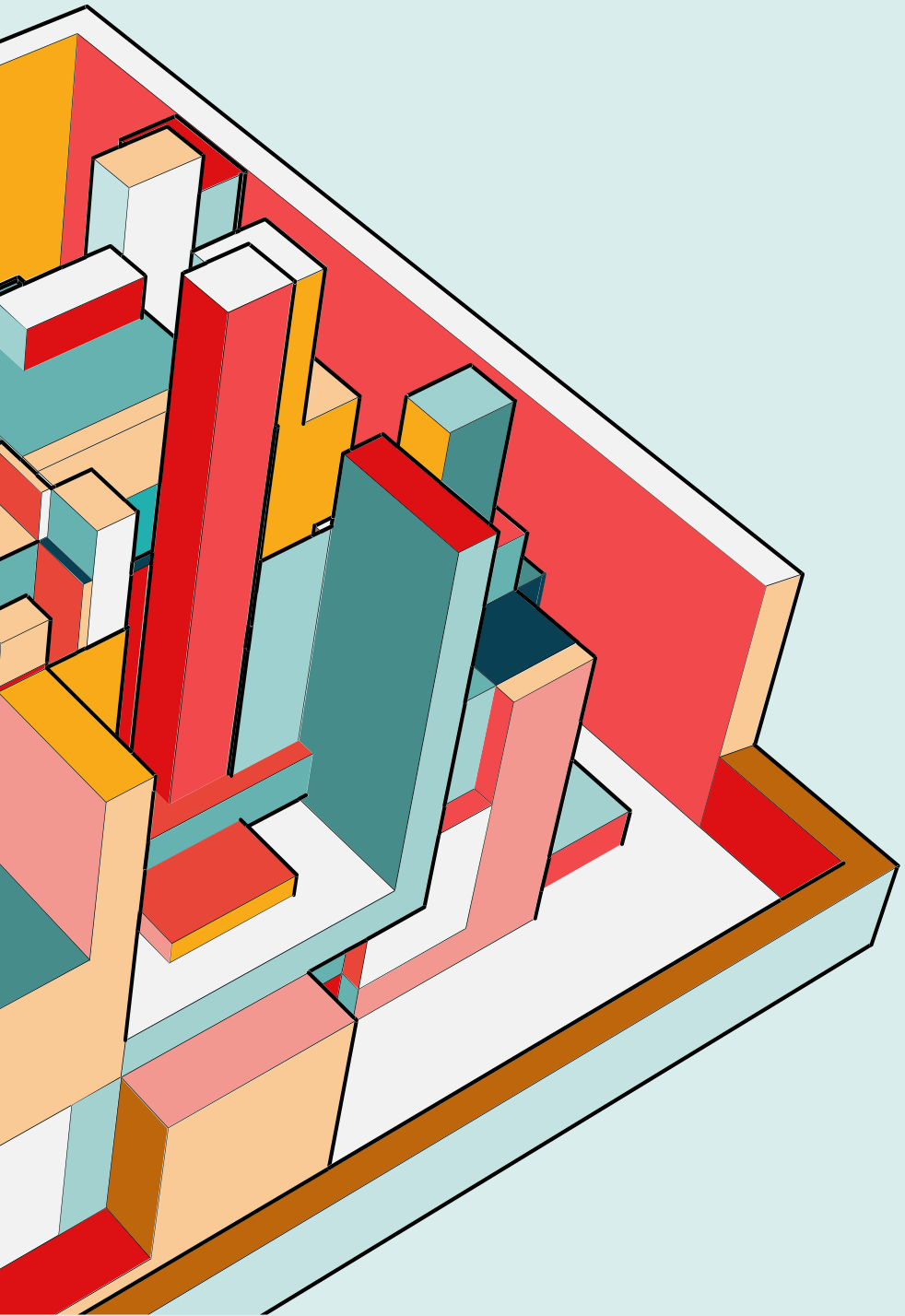

Я режу пиццу любым доступным инструментом.

```
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)...
}
```

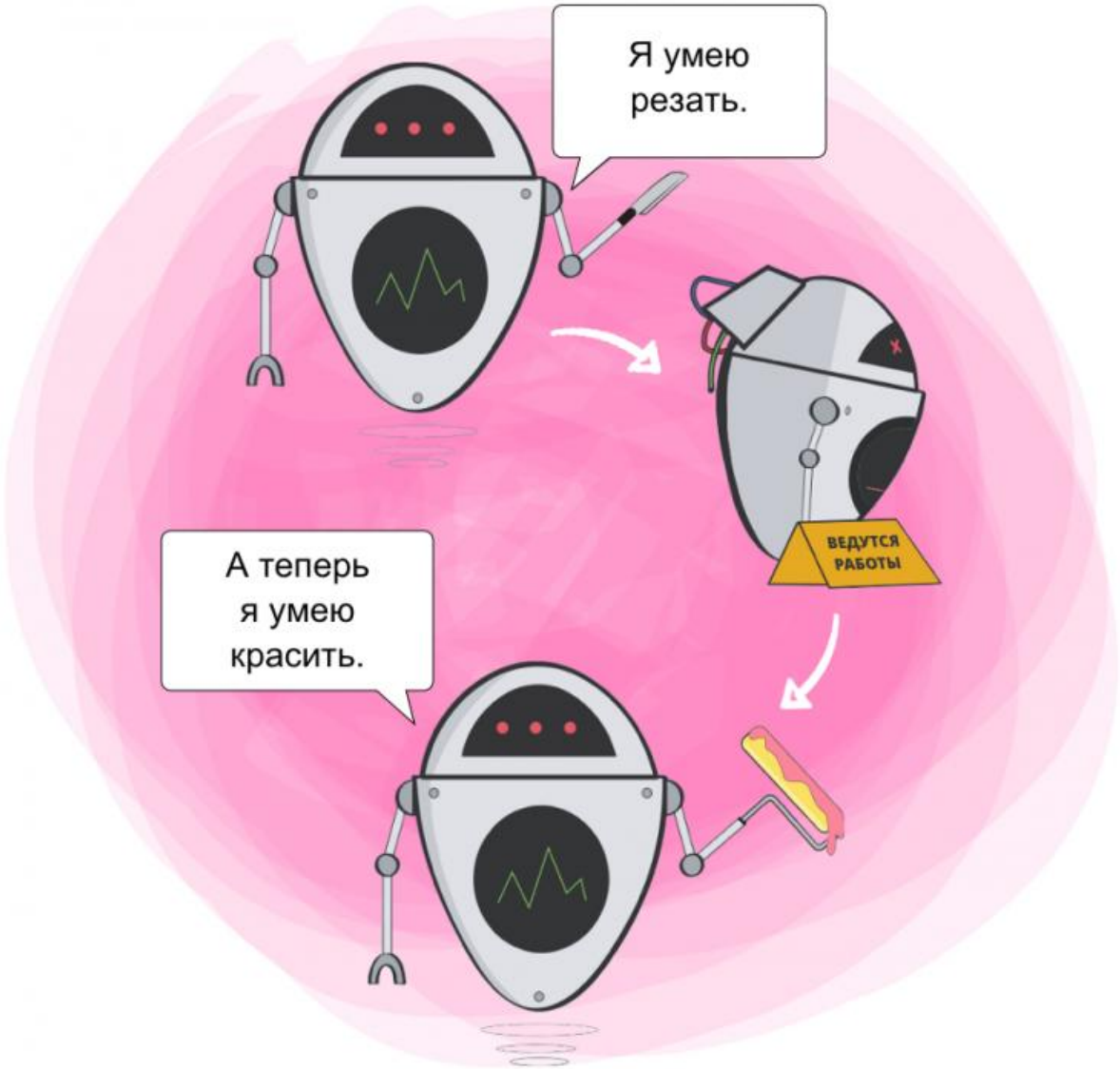
```
interface IPrinter
{
    void Print(string text);
}

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

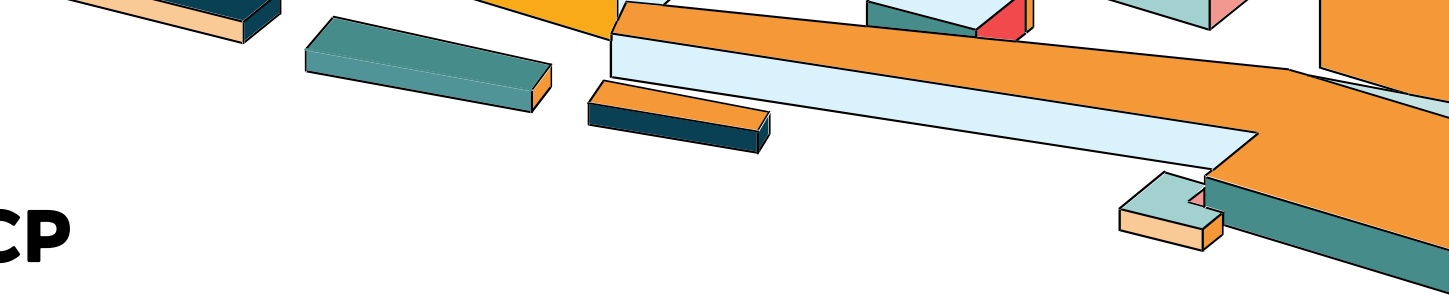
Объекты должны быть открыты для расширения, но закрыты для модификации


Я умею
резать.
ВЕДУТСЯ
РАБОТЫ
А теперь
я умею
красить.


Я умею
резать.
ВЕДУТСЯ
РАБОТЫ
А теперь
я умею резать
и красить.

```
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 OCP

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();
    }
}
```

Option 1.

```
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()...
}
```

Option 2.

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

abstract class MealBase...

class PotatoMeal : MealB
{
    protected override v

    protected override v

    protected override void Prepare()...
}

class SaladMeal ...
```

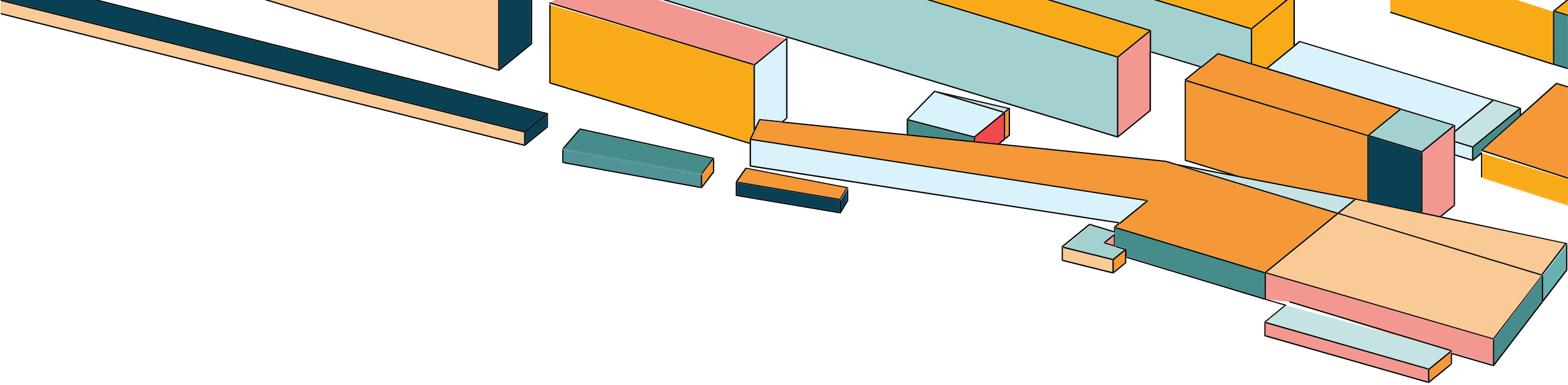
```
abstract class MealBase
{
    public void Make()
    {
        Prepare();
        Cook();
        FinalSteps();
    }
    protected abstract void Prepare();
    protected abstract void Cook();
    protected abstract void FinalSteps(
}
```

Option 1.

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

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

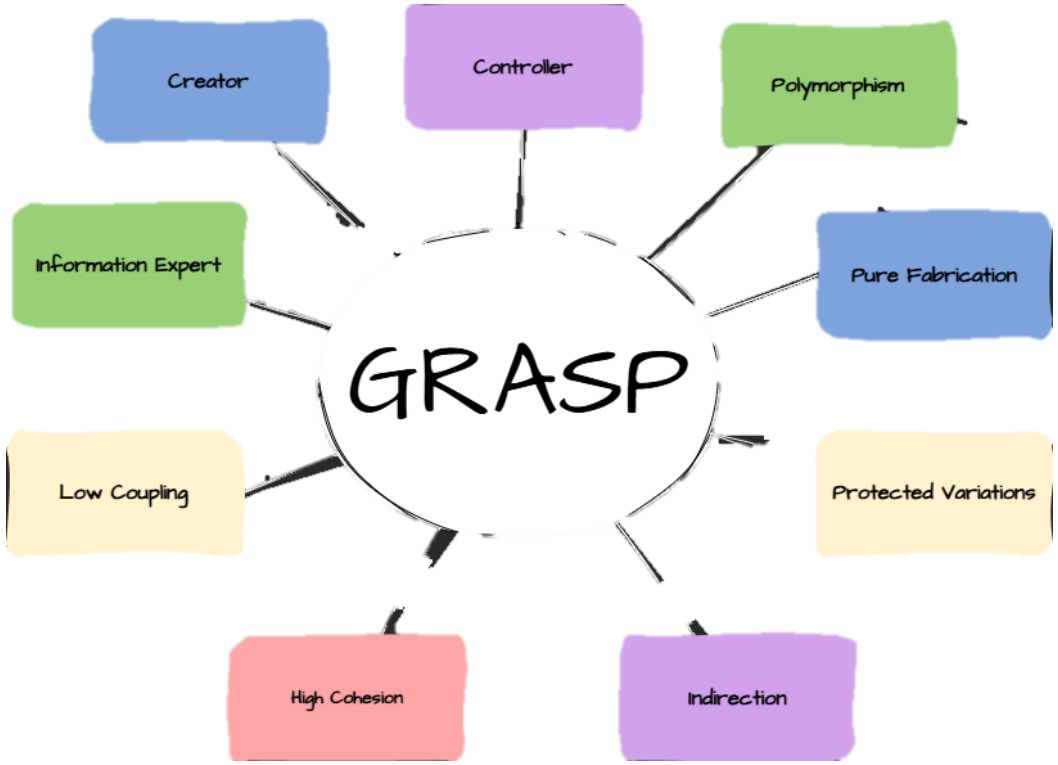
Option 2.

```
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



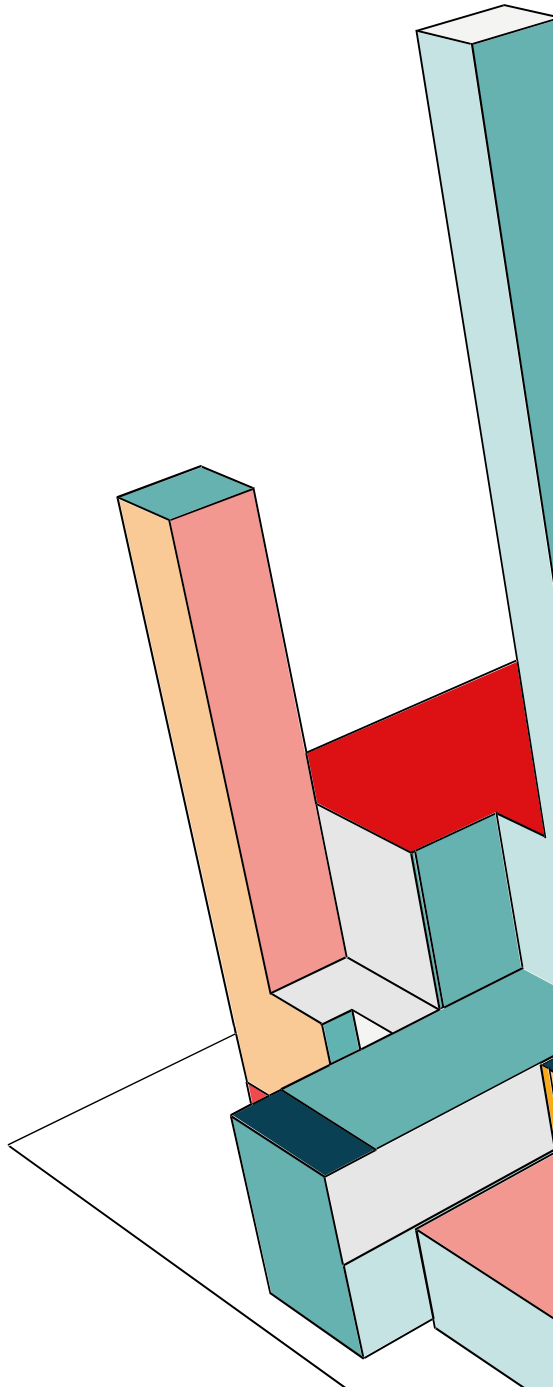
КЛАССИФИКАЦИЯ ШАБЛОНОВ GRASP

Основные шаблоны

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

Дополнительные шаблоны

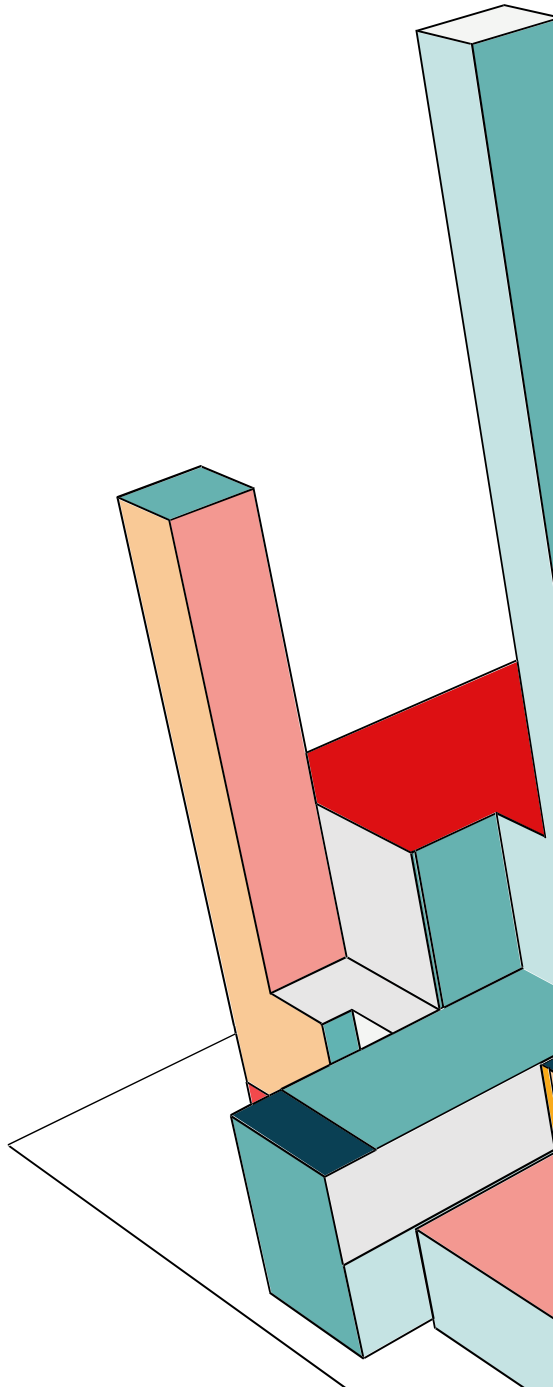
- Protected Variations (Устойчивость к изменениям)
- Polymorphism (Полиморфизм)
- Indirection (Перенаправление)
- Pure Fabrication (чистая выдумка, синтетика)



HIGH COHESION & LOW COUPLING

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

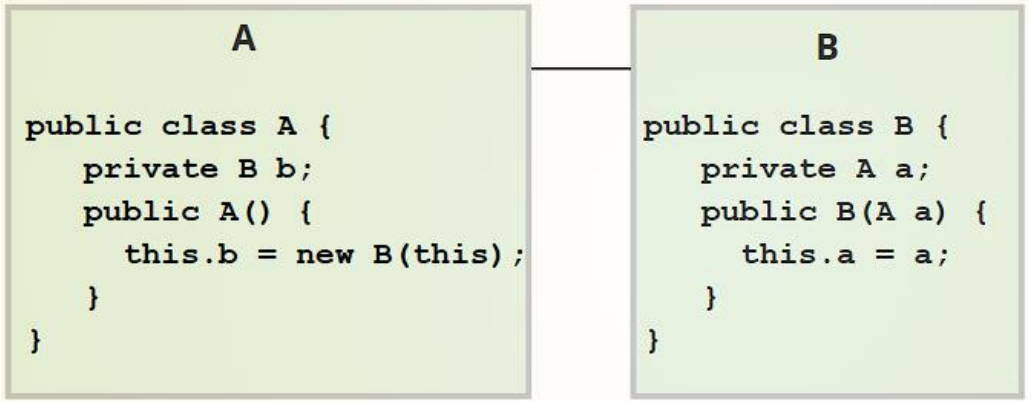
Эти два принципа рассматриваются совместно, и говорят о том, что классы целесообразно строить таким образом, чтобы они имели сильную внутреннюю связь и были слабо связаны между собой



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

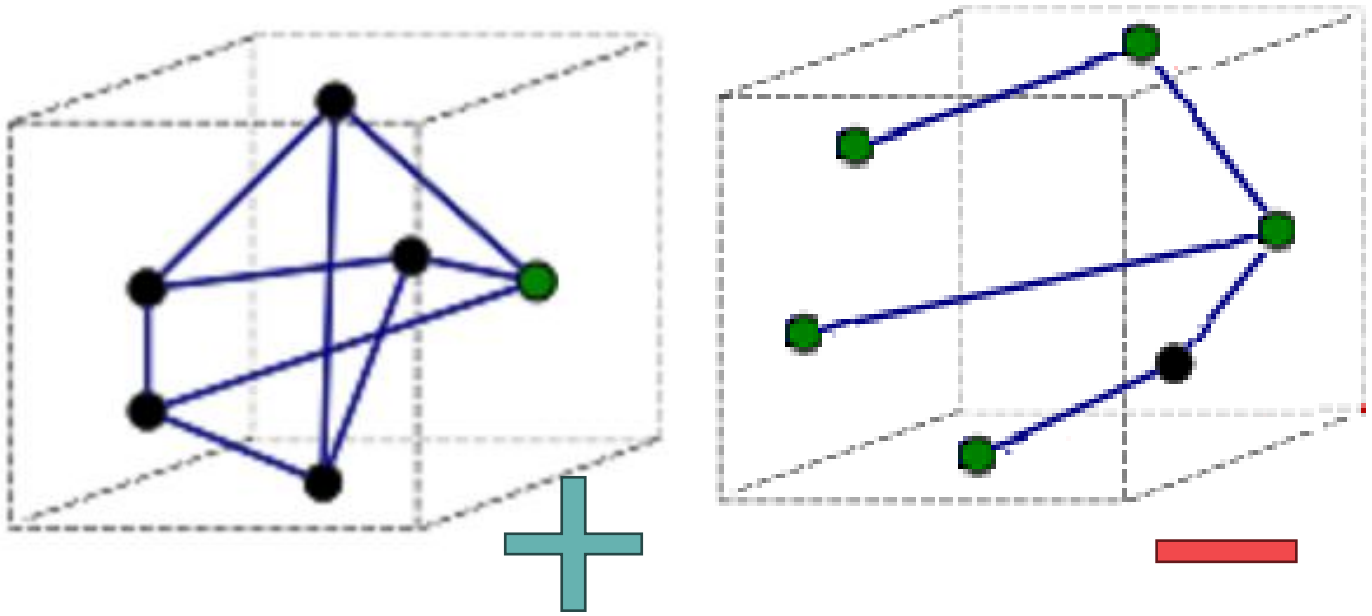


Если возвести Low Coupling в абсолют,
то вся функциональность будет реализована
в одном единственном классе

ВЫСОКАЯ СЦЕПЛЕННОСТЬ

Класс должны содержать сцепленную бизнес — логику!

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



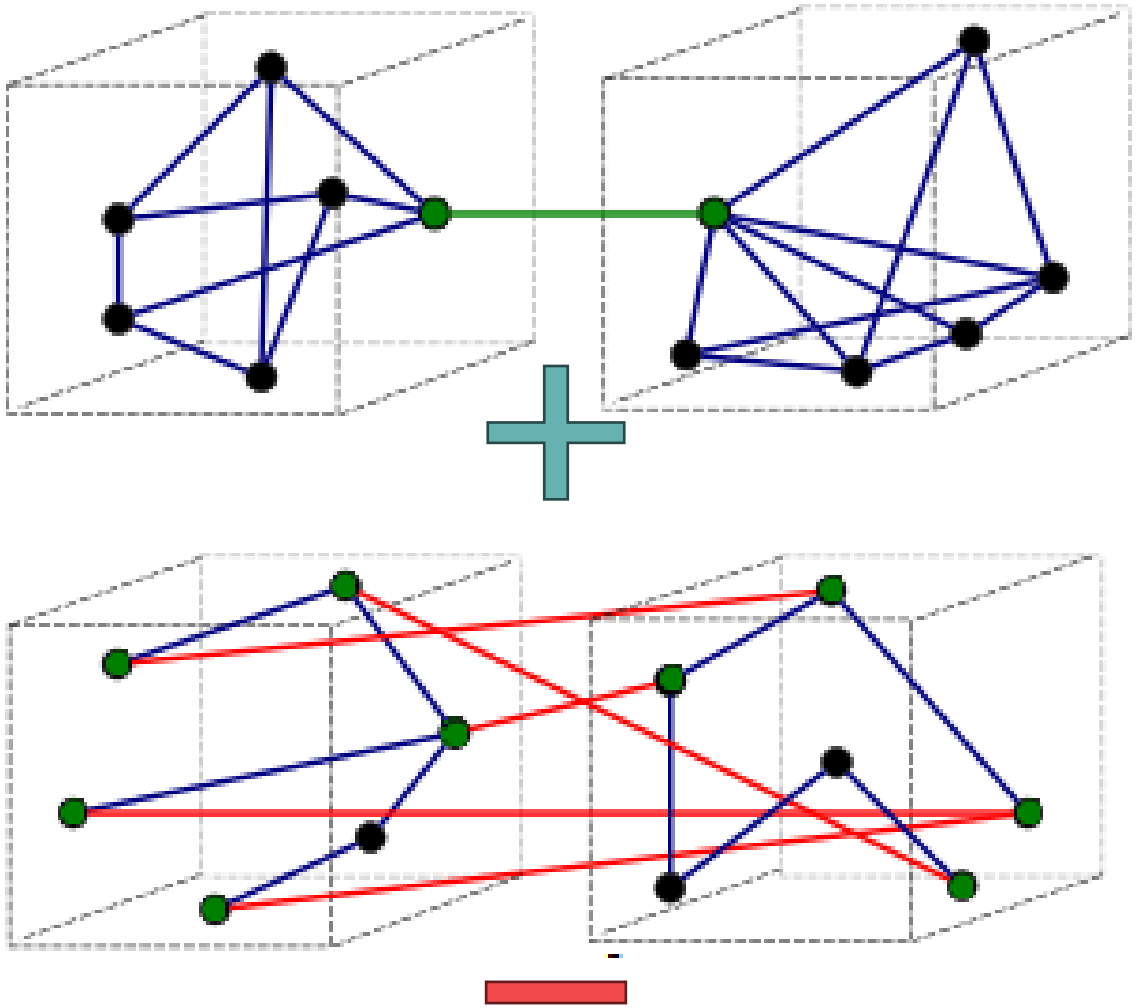
приватное свойство или метод

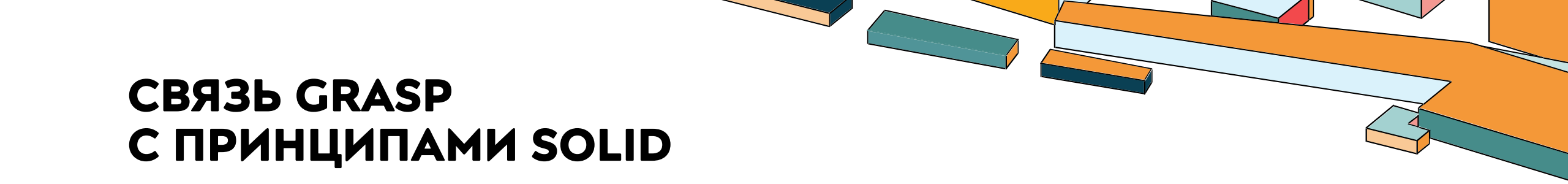
публичное свойство или метод

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

Высокая сцепленность ↔ *Единственная ответственность*

Низкое связывание ↔ *Единственная ответственность*

Низкое связывание ↔ *Разделение интерфейсов*

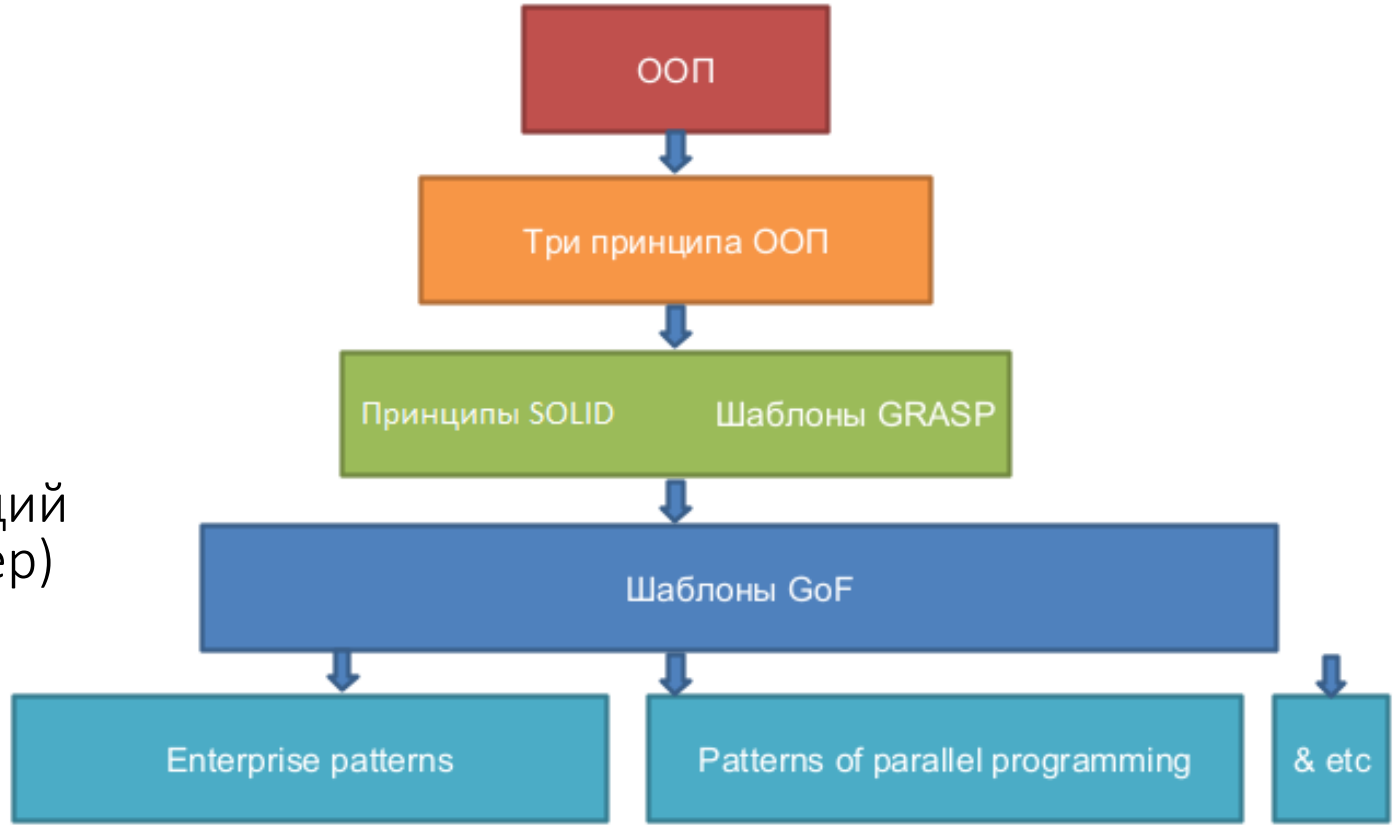
Устойчивость к изменениям ↔ *Открытость/закрытость*

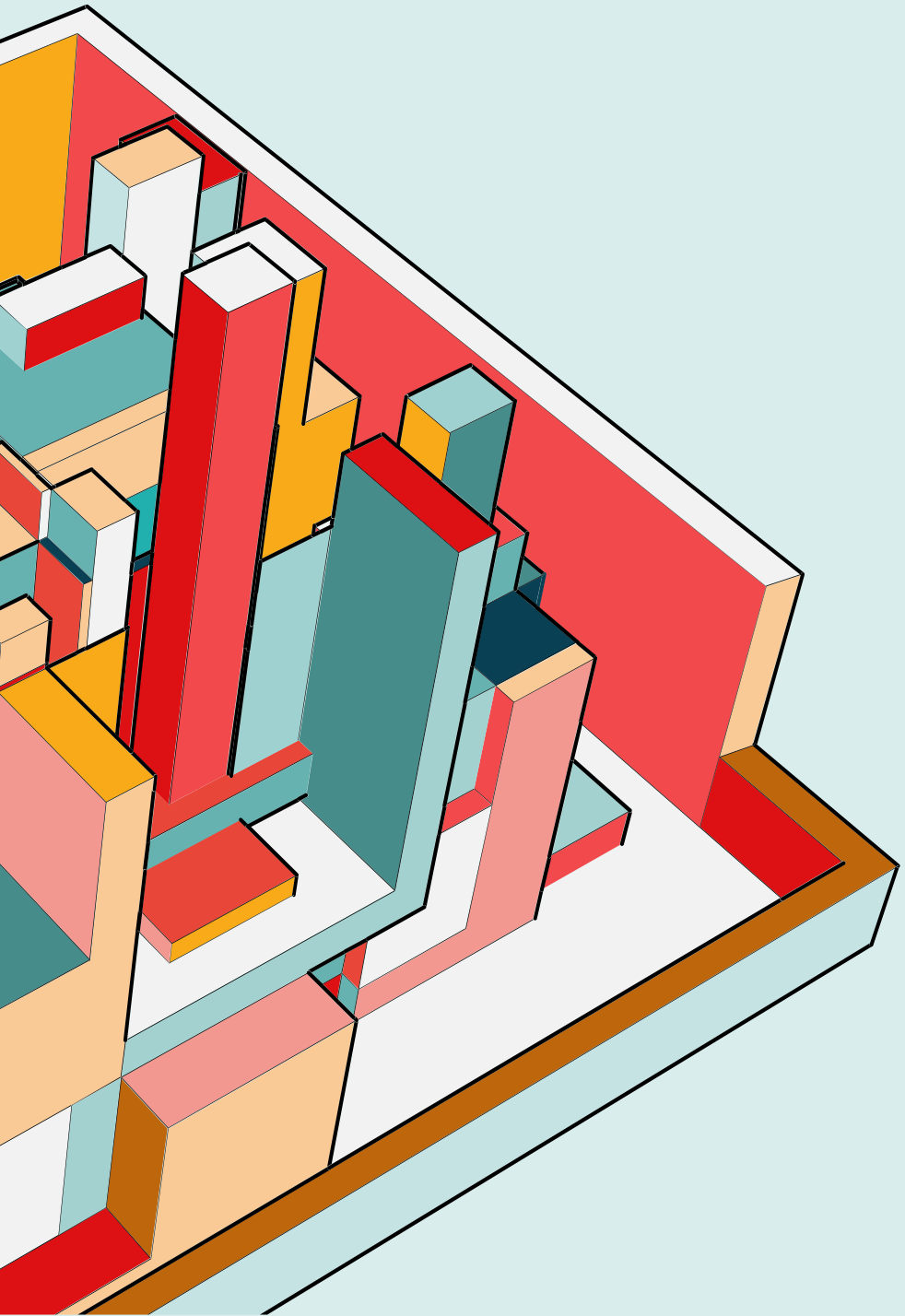
Устойчивость к изменениям ↔ *Инверсия зависимостей*

Полиморфизм ↔ *Принцип подстановки Барбары Лисков*

SOLID И GRASP

- Отдавать предпочтение "простым" классам, экспертам своей ответственности
- При усложнении определенная функциональность выносится в отдельную структуру
- Точки изменения защищаются с помощью абстракций
- Для решения задачи можно синтезировать класс, отсутствующий в предметной области (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.

```java
// Биатлонист с кучей ненужных методов
class Biathlete {
    private String name;
    private int age;
    private String skiBrand;  // Пока не нужно
    private String rifleModel; // Пока не нужно

    void shoot() { /* ... */ }
    void ski() { /* ... */ }
    void cleanRifle() {} // Не используется
    void adjustSkiBindings() {} // Не используется
}
```

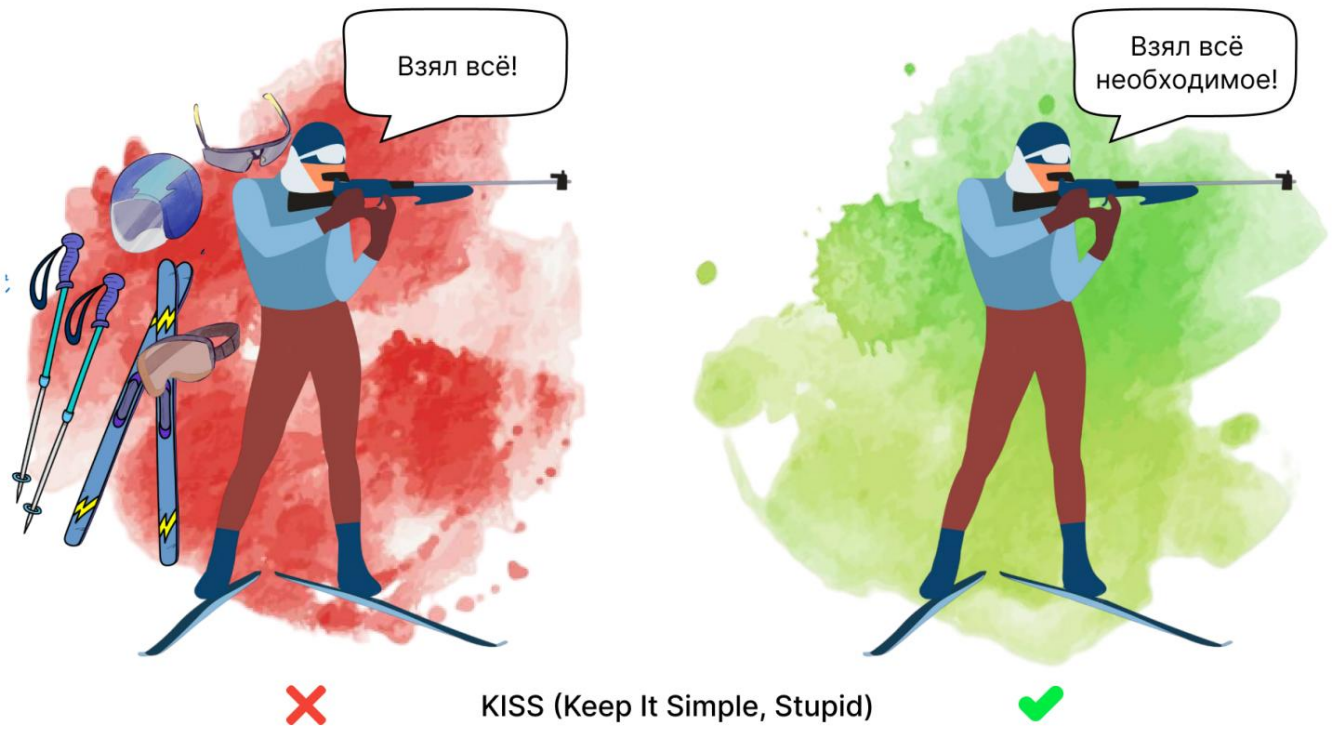
```java
// Только основная функциональность
class Biathlete {
    private String name;
    private int age;

    void shoot() {
        System.out.println("Выстрел");
    }

    void ski() {
        System.out.println("Лыжный ход");
    }
}
```

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 требуется.



```java
public class OvercomplicatedShootingScore {
    public int calculateScore(boolean[] hits, int seriesNumber, int windSpeed, boole
        int score = 0;
        double windFactor = 1.0;

        // Учёт ветра (избыточная сложность для примера)
        if (windSpeed > 5) {
            windFactor = 0.9;
        } else if (windSpeed > 10) {
            windFactor = 0.8;
        } else if (windSpeed > 15) {
            windFactor = 0.7;
        }

        // Учёт положения (стоя/лёжа)
        if (isStanding) {
            windFactor *= 0.95;
        }

        // Подсчёт попаданий с "корректировками"
        for (boolean hit : hits) {
            if (hit) {
                score += 10 * windFactor;
            } else {
                score -= 5 * (1.0 / seriesNumber);
            }
        }

        // Дополнительная "награда" за серию
        if (seriesNumber > 3) {
            score += 2 * seriesNumber;
        }

        return (int) Math.round(score);
    }
}
```

```java
public class SimpleShootingScore {
    /**
     * Считает штрафные круги по результатам стрельбы.
     * @param hits Массив попаданий (true = попадание, false = промах).
     * @return Количество штрафных кругов.
     */
    public int calculatePenalty(boolean[] hits) {
        int misses = 0;
        for (boolean hit : hits) {
            if (!hit) misses++;
        }
        return misses; // 1 промах = 1 штрафной круг
    }
}
```

DRY

Don’t Repeat Yourself / Не повторяйтесь

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

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

```java
public class Biathlete {

    public void trainShooting() {
        System.out.println("Занять позицию");
        System.out.println("Прицелиться");
        System.out.println("Сделать 5 выстрелов");
    }

    public void trainCombined() {
        System.out.println("Пробежать 5 км");
        // Дублирование
        System.out.println("Занять позицию");
        System.out.println("Прицелиться");
        System.out.println("Сделать 5 выстрелов");
    }

    public void trainInterval() {
        System.out.println("Спринт 1 км");
        // Дублирование
        System.out.println("Занять позицию");
        System.out.println("Прицелиться");
        System.out.println("Сделать 5 выстрелов");
    }
}
```

```java
public class Biathlete {

    private void shoot() {
        System.out.println("Занять позицию");
        System.out.println("Прицелиться");
        System.out.println("Сделать 5 выстрелов");
    }

    public void trainShooting() {
        shoot();
    }

    public void trainCombined() {
        System.out.println("Пробежать 5 км");
        shoot();
    }

    public void trainInterval() {
        System.out.println("Спринт 1 км");
        shoot();
    }
}
```

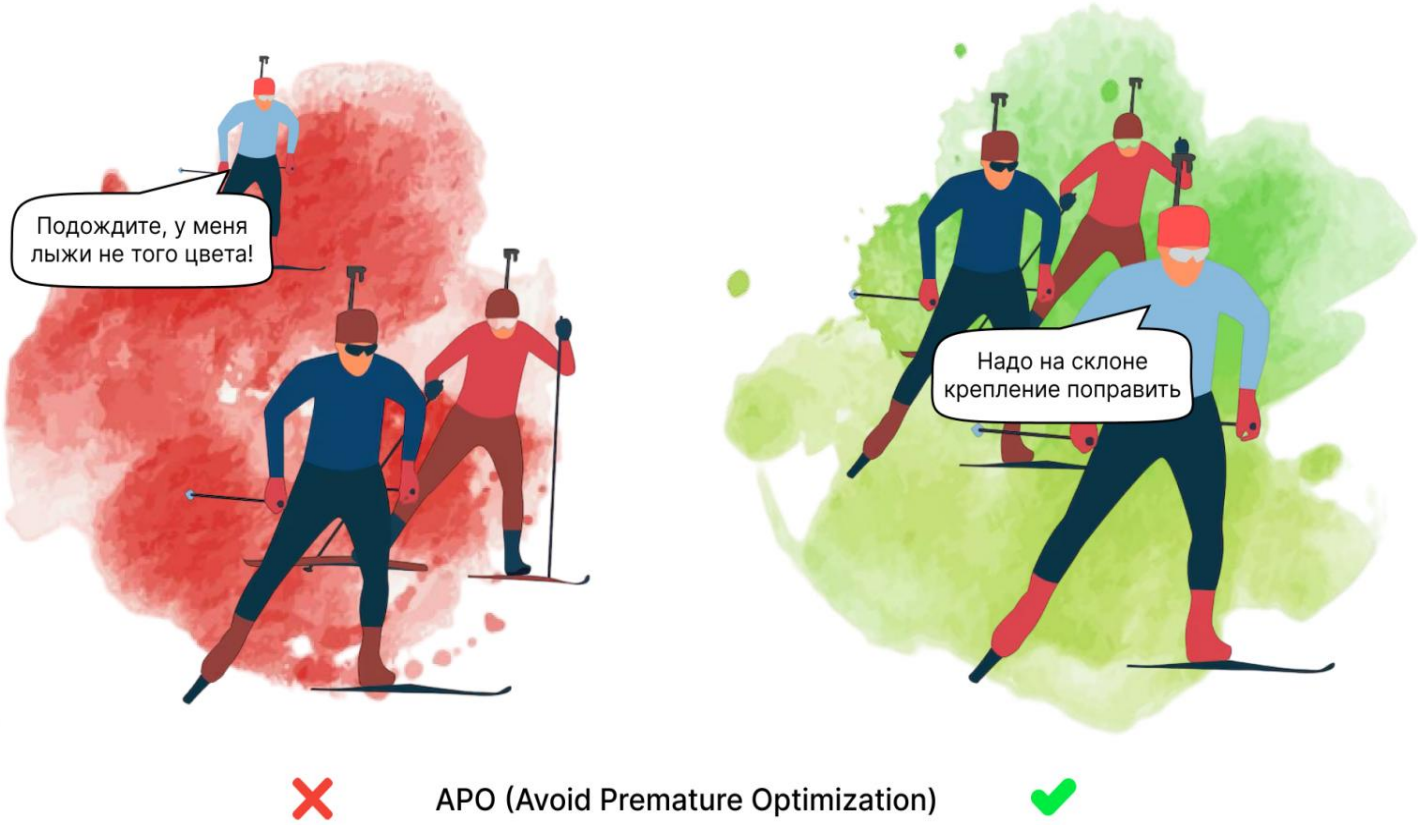
APO

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.



```java
public class Biathlete {

    private String name;
    private int age;

    private StringBuilder trainingLog = new StringBuilder();

    public Biathlete(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void train() {
        // Сложная логика с кэшированием и оптимизацией строк
        trainingLog.append("Тренировка началась\n");
        trainingLog.append("Выполняются упражнения\n");
        trainingLog.append("Тренировка завершена\n");

        System.out.println(trainingLog.toString());
    }

    public void compete() {
        // Аналогичная сложная оптимизация
        trainingLog.append("Соревнование началось\n");
        trainingLog.append("Участие в гонке\n");
        trainingLog.append("Соревнование завершено\n");

        System.out.println(trainingLog.toString());
    }
}
```

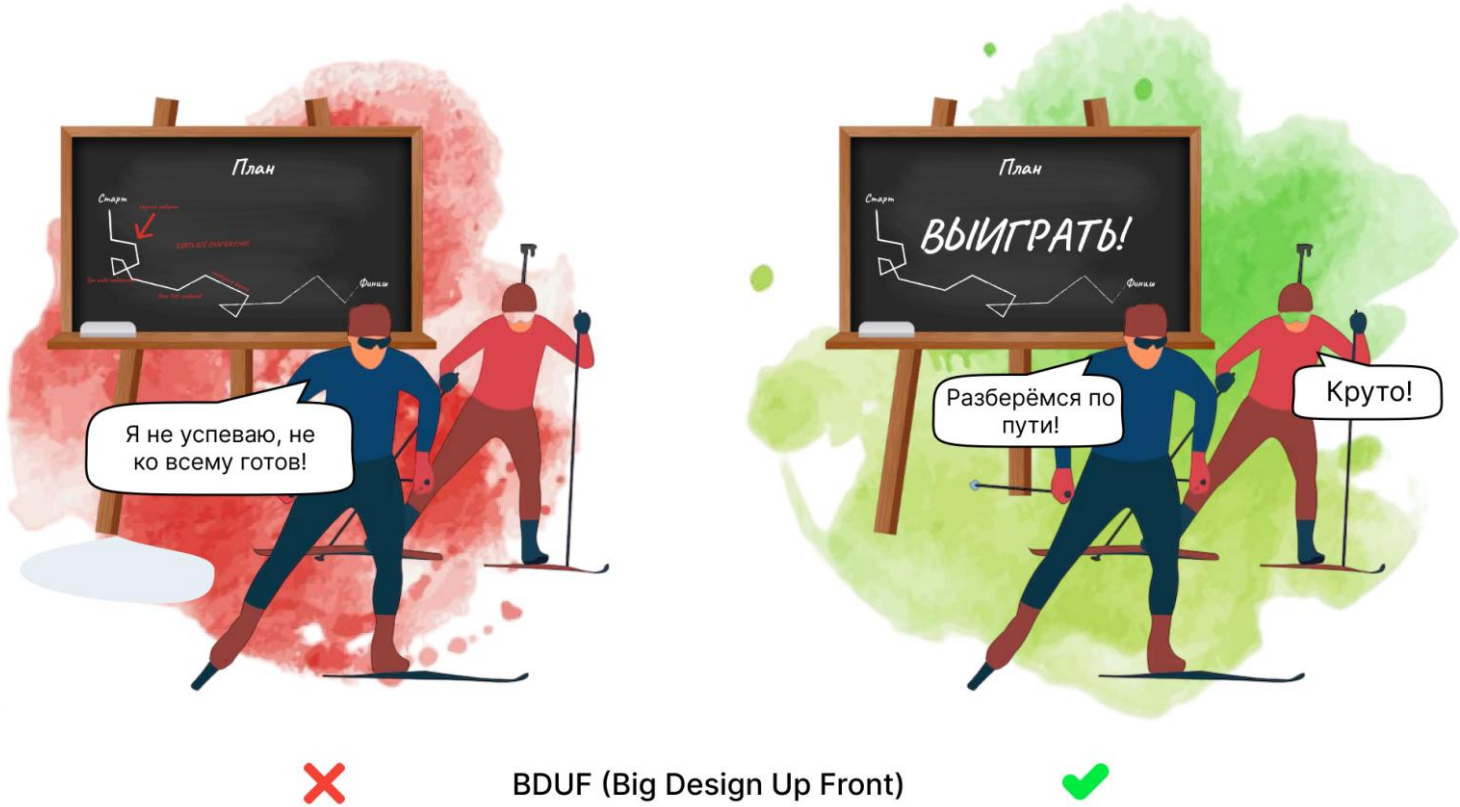
```java
public class Biathlete {

    private String name;
    private int age;

    public Biathlete(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void train() {
        System.out.println(name + " тренируется");
    }

    public void compete() {
        System.out.println(name + " участвует в соревнованиях");
    }
}
```

BDUF

Big Design Up Front / Глобальное проектирование прежде всего

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

Принцип предупреждает, что чрезмерное планирование на ранних этапах может привести к ненужной сложности и плохой адаптации к изменениям.



BDUF (Big Design Up Front)

```java
public class CompetitionSystem {

    private List<Athlete> athletes;
    private List<Judge> judges;
    private List<Reporter> reporters;
    private List<Fan> fans;
    private List<MedicalTeam> medics;
    private LiveStream stream;
    private StatisticSystem stats;
    private VIPLounge vipLounge;

    public void startCompetition() {
        authenticateAllUsers();
        initMedicalProtocols();
        checkVIPAccess();
        launchLiveStream();
        notifyPress();
        beginRace();
    }

    private void authenticateAllUsers() {
        // Сложная логика аутентификации
    }

    private void initMedicalProtocols() {
        // Подключение медперсонала
    }

    private void checkVIPAccess() {
        // Работа с VIP-гостями
    }

    private void launchLiveStream() {
        // Старт видеотрансляции
    }

    private void notifyPress() {
        // Информирование репортёров
```

```java
public class CompetitionSystem {

    private List<Athlete> athletes;

    public CompetitionSystem(List<Athlete> athletes) {
        this.athletes = athletes;
    }

    public void startRace() {
        System.out.println("Гонка началась!");
        for (Athlete athlete : athletes) {
            athlete.run();
        }
    }
}
```

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.

БРИТВА ОККАМА

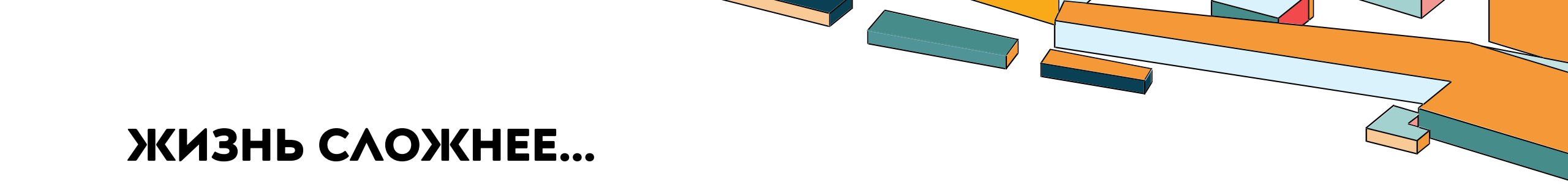
Уильям Оккам: Не следует множить сущее без необходимости

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

```java
public interface Trainable {
    void executeTrainingRoutine();
}

public interface Competitor {
    void executeCompetitionRoutine();
}

public class TrainingService {
    public void prepare() {
        System.out.println("Подготовка к тренировке");
    }
}

public class CompetitionService {
    public void prepare() {
        System.out.println("Подготовка к соревнованию");
    }
}
```

```java
public class Biathlete implements Trainable, Competitor {

    private TrainingService trainingService = new TrainingService();
    private CompetitionService competitionService = new CompetitionService();

    @Override
    public void executeTrainingRoutine() {
        trainingService.prepare();
        System.out.println("Биатлонист тренируется");
    }

    @Override
    public void executeCompetitionRoutine() {
        competitionService.prepare();
        System.out.println("Биатлонист участвует в соревнованиях");
    }
}
```

```java
public class Biathlete {

    private String name;

    public Biathlete(String name) {
        this.name = name;
    }

    public void train() {
        System.out.println(name + " тренируется");
    }

    public void compete() {
        System.out.println(name + " участвует в соревнованиях");
    }
}
```

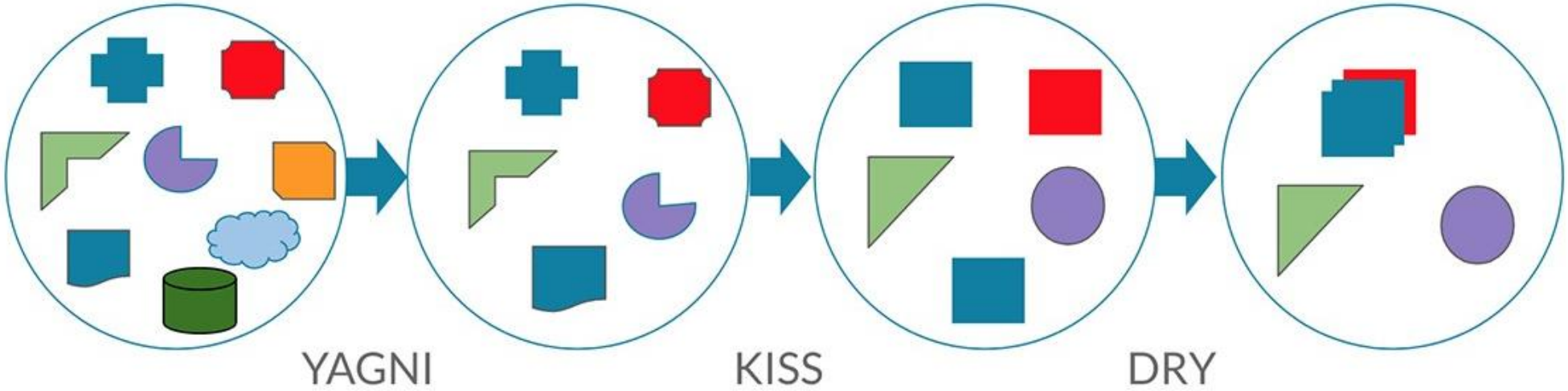


ЖИЗНЬ СЛОЖНЕЕ...

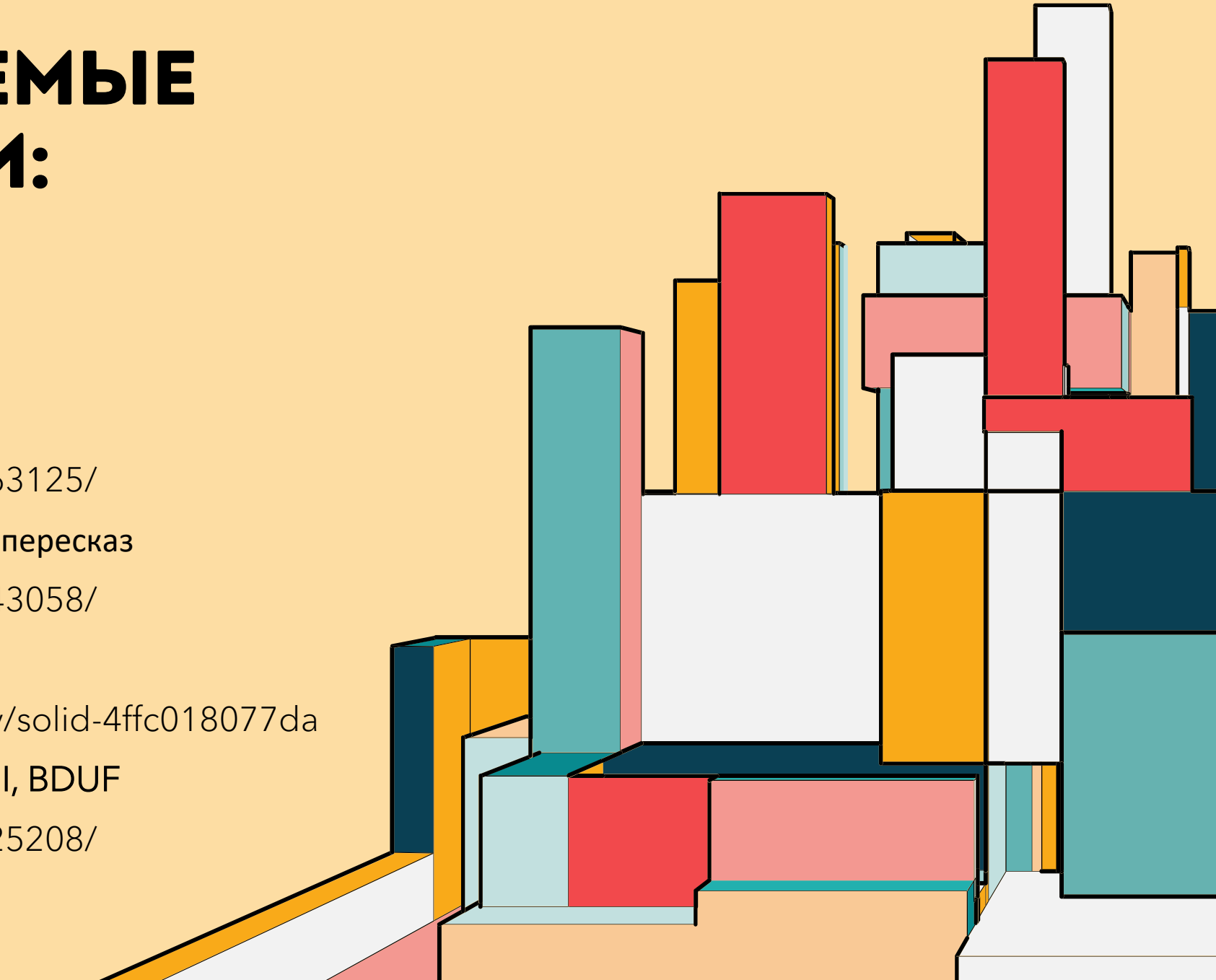
Бывает, что SOLID противоречит ягни, драй и кису.

Иногда они успешно сосуществуют, но есть достаточно случаев когда надо выбирать.

Лично я агитирую за выбор соблюдения SOLID.



ИСПОЛЬЗУЕМЫЕ ИСТОЧНИКИ:



📕 Всё ООП в одной статье

https://habr.com/ru/articles/463125/

📗 Чистая архитектура: краткий пересказ

https://habr.com/ru/articles/443058/

📕 Принципы SOLID

https://medium.com/webbdev/solid-4ffc018077da

📗 Бритва Оккама, KISS, YAGNI, BDUF

https://habr.com/ru/articles/925208/

СПАСИБО!

Виденин Сергей

@videninserg