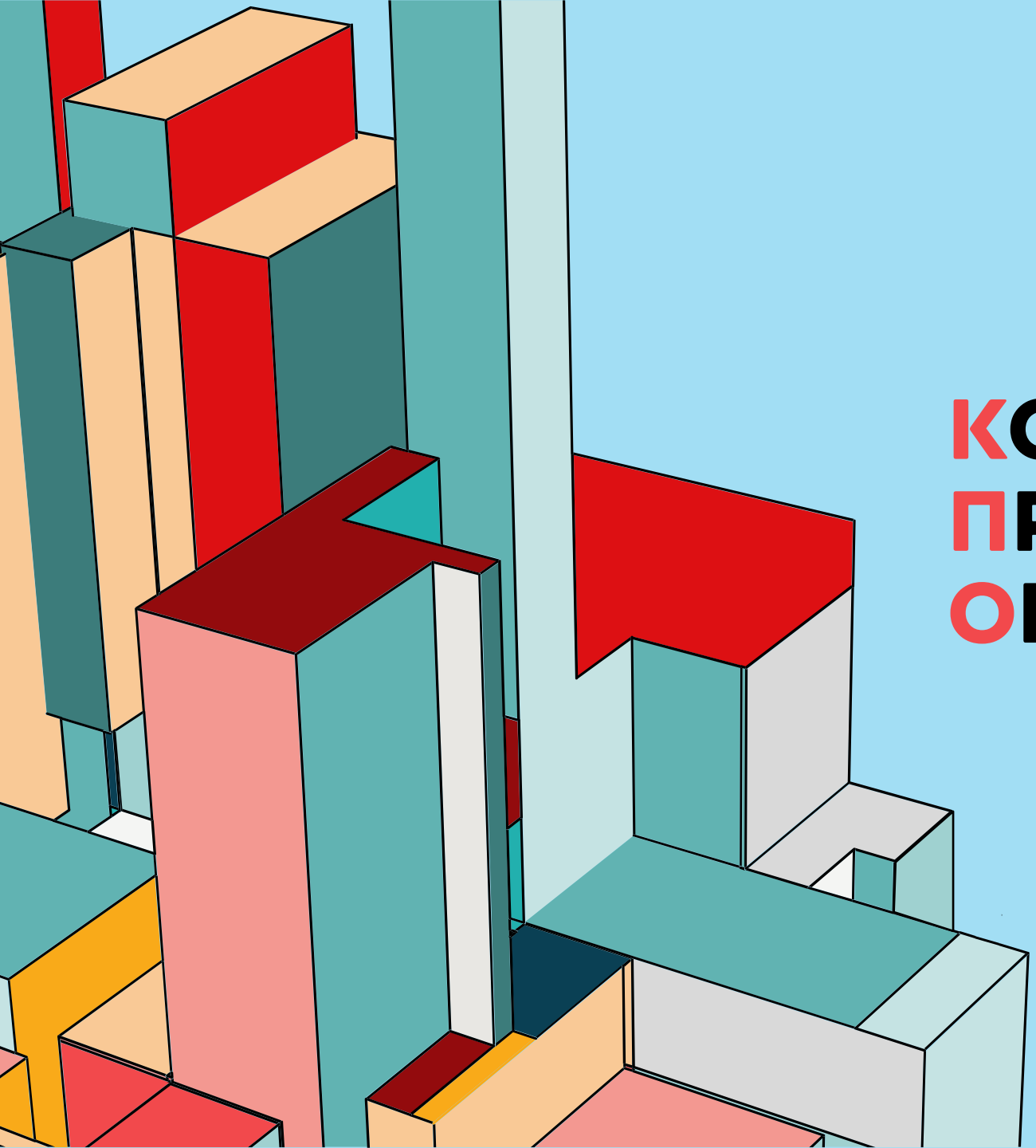
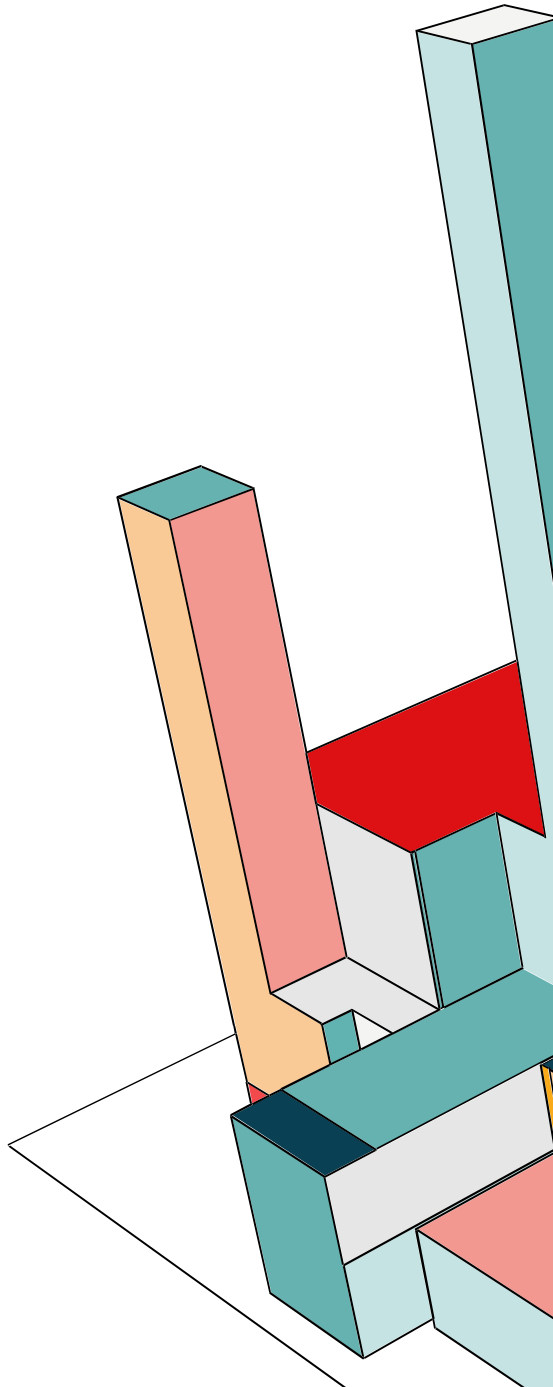
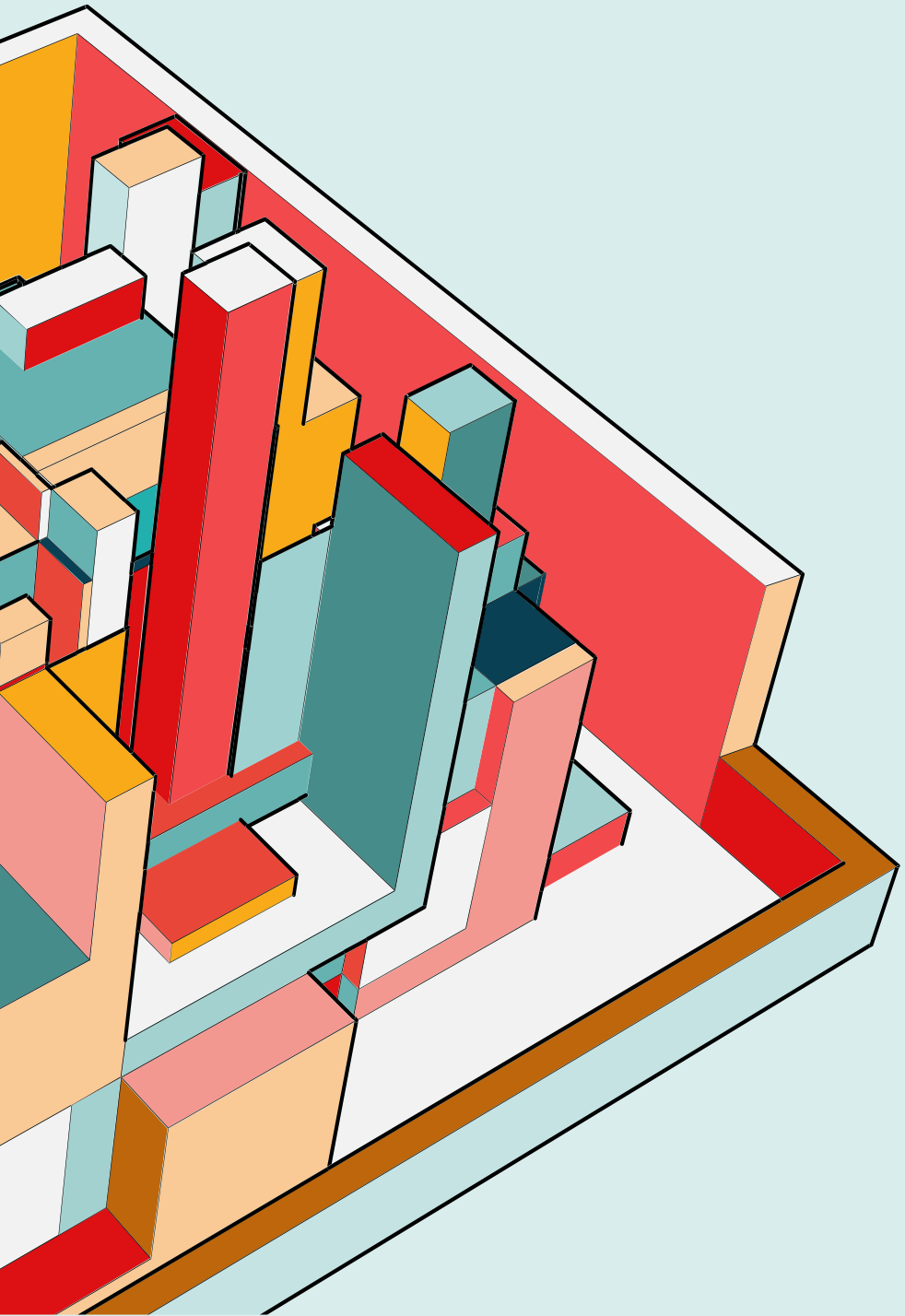


КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ПЛАН ЛЕКЦИИ № 2

- Вспомним DIP
- IoC, DI, DI-Container
- Жизненный цикл зависимостей
- Поговорим о важности тестирования
- Юнит-тестирование





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.

```
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);
    }
}
```

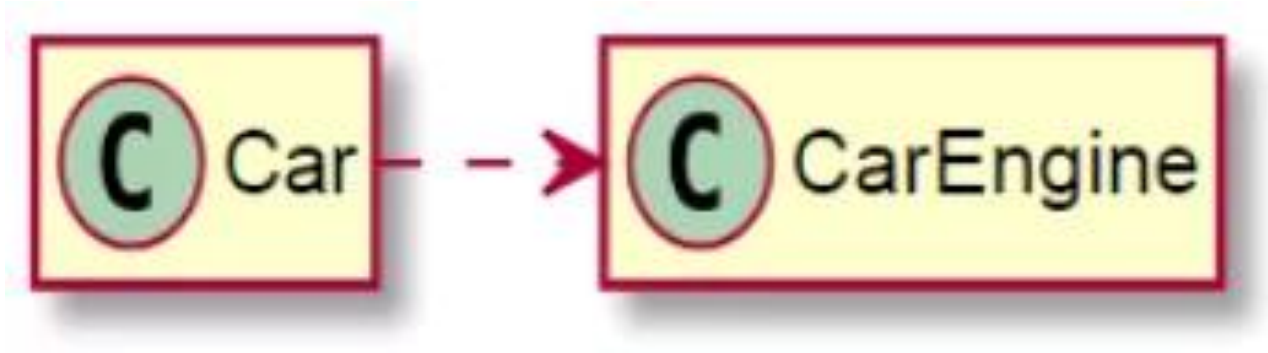
```
class Book
{
    public string Text { get; set; }
    public HtmlPrinter Printer { get; set; }

    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)...
}
```

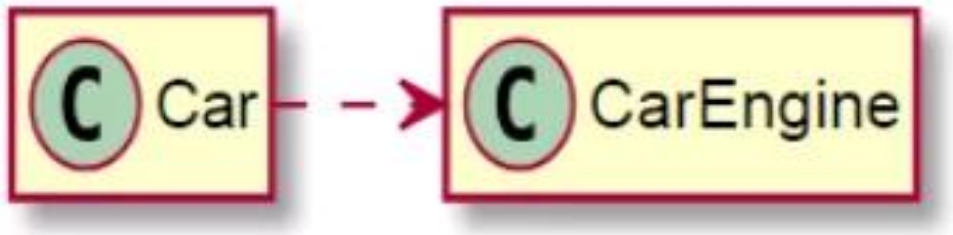
```
static void Main(string[] args)
{
    Book book = new Book(new ConsolePrinter());
    book.Print();

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

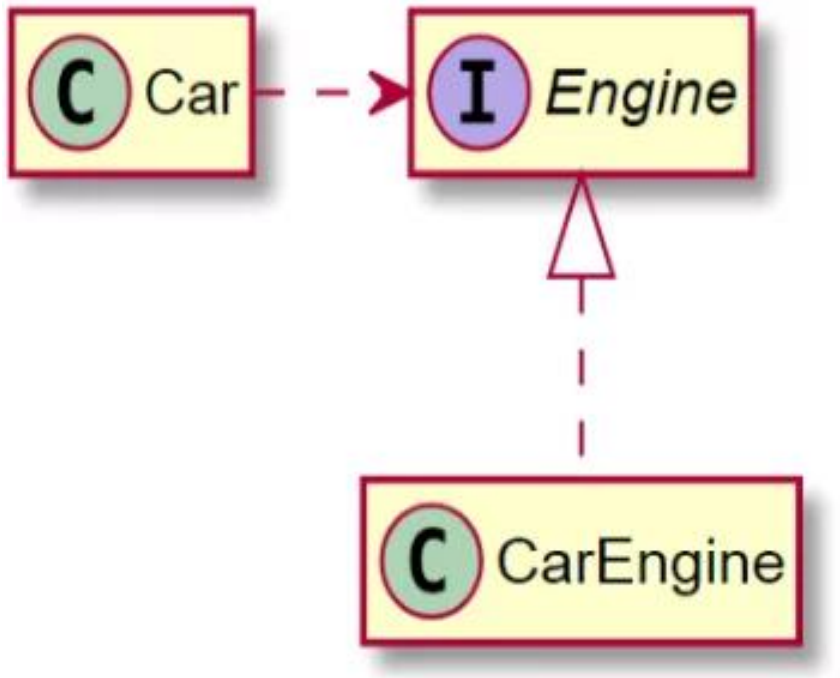
НАЗАД К ОСНОВАМ: ВНЕДРЕНИЕ ЗАВИСИМОСТИ (DI)

Объяснение для пятилетних

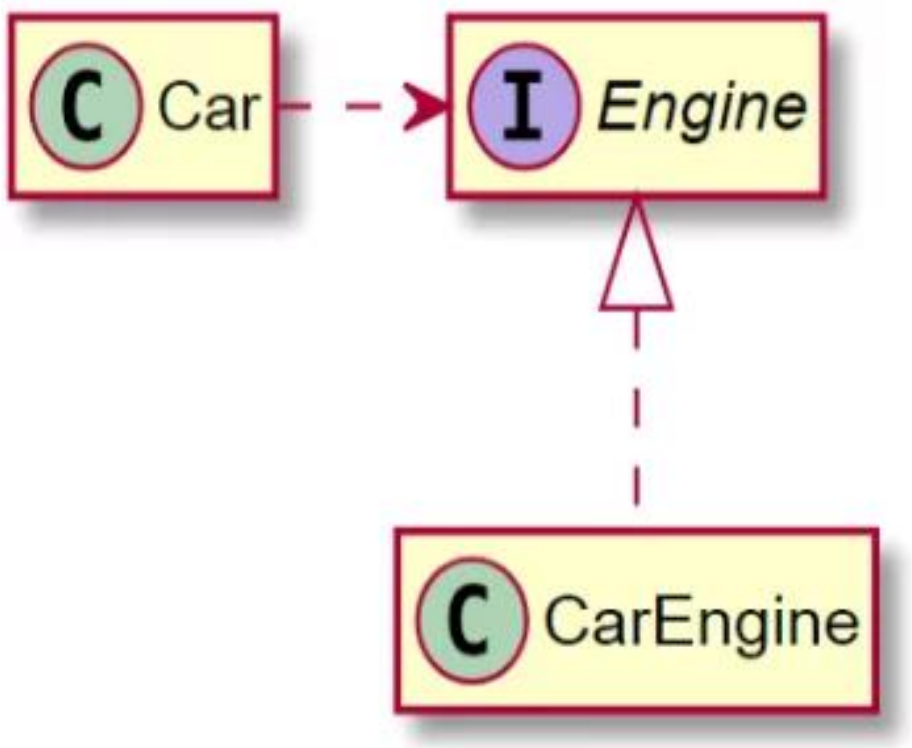


НАЗАД К ОСНОВАМ: ВНЕДРЕНИЕ ЗАВИСИМОСТИ (DI)



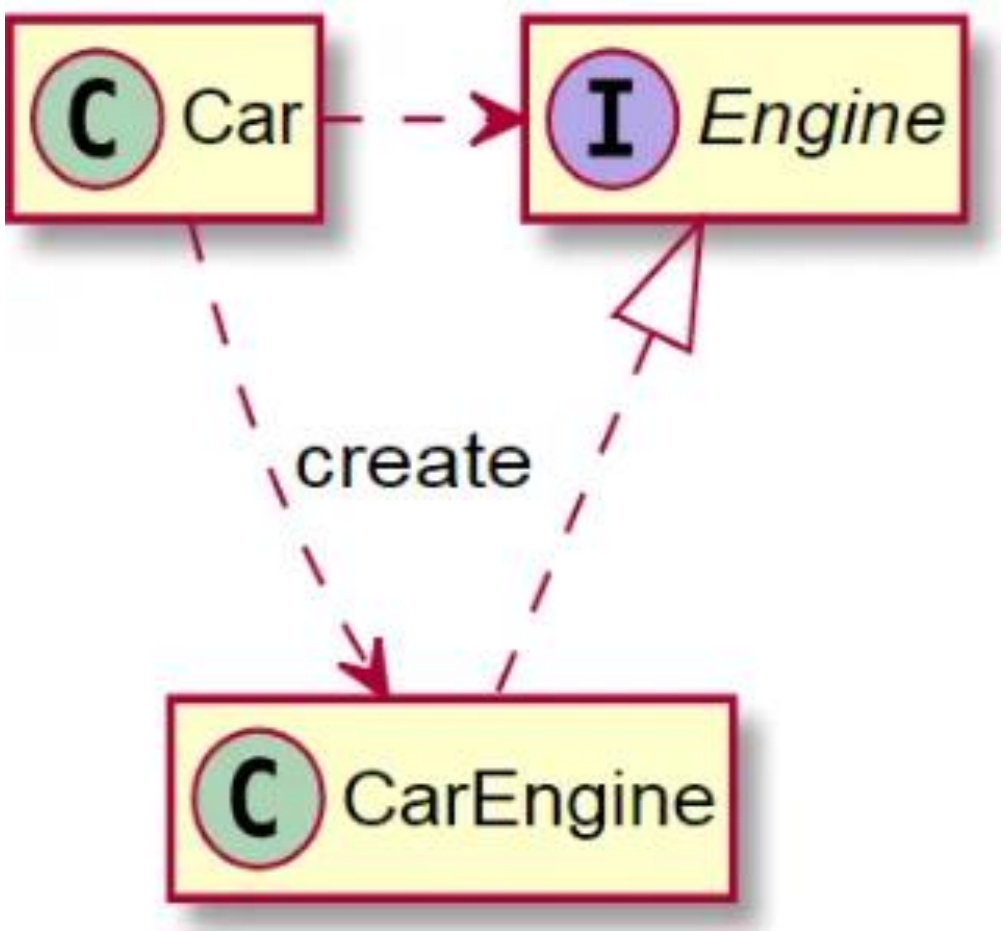


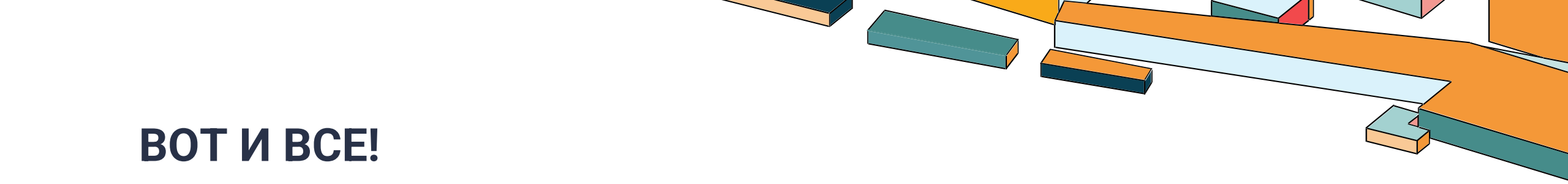
НАЗАД К ОСНОВАМ: ВНЕДРЕНИЕ ЗАВИСИМОСТИ (DI)




```java
public interface Engine {
    boolean isStart();
}

class CarEngine implements Engine {

    @Override
    public boolean isStart() {
        return true;
    }
}

class Car {

    public void start() {
        Engine engine = new CarEngine();
        if (engine.isStart()) {
            System.out.println("Start!");
        }
    }
}
```

НАЗАД К ОСНОВАМ: ВНЕДРЕНИЕ ЗАВИСИМОСТИ (DI)





ВОТ И ВСЕ!

Концепция Dependency Injection состоит в том, чтобы перенести ответственность за создание экземпляра объекта за пределы класса и передать уже созданный экземпляр объекта обратно.

```
Car car = new Car(new CarEngine());
```

```
Car car = new Car();
car.setEngine(new CarEngine());
```

```
class Car {
    private Engine engine = new CarEngine();
    ...
}
```

```
class Car {

    private Engine engine;

    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        if (engine.isStart()) {
            System.out.println("Start!");
        }
    }
}
```

DEPENDENCY INJECTION

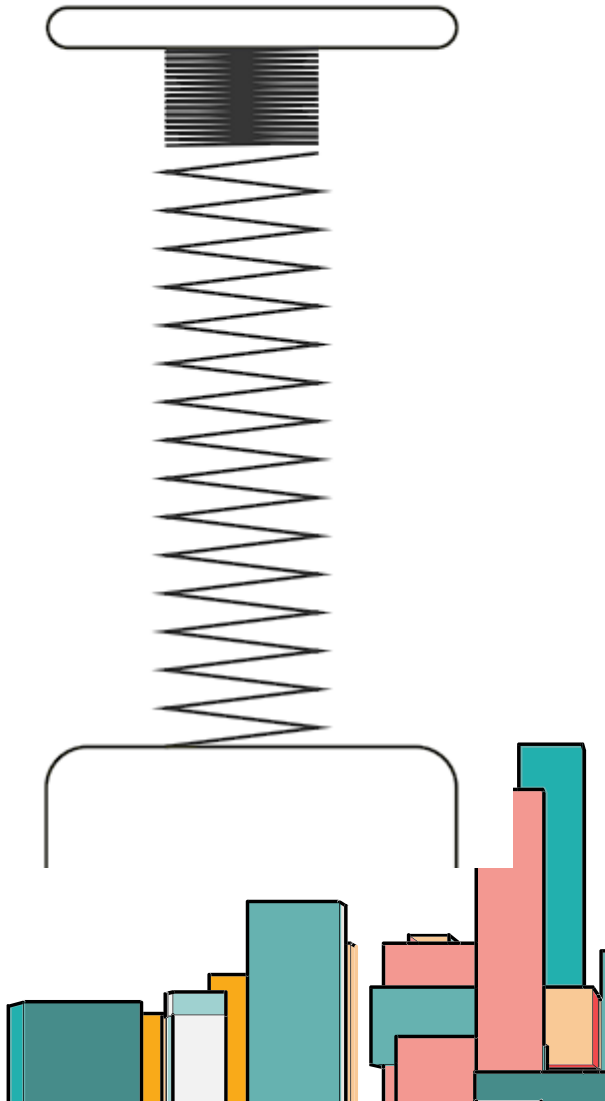
- **Dependency Injection разрывает жёсткую связь между классом и его вспомогательными сервисами.** Благодаря ему идеально реализуется Low Coupling

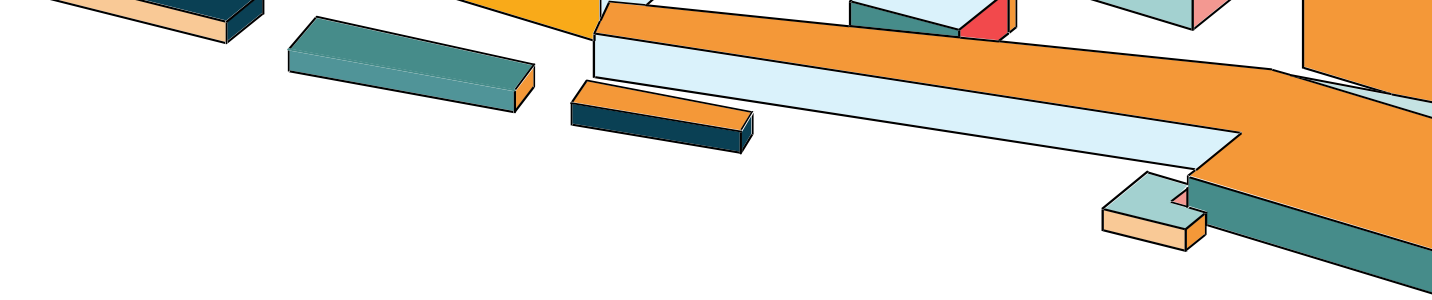
- **DI улучшает тестируемость кода.** Все классы минимально знают друг о друге и тестировать их легко – все зависимости заменяемы

- **При переносе модуля в другие приложения уменьшается число классов адаптирующих код.** Это редкий случай, но слабая связанность решает и этот вопрос

- **DI позволяет проще переносить классы в другие приложения, находящиеся на верхних уровнях, а нижние — менять на другие реализации.** Такая модульность или слоистость приложения позволяет делать в системе намного больше смелых изменений, чем в закрепощённом проекте

Dependency Injection ориентируется на создание слабой связанности.





ВНЕДРЕНИЕ ЧЕРЕЗ КОНСТРУКТОР

Constructor Injection

1

ПЛЮСЫ

\+ Этот подход легко реализовать, нет никаких подводных камней

\+ Все зависимости обязательны

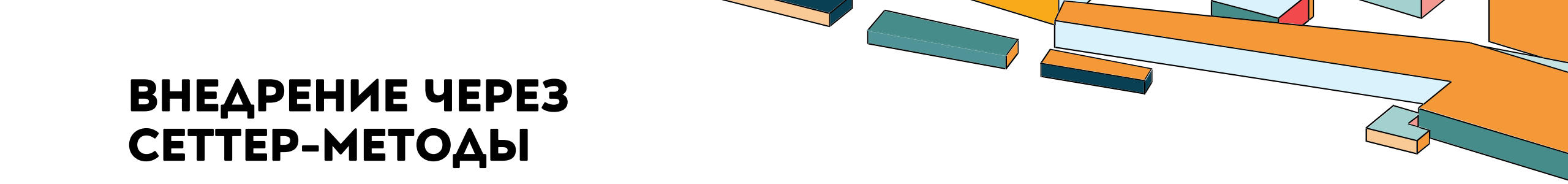
Low Coupling говорит, что связь между классами должна быть минимальна. Если вы будете прокидывать пять и более объектов, то, что-то идёт не так, и сам по себе класс будет закрепощён

МИНУСЫ

× Нет возможности не внедрять в классы зависимости. Все они обязательные

— Нельзя добавить динамичности, когда сначала пробрасывается один набор классов, а затем другой

× Если аргументов слишком много, то реализация будет выглядеть некрасиво. Это будет неправильно с точки зрения Low Coupling



ВНЕДРЕНИЕ ЧЕРЕЗ СЕТТЕР-МЕТОДЫ

Setter Injection

2

ПЛЮСЫ

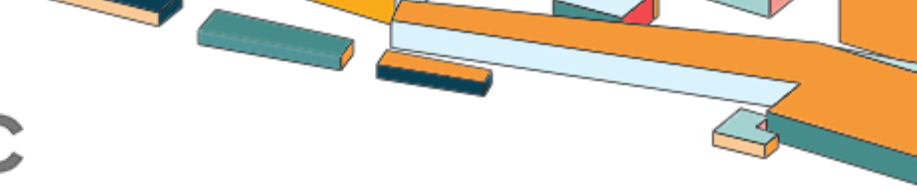
+ Есть возможность выбора, внедрять класс или нет
+ Внедрять классы можно динамически

МИНУСЫ

× Можно забыть внедрить зависимость и не получить чего-то в конечном результате

— Надо не забывать делать проверки на null и обрабатывать ситуации, если вдруг зависимость не была прокинута изначально

× Для каждой зависимости необходимо описывать свой сеттер. Код немного увеличивается в размерах



3 внедрение через интерфейс

Interface Injection

В этом подходе прокидывание зависимости происходит один к одному. Система сама догадывается о том, какую сущность необходимо прокинуть, и для этого ничего дополнительного делать не нужно

Здесь каждый интерфейс должен реализовываться одним классом. Иначе невозможно будет определить, какой класс необходимо использовать для внедрения

Плюсы и минусы этого подхода точно такие же, как у Injection через конструктор

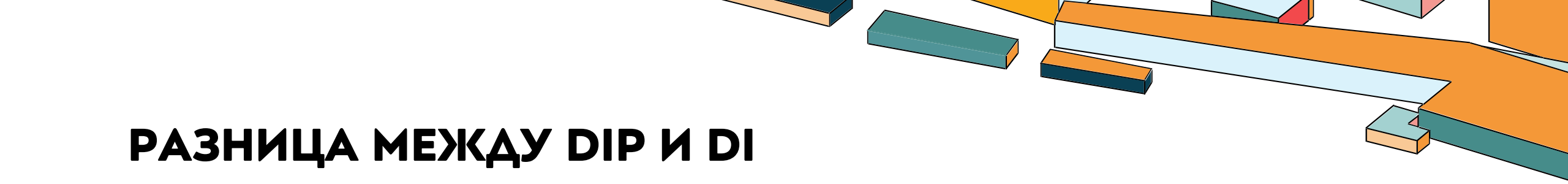
4 внедрение через свойство

Property Injection

Пользоваться мы им не рекомендуем

Property injection похож на подход с сеттер-методами. Разница в том, что достаточно сделать свойства класса публичными и добавить в коде проверки на null. Здесь всё очень минимально: определили свойства, прокинули зависимости и сделали нужные проверки

В отличии от сеттер-методов property injection лишает вас возможности писать бизнес-логику и проверки. При наступлении необходимости перехода на сеттер-методы, вам придётся много переписывать



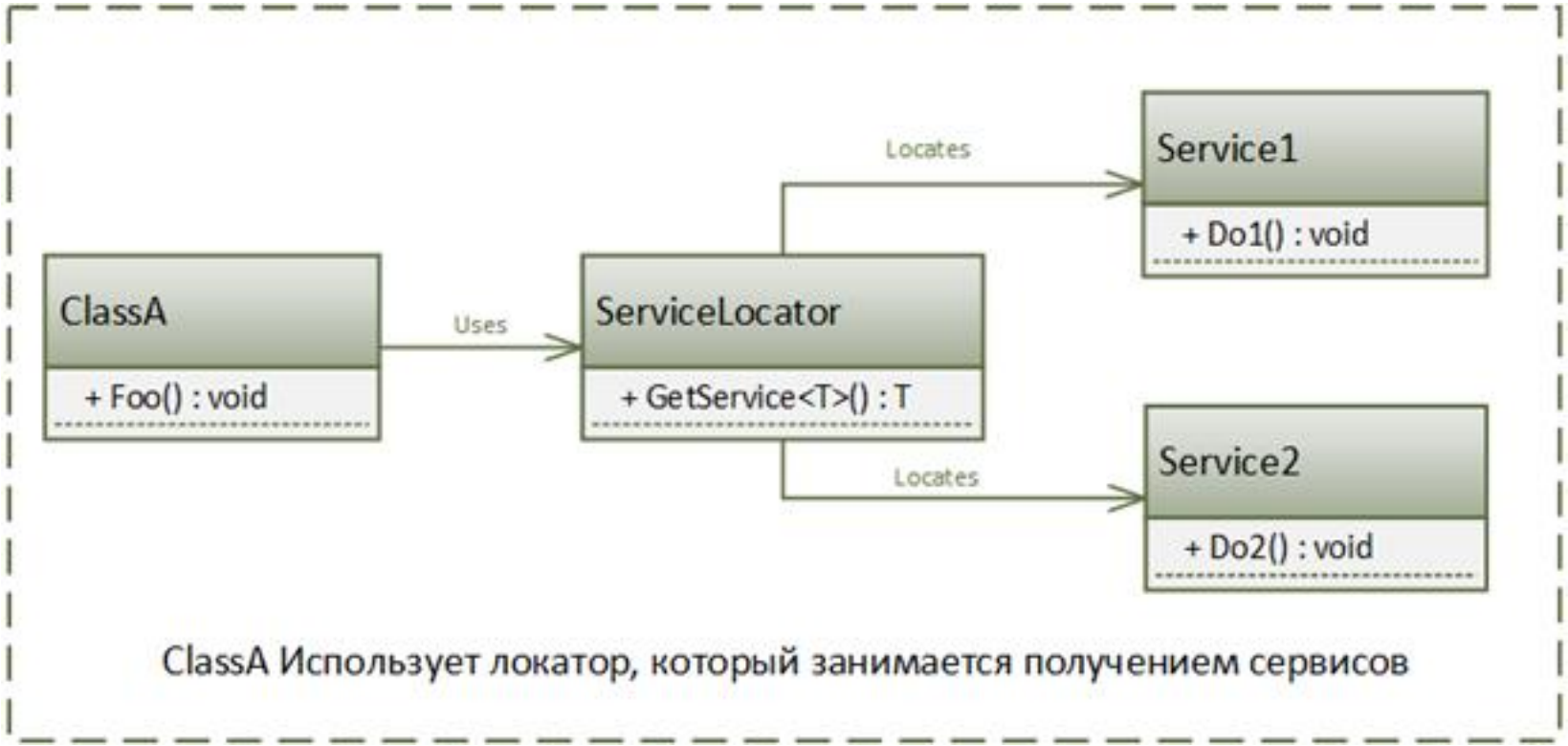
РАЗНИЦА МЕЖДУ DIP И DI

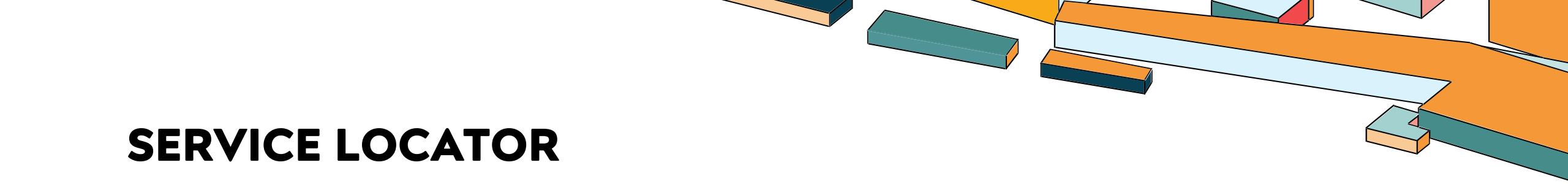
Задача Dependency Injection заключается в том, чтобы предоставлять программному компоненту внешнюю зависимость. То есть он задаёт способ, которым будут доставляться объекты в конкретный instance

Инверсия про то, как относиться к работе с кодом, а внедрение – про способы:

- Dependency Inversion Principle задаёт общий принцип инвертирования зависимости
- Dependency Injection рассказывает о конкретных способах внедрения зависимости

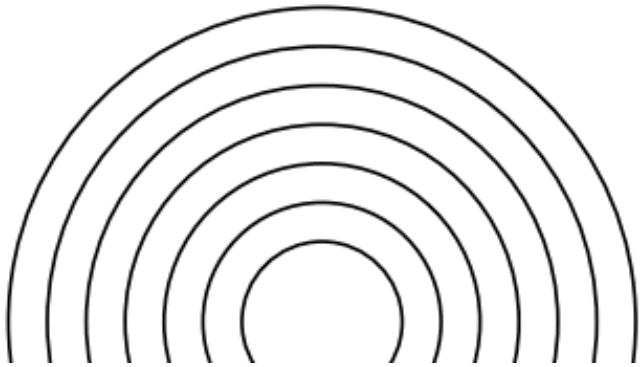
И ТУТ ПОЯВЛЯЕТСЯ ....





SERVICE LOCATOR

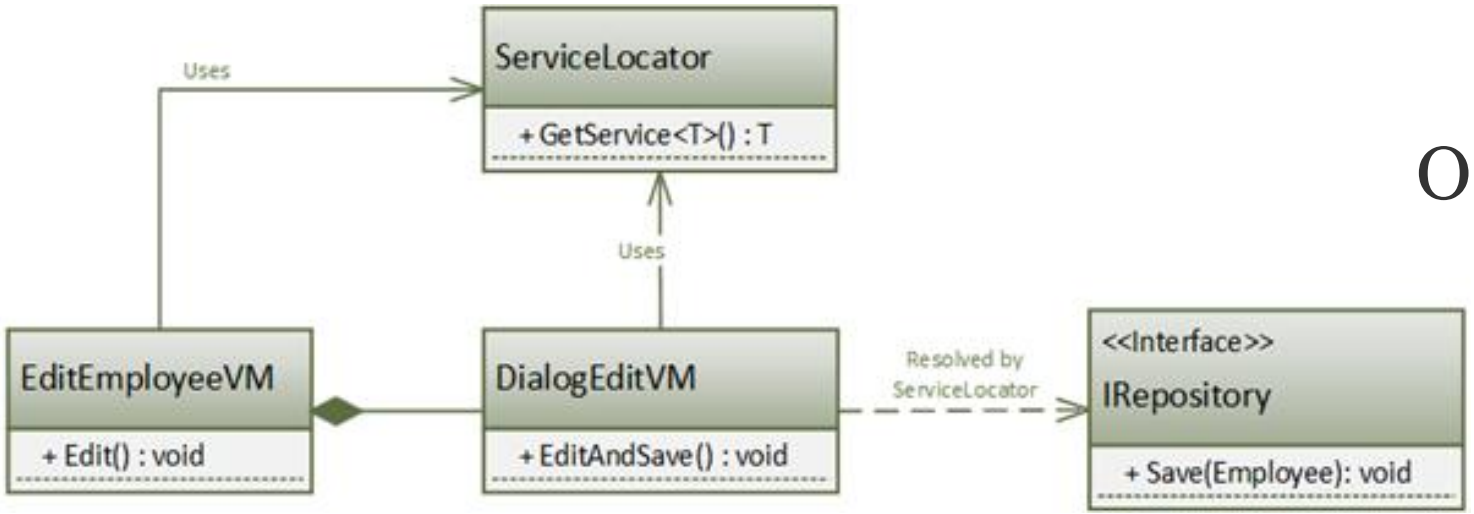
В приложении процесс инстанцирования объектов можно абстрагировать в отдельный слой. Для этого мы вместо создания класса по месту, просим некоторую сущность создать его за нас. Такой сущностью будет Service Locator.



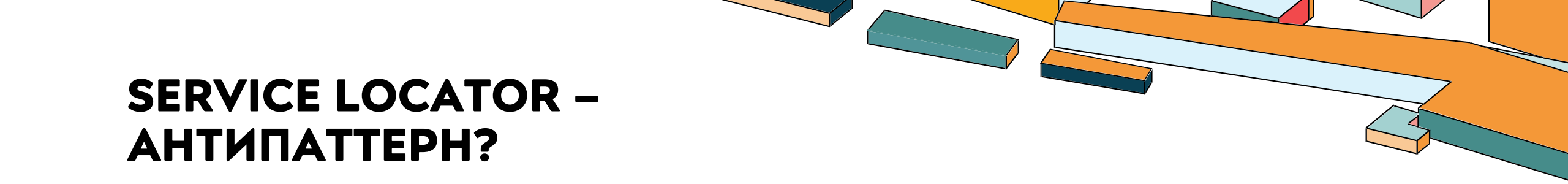
- → Задача Service Locator в том, чтобы хранить и возвращать объект по запросу. **Способы хранения информации о создании класса при этом ничем не регламентированы**
- → Service Locator не имеет канонической реализации. Важна именно идея того, как создаются объекты системы, а не то, как это будет выглядеть в коде. **В таком подходе создаваемый класс начинает определяться локатором, а не внутри реализуемого метода**
- → **Благодаря этому подходу, система становится слабо связанной и частично удовлетворяется Low Coupling.** Это позволяет разделить бизнес-логику и процесс создания классов. Так у нас упрощается сопровождение и расширение кода. При этом тестирование кода в разы улучшается

ЧЕМ ЖЕ ПЛОХ SERVICE LOCATOR?

ОПАСНОСТЬ ПАТТЕРНА




```
class EditEmployeeViewModel
{
    private IServiceLocator _serviceLocator;
    public EditEmployeeViewModel(IServiceLocator serviceLocator)
    {
        _serviceLocator = serviceLocator;
    }
}
```



SERVICE LOCATOR – АНТИПАТТЕРН?

К 2010 году популярность паттерна упала к минимуму, появились более продвинутые практики. Service Locator стал считаться антипаттерном, рекомендуем не использовать его в своих приложениях. К такому решению подталкивает то, что он является аналогом глобального объекта, к которому обращаются из всех частей приложения.

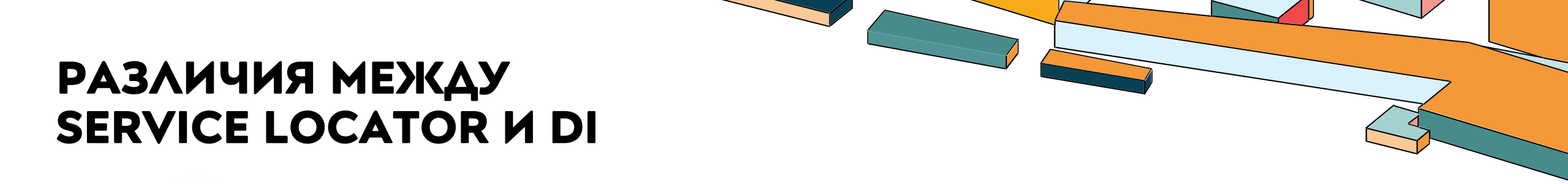
плюс подхода

\+ **Инстанцирование классов отделено от основного кода.** Они входят в свою специальную абстракцию → в тестах проще подменить одну реализацию на другую; не нужно знать, как создавать классы и какие аргументы передавать в конструктор

минус подхода

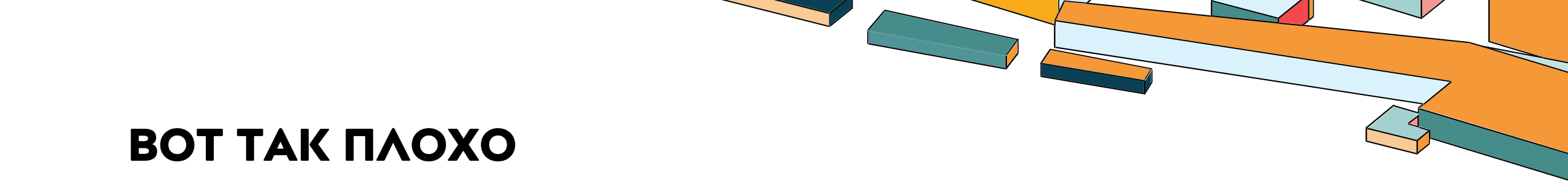
— **Вместо зависимости от конкретного класса мы во всем приложении начинаем зависеть от Service Locator.** Появляется единая точка отказа → слабая связанность решается только частично

РАЗЛИЧИЯ МЕЖДУ SERVICE LOCATOR И DI



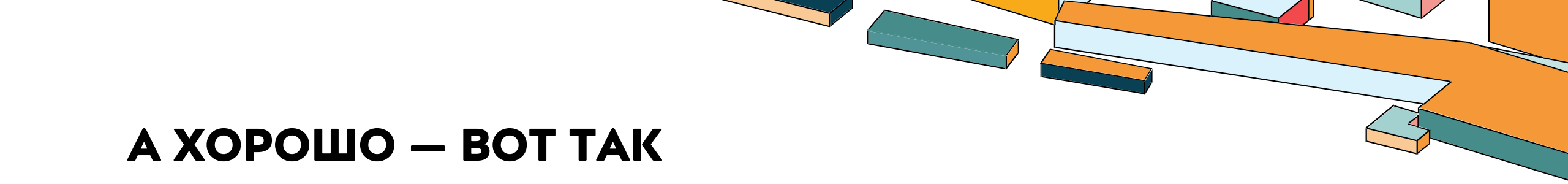
проявляются в том, как они работают с зависимостями:

- В случае Service Locator конечные классы зависят от локатора
- В Dependency Injection зависимость внедряется на верхних уровнях
- В Service Locator конечные классы знают про существование Service Locator и зависят от него
- В Dependency Injection зависимость внедряется на верхних уровнях или в отдельной абстракции, а конечные классы ничего не знают о DI, и это очень весомый плюс



ВОТ ТАК ПЛОХО

```
public class OrderProcessor : IOrderProcessor
{
    public void Process(Order order)
    {
        var validator = Locator.Resolve<IOrderValidator>();
        if (validator.Validate(order))
        {
            var shipper = Locator.Resolve<IOrderShipper>();
            shipper.Ship(order);
        }
    }
}
```

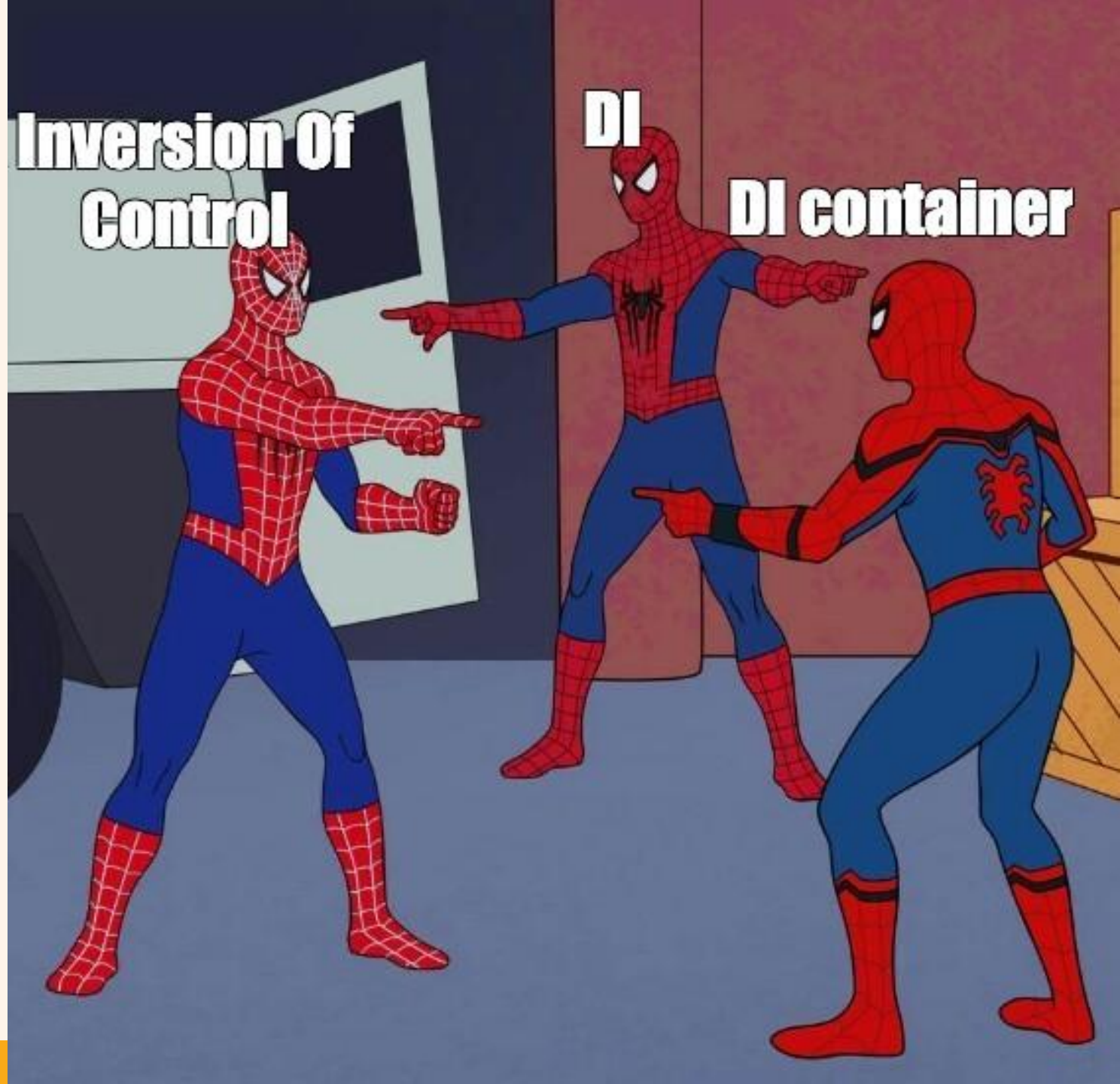


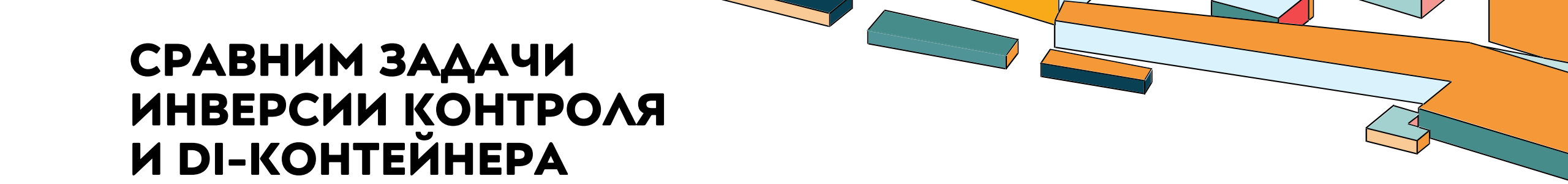
А ХОРОШО — ВОТ ТАК

```
var validator = Locator.Resolve<IOrderValidator>();
var shipper = Locator.Resolve<IOrderShipper>();
var processor = new OrderProcessor(validator, shipper);
processor.Process(order);
```

```
public class OrderProcessor : IOrderProcessor
{
    public OrderProcessor(IOrderValidator validator, IOrderShipper shipper)

    public void Process(Order order)
}
```

DI-КОНТЕЙНЕР



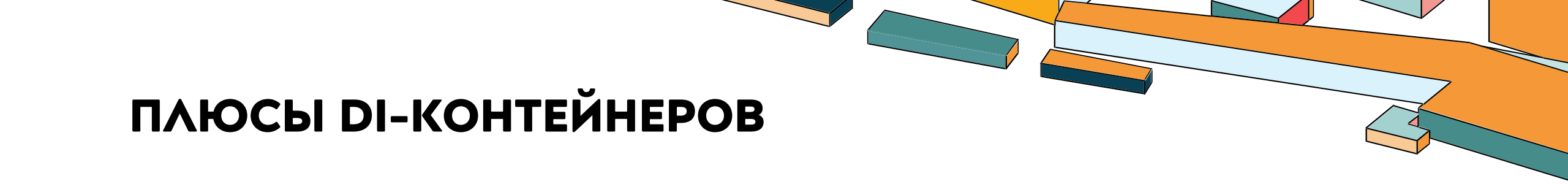


СРАВНИМ ЗАДАЧИ ИНВЕРСИИ КОНТРОЛЯ И DI-КОНТЕЙНЕРА

Инверсия контроля — это общий принцип, определяющий поток в нашей программе.
Он определяет верхнеуровневую идею и сам подход. При этом он не является программной реализацией.

DI-**контейнер — это библиотека или фреймворк, который реализует концепцию** DI
Это уже выраженная в коде библиотека, которую можно скачать с GitHub и использовать в проекте.

В реализации DI-**контейнера всегда используется** Service Locator.
Но его область применения ограничена самой библиотекой, и внутри DI-контейнера он отлично справляется со своей задачей.
Работа и обращение к Service Locator из кода запрещены.



ПЛЮСЫ DI-КОНТЕЙНЕРОВ

- **Это универсальное решение для сбора полноценного приложения из множества компонентов.** Если в обоих проектах внедрён DI, то можно скопировать нужные классы во второе приложение. Их адаптация займёт буквально пару минут и в этом одна из их сильных сторон

- **DI-контейнеры из-за своей архитектуры делают слабо связанные классы, полностью удовлетворяющие Low Coupling. Используя их, можно легко подменить один класс другим,** поскольку сборка проекта происходит на уровне конфигурационного файла

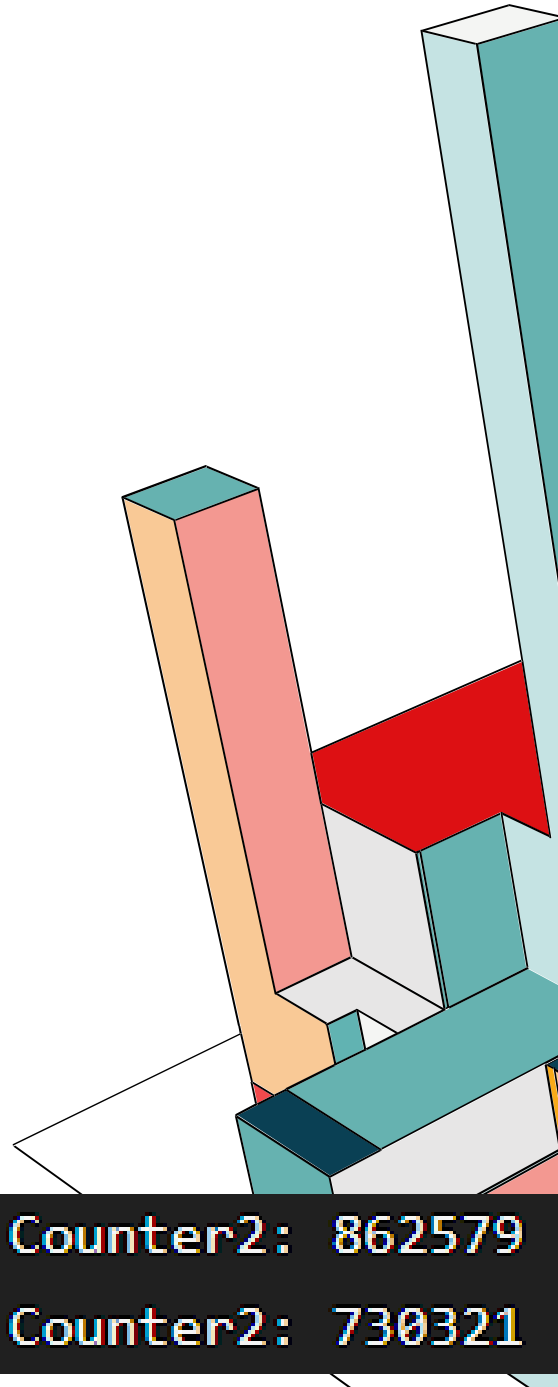
- Классы друг о друге не знают, и юнит-тесты пишутся легко без каких-либо сложностей

ЖИЗНЕННЫЙ ЦИКЛ ЗАВИСИМОСТЕЙ

С точки зрения жизненного цикла сервисы могут представлять
один из следующих типов:

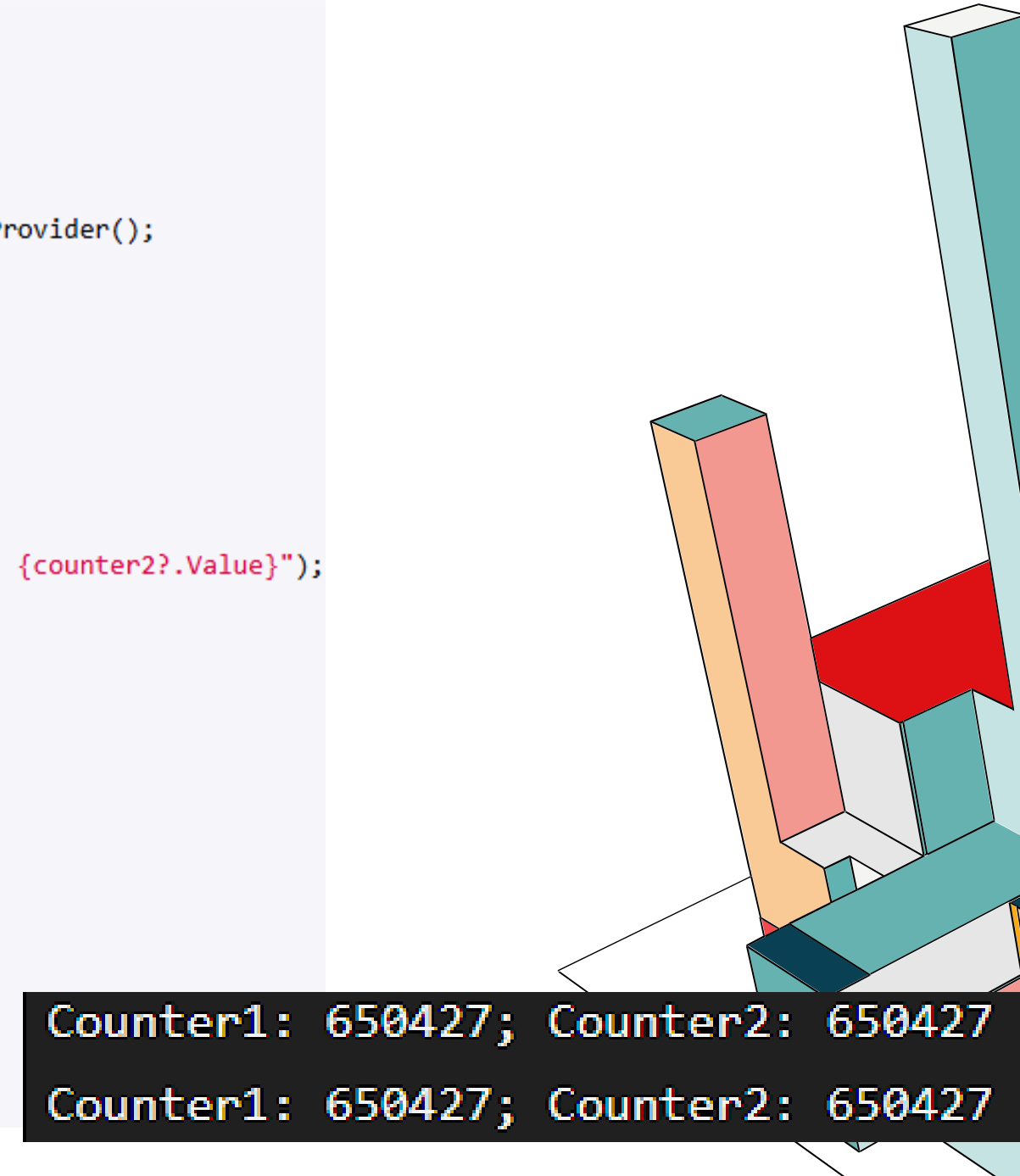
- **Transient**: при каждом обращении к сервису создается новый объект сервиса
- **Scoped**: создается отдельный контекст или scope, и в рамках этого контекста при всех обращениях у сервису будет использоваться один и тот же объект сервиса.
- **Singleton**: объект сервиса создается при первом обращении к нему, все последующих обращениях используется один и тот же ранее созданный
объект сервиса

```
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection()
    .AddTransient<ICounter, RandomCounter>();

using ServiceProvider serviceProvider = services.BuildServiceProvider();

PrintCounters();
PrintCounters();

void PrintCounters()
{
    var counter1 = serviceProvider.GetService<ICounter>();
    var counter2 = serviceProvider.GetService<ICounter>();
    Console.WriteLine($"Counter1: {counter1?.Value}; Counter2: {counter2?.Value}");
}

public interface ICounter
{
    int Value { get; }
}
public class RandomCounter : ICounter
{
    static Random rnd = new Random();
    private int _value;
    public RandomCounter() => _value = rnd.Next(0, 1000000);
    public int Value => _value;
}
```



```
Counter1: 939614; Counter2: 862579
Counter1: 741773; Counter2: 730321
```

```
using Microsoft.Extensions.DependencyInjection;

IServiceCollection services = new ServiceCollection()
    .AddSingleton<ICounter, RandomCounter>();

using ServiceProvider serviceProvider = services.BuildServiceProvider();

PrintCounters();
PrintCounters();
void PrintCounters()
{
    var counter1 = serviceProvider.GetService<ICounter>();
    var counter2 = serviceProvider.GetService<ICounter>();
    Console.WriteLine($"Counter1: {counter1?.Value}; Counter2: {counter2?.Value}");
}

public interface ICounter
{
    int Value { get; }
}
public class RandomCounter : ICounter
{
    static Random rnd = new Random();
    private int _value;
    public RandomCounter() => _value = rnd.Next(0, 1000000);
    public int Value => _value;
}
```



ИСПОЛЬЗУЕМЫЕ ИСТОЧНИКИ

📕 **Внедрение зависимости** (DI)

https://topjava.ru/blog/back-to-basics-dependency-injection

📕 Service Locator.

https://sergeyteplyakov.blogspot.com/2013/03/di-service-locator.html

📕 **Отличие** Dependency Injection **от** Service Locator

https://habr.com/ru/articles/465395/

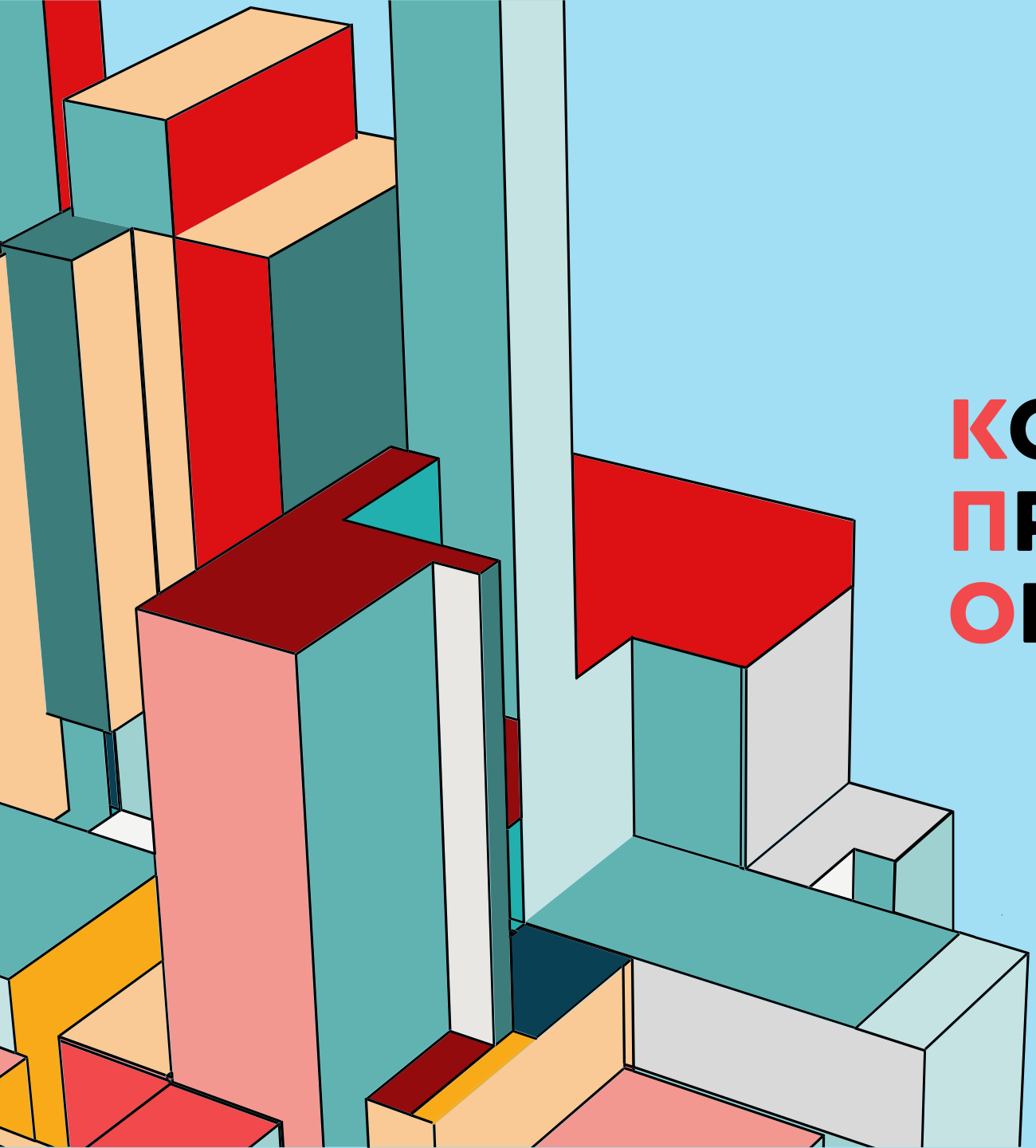
📕 **Жизненный цикл зависимостей**

https://metanit.com/sharp/dotnet/1.3.php

СПАСИБО!

Виденин Сергей

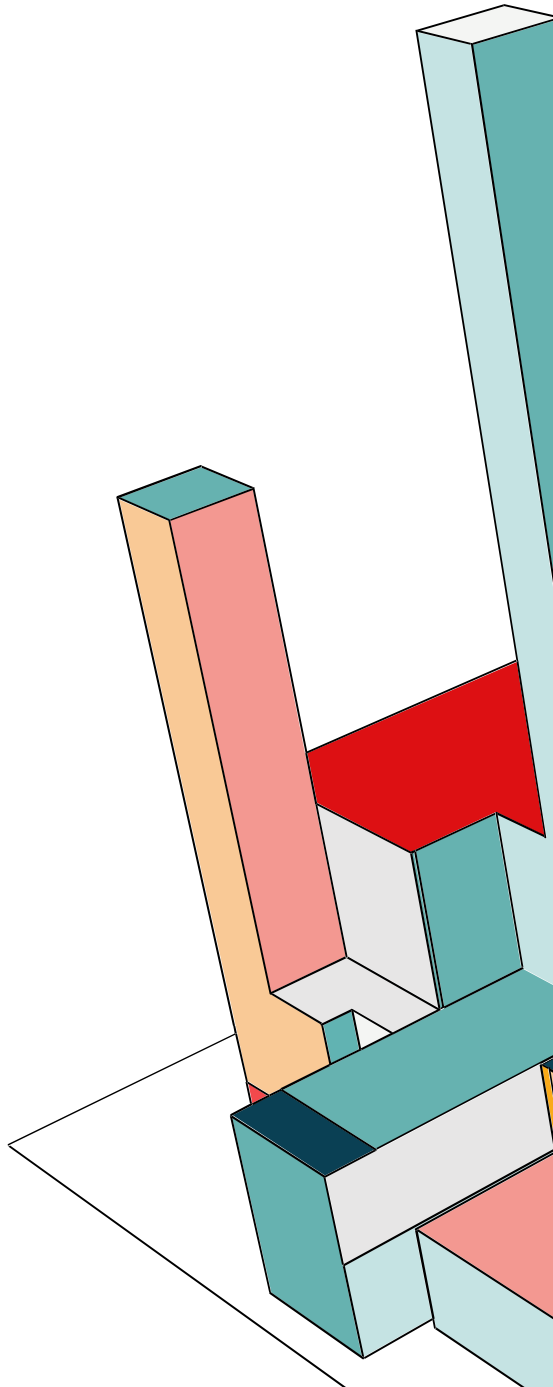
@videninserg



КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ПЛАН ЛЕКЦИИ № 3

1. Тестирование ПО
2. Этапы и виды тестирования
3. Юнит тестирование
4. Принципы эффективного юнит тестирования
   - как измерить эффективность тестов
   - как писать эффективные тесты
   - когда нужно использовать моки

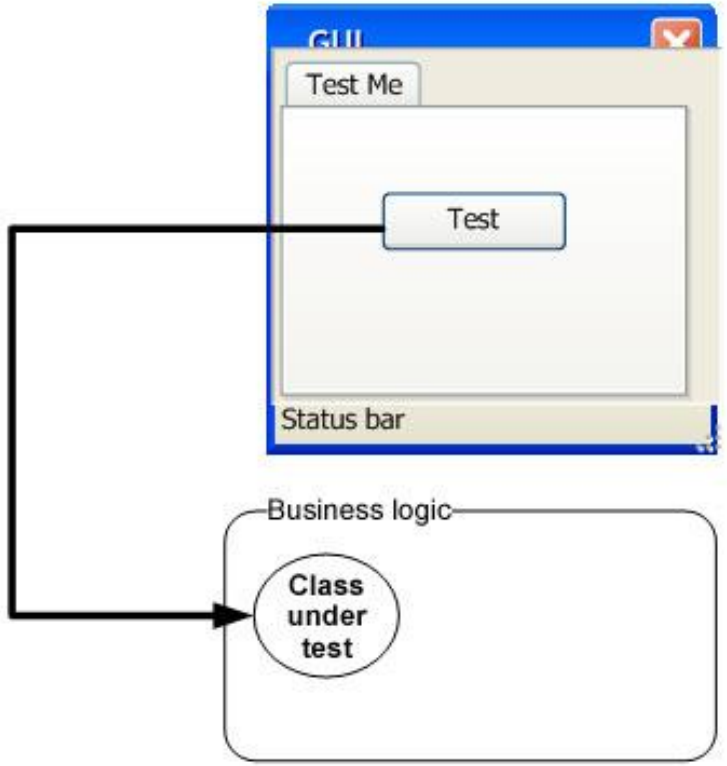


ЧТО ТАКОЕ ТЕСТИРОВАНИЕ

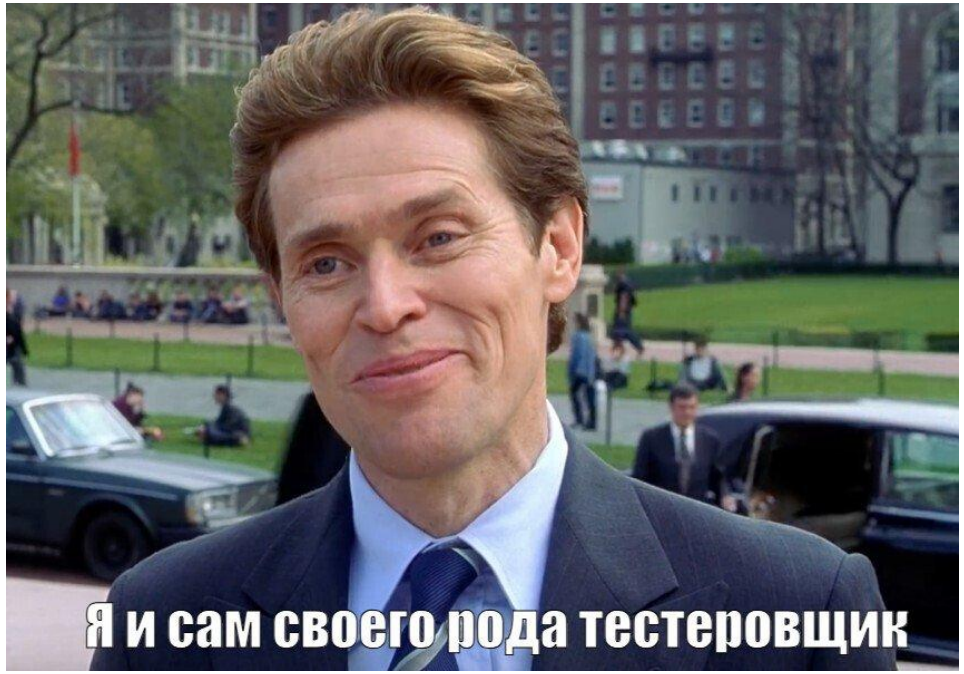
Тестирование показывает, соответствует ли ПО ожиданиям разработчиков.

Тестирование проводят тестировщики — они отвечают за то, что продукт соответствует всем заданным требованиям.

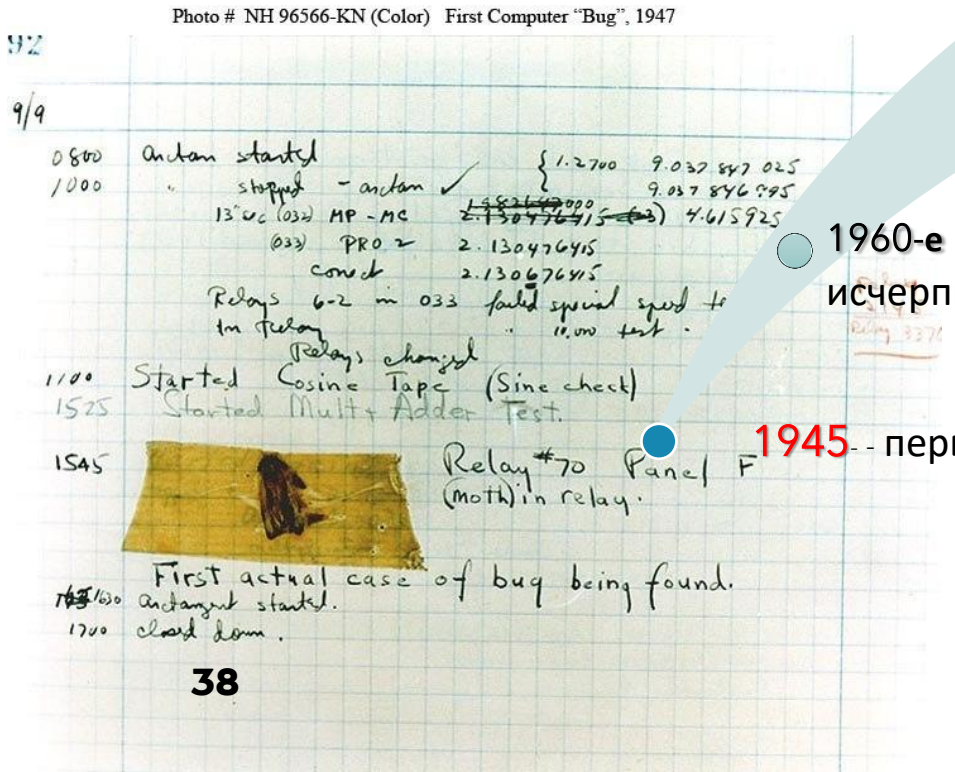
ВСЕ МЫ НЕМНОГО... ТЕСТЕРЫ



Добавляете кнопку и проверяете, соответствует ли полученный результат вашим ожиданиям



ЭТАПЫ РАЗВИТИЯ ТЕСТИРОВАНИЯ



1960-**е**
исчерпывающее тестирование

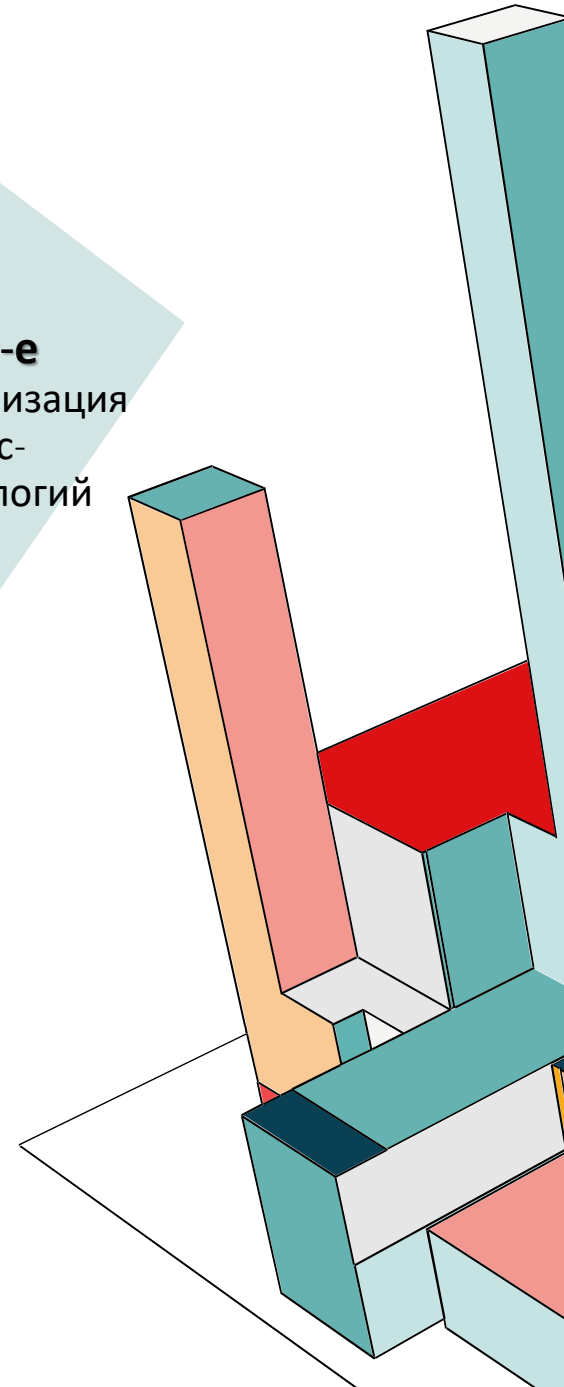
1970-**е**
поиск дефектов

1980-**е**
предупреждение дефектов

1990-**е**
обеспечение качества

2000-**е**
оптимизация бизнес-технологий

1945 - - первый случай, когда был найден баг



ПАРАДОКС ТЕСТИРОВАНИЯ

1. С одной стороны, тестирование позволяет убедиться, что продукт работает хорошо.
2. С другой – выявляет ошибки в ПО, показывая, что продукт не работает.

Вторая цель тестирования является более продуктивной с точки зрения улучшения качества, так как не позволяет игнорировать недостатки ПО.

Программа НЕ работает
(60-е)

VS

Программа работает
(70-е)

80-Е: ПРЕДУПРЕЖДЕНИЕ ДЕФЕКТОВ

- Появляются идеи о необходимости методологии тестирования, в частности, что тестирование должно включать в себя проверки на всех стадиях разработки ПО.

- В ходе тестирования надо проверить не только собранную программу, но и требования, код, архитектуру, сами тесты.

- В середине 1980-х появились первые инструменты для автоматизированного тестирования

90-Е: ОБЕСПЕЧЕНИЕ КАЧЕСТВА

- В начале 1990-х в понятие «тестирование» стали включать планирование, проектирование, создание, поддержку и выполнение тестов и тестовых окружений, и это означало переход от тестирования к обеспечению качества, охватывающего весь цикл разработки ПО.

- Начинают появляться различные программные инструменты для поддержки процесса тестирования: более продвинутые среды для автоматизации с возможностью создания скриптов и генерации отчетов, системы управления тестами, ПО для проведения нагрузочного тестирования.

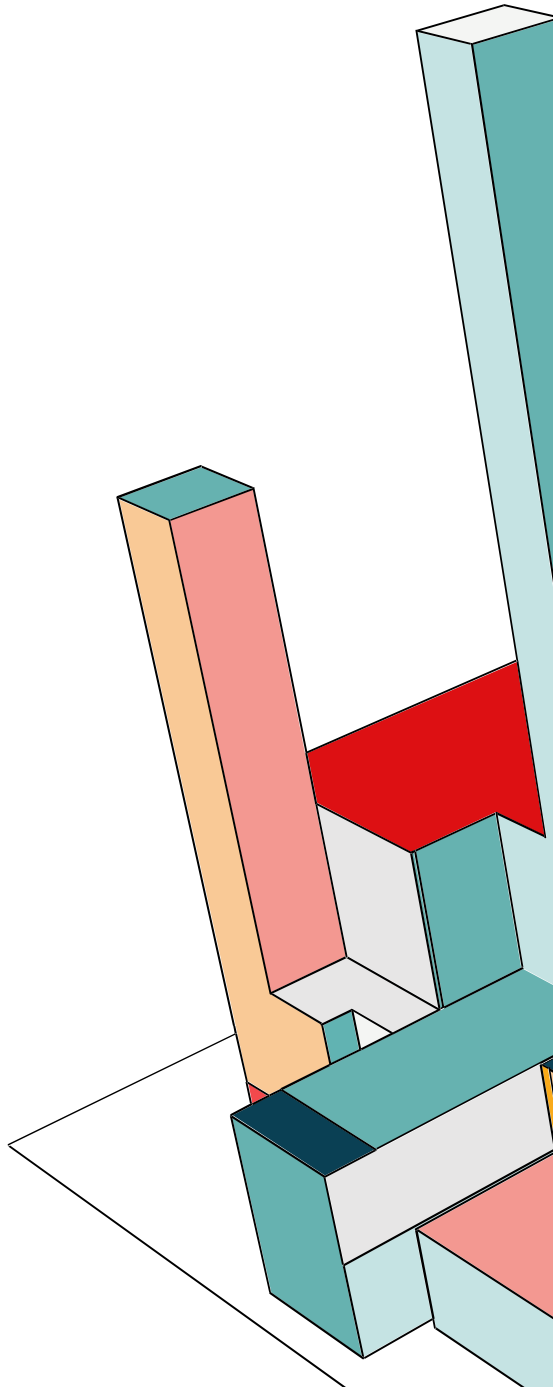
2000-Е: ТЕСТИРОВАНИЕ И БИЗНЕС

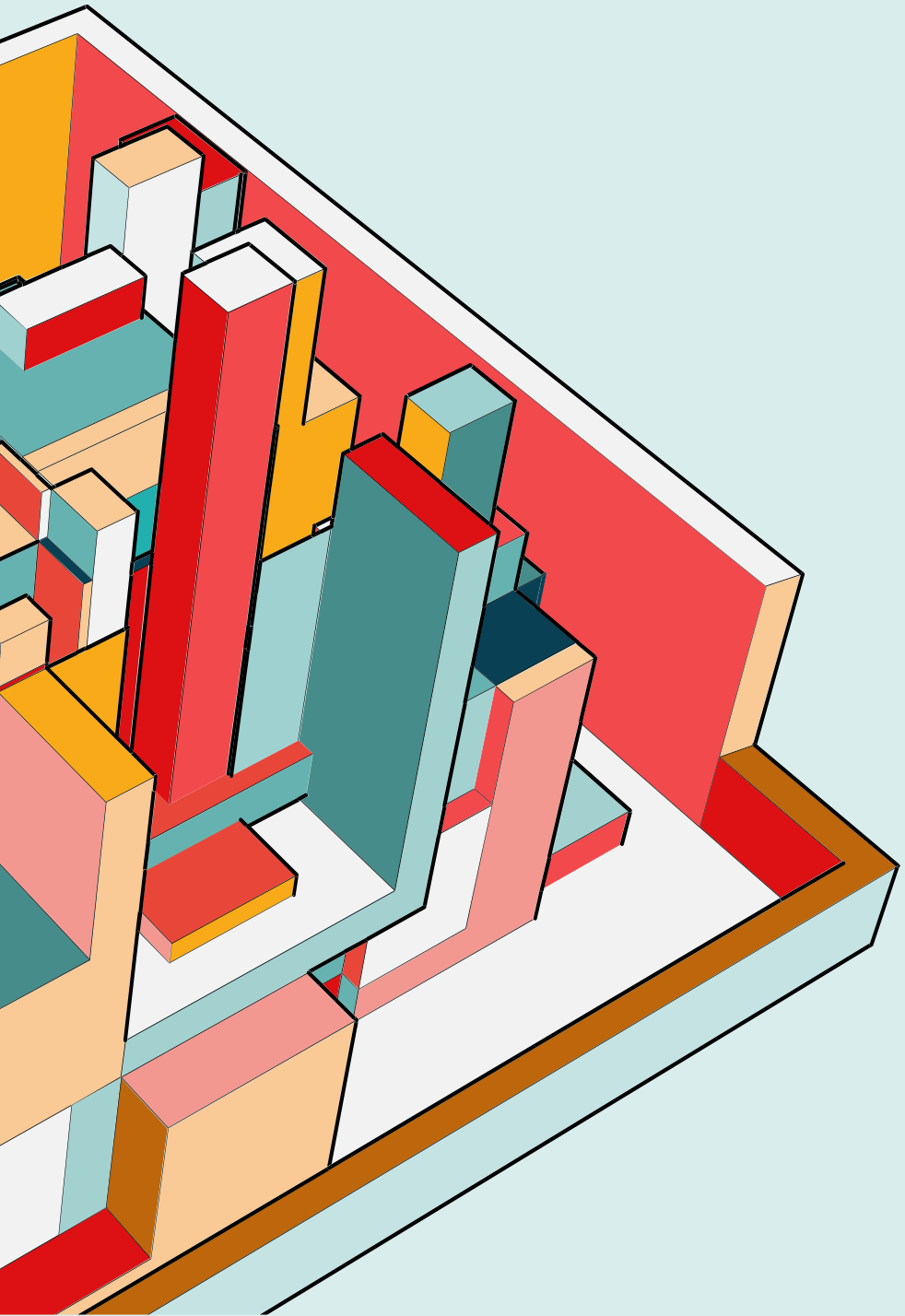
- В 2000-х появилось еще более широкое определение тестирования, когда в него было добавлено понятие «оптимизация бизнес-технологий»

- Основной подход заключается в оценке и максимизации значимости всех этапов жизненного цикла разработки ПО для достижения необходимого уровня качества, производительности, доступности.

КАКИЕ БЫВАЮТ ЭТАПЫ ТЕСТИРОВАНИЯ

1. Проработка требований к продукту
2. Анализ требований
3. Разработка стратегии и плана тестирования
    - Выбор методов тестирования
    - Анализ потенциальных рисков
    - Планирование ресурсов
4. Создание тестовой документации
5. Тестирование
6. Эксплуатация и поддержка

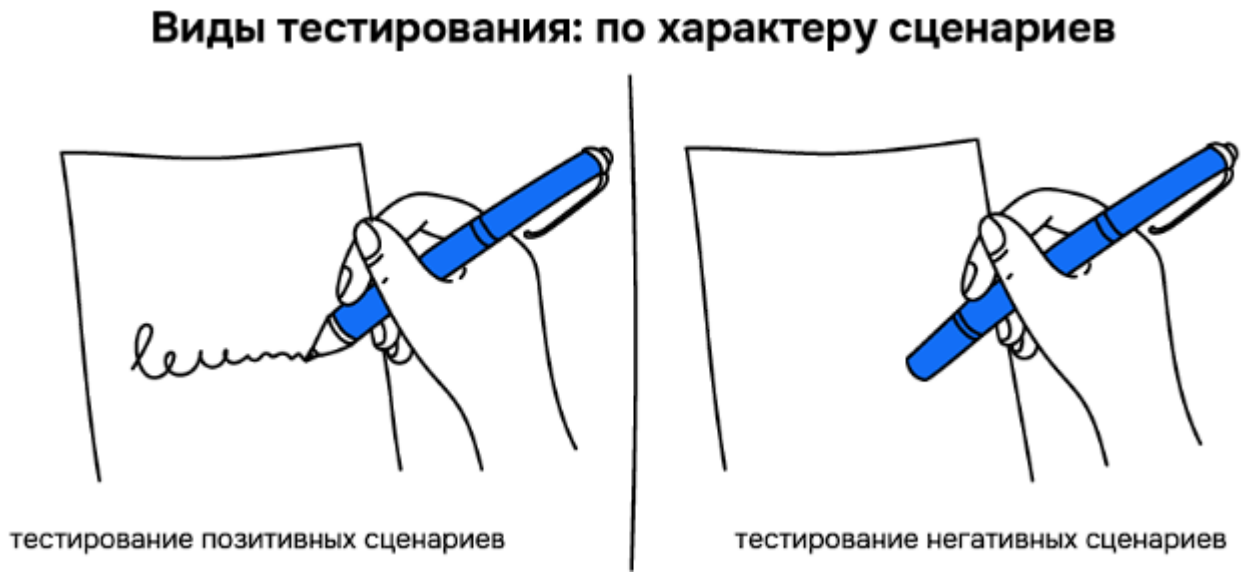




ВИДЫ ТЕСТИРОВАНИЯ

ПО ХАРАКТЕРУ СЦЕНАРИЕВ

Сценарий в тестировании — это описание того, как пользователь будет взаимодействовать с готовым продуктом.



ПО КРИТЕРИЯМ ЗАПУСКА ПРОГРАММЫ ИЛИ КОДА

Критерии запуска программы или кода означают условия, которые необходимо выполнить для запуска тестов.

ПО СТЕПЕНИ АВТОМАТИЗАЦИИ ТЕСТИРОВАНИЯ

Ручное тестирование позволяет проверить различные аспекты программы: удобство использования, внешний интерфейс, а также воспроизводить нестандартные ситуации, которые может быть сложно автоматизировать.

Автоматизированные тесты могут проверить функциональность, производительность, совместимость и тд.

ПО ОБЪЕКТАМ ТЕСТИРОВАНИЯ

Эта группа объединяет в себе виды, которые предполагают определение того, какие части программы или системы подвергаются тестированию

ПО СТЕПЕНИ ЗНАНИЯ СИСТЕМЫ

Эта группа объединяет в себе виды, которые используются в зависимости от этого, насколько тестировщик знаком с тестируемым продуктом.

ПО ВРЕМЕНИ ПРОВЕДЕНИЯ ТЕСТИРОВАНИЯ

В эту группу попадают виды тестирования, которое проводят в разные моменты разработки продукта: например, до выкатки на прод и после.

ПРИМЕРЫ ТЕСТИРОВАНИЯ ПО

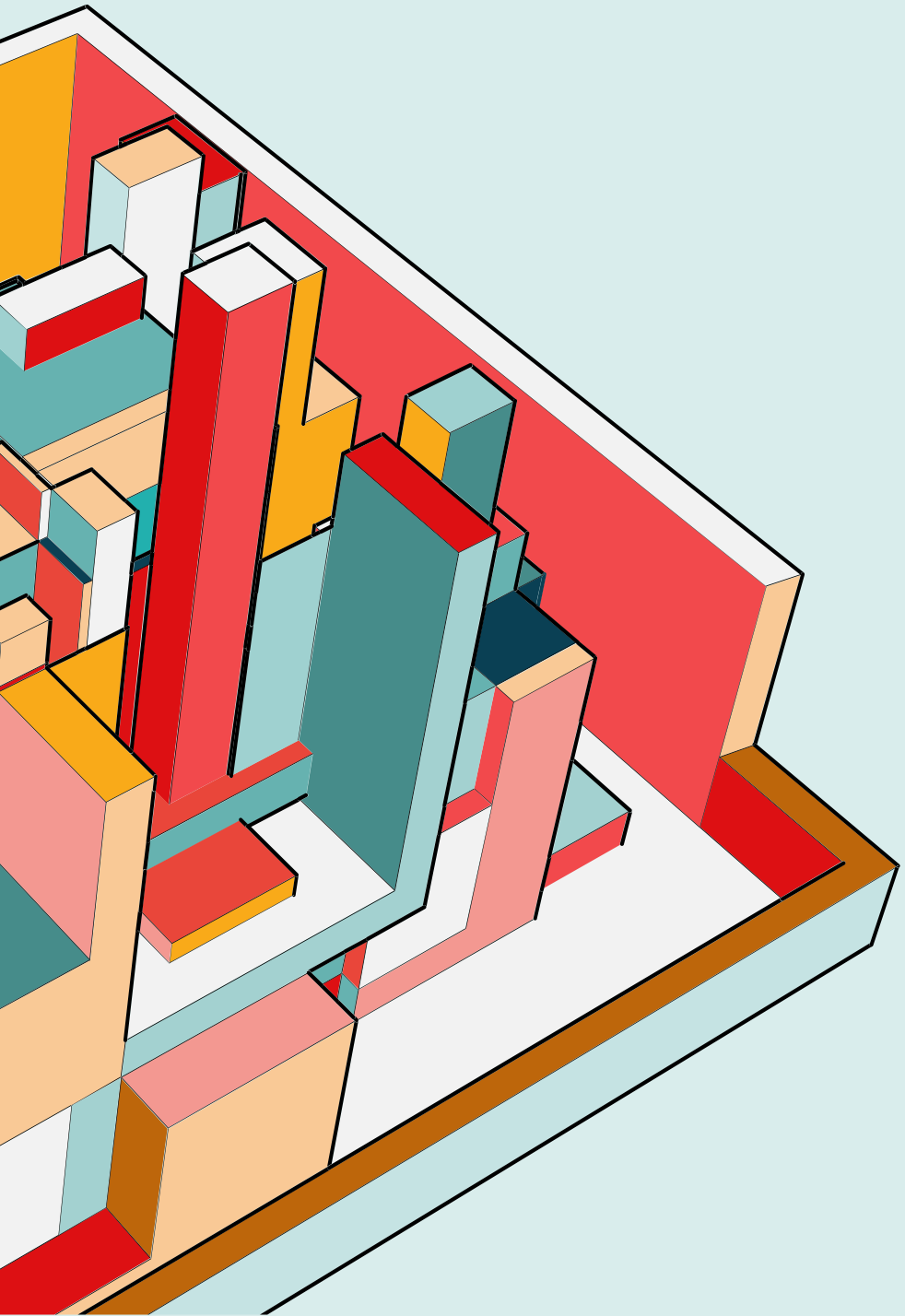
🔍 **Пример функционального тестирования**: проверим, корректно ли работает функция вычисления квадратного корня. Введем число 9, ожидаемый результат — 3. Если результат соответствует ожидаемому, функция работает корректно.

⏱️ **Пример нефункционального тестирования**: оценим производительность программы, выполняющей сложные математические расчеты. Замерим время выполнения задачи и сравним с требуемым значением.

🤖 **Пример автоматизированного тестирования**: напишем скрипт, который будет автоматически проверять корректность работы функций программы путем сравнения ожидаемых результатов с фактическими.

✋ **Пример ручного тестирования**: проверим, насколько удобно и понятно пользователю интерфейс программы, выполнив все основные операции вручную.

👥 **Пример тестирования с использованием группы пользователей**: пригласим группу пользователей для тестирования новой версии сайта и соберем их отзывы и предложения по улучшению.

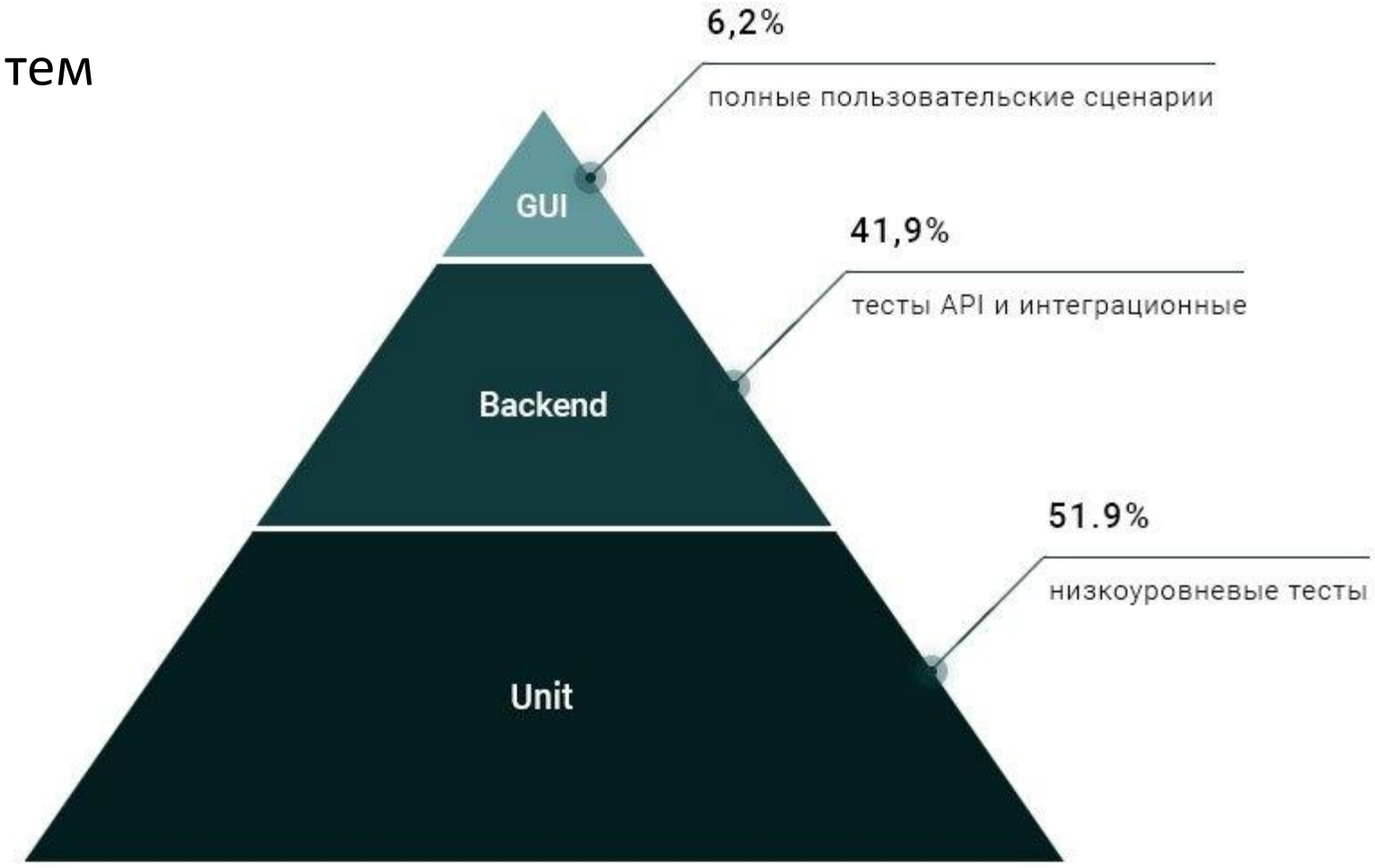


ЮНИТ-ТЕСТИРОВАНИЕ

Первый бастион
на борьбе с багами

ПИРАМИДА МАЙКА КОНА:

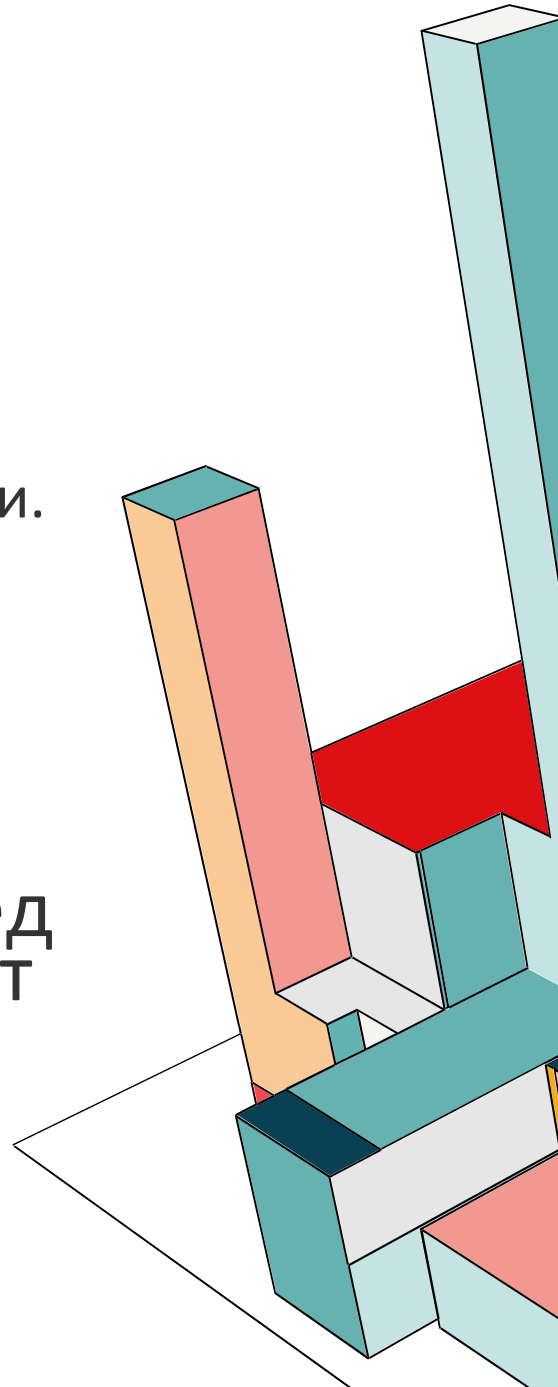
Чем выше тест в пирамиде, тем больше частей программы он затрагивает.



НЕ НУЖНО ПИСАТЬ ТЕСТЫ, ЕСЛИ

- Вы делаете простой сайт-визитку.
- Вы занимаетесь рекламным сайтом с большим объемом статики.
- Вы делаете проект для выставки. Софт будет работать 1-2 дня

- Вы всегда пишете код без ошибок, обладаете идеальной памятью и даром предвидения. Ваш код настолько крут, что изменяет себя сам, вслед за требованиями клиента. Иногда код объясняет клиенту, что его требования — ~~гов~~ не нужно реализовывать



ОН НЕ ТЕСТИРУЕТ

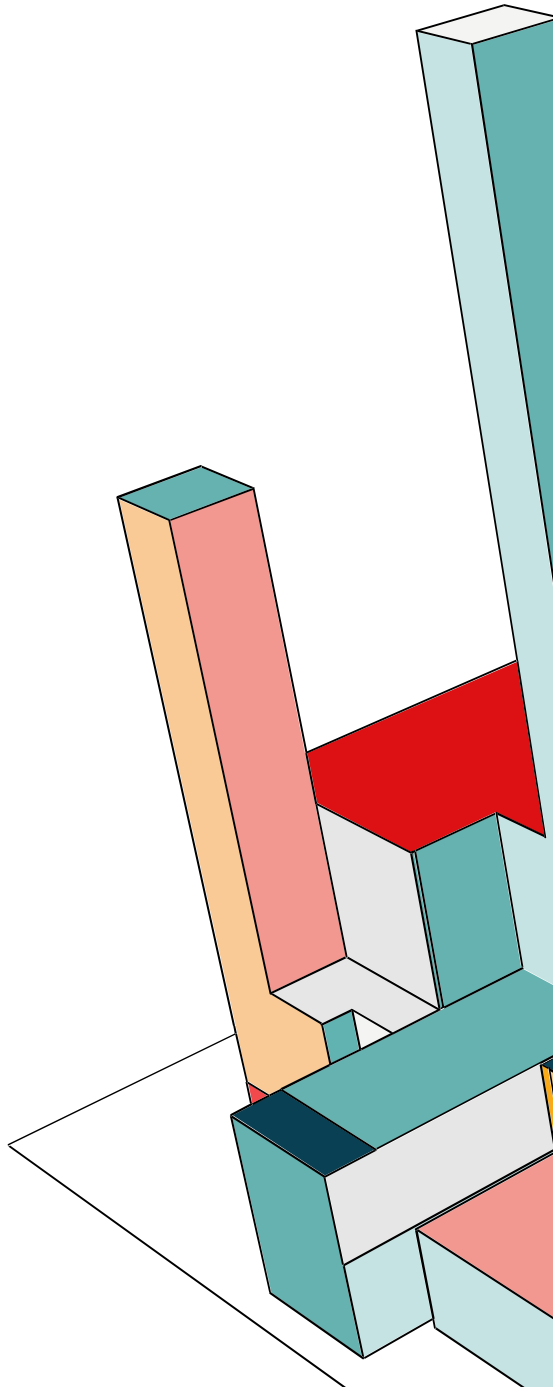
КЛАССИФИКАЦИЯ ЛЕГАСИ ПРОЕКТОВ

1. Без покрытия тестами.

2. С тестами, которые никто не запускает и не поддерживает.

3. С серьезным покрытием. Все тесты проходят.

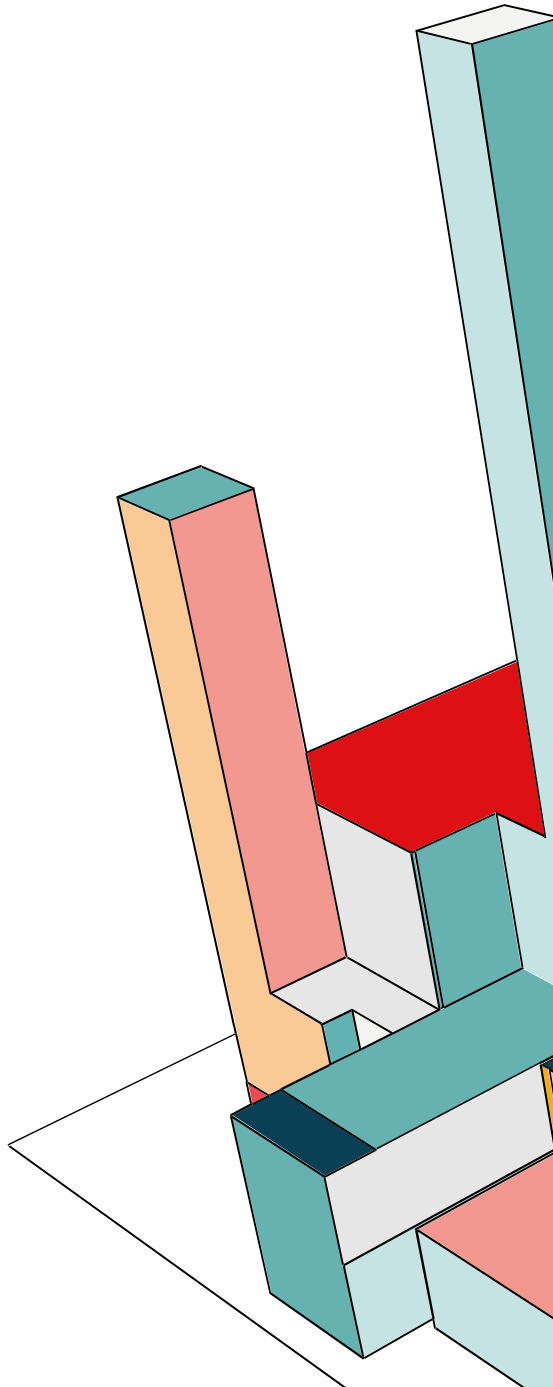
Я убежден, что бездумное написание тестов
не только не помогает, но вредит проекту.



ВАШИ ТЕСТЫ ДОЛЖНЫ:

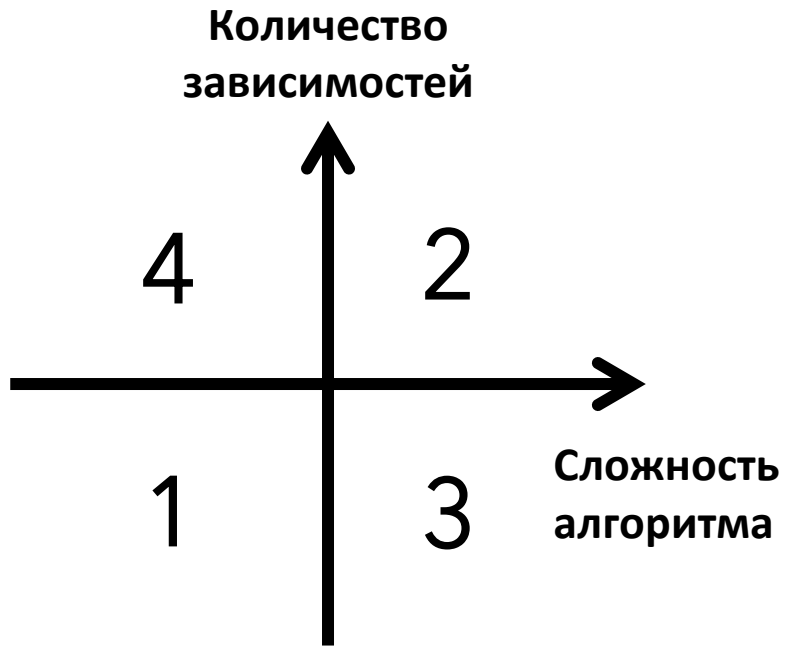
- Быть достоверными
- Не зависеть от окружения, на котором они выполняются
- Легко поддерживаться
- Легко читаться и быть простыми для понимания (даже новый разработчик должен понять что именно тестируется)
- Соблюдать единую конвенцию именования
- Запускаться регулярно в автоматическом режиме

Чтобы достичь выполнения этих пунктов, нужны терпение и воля.



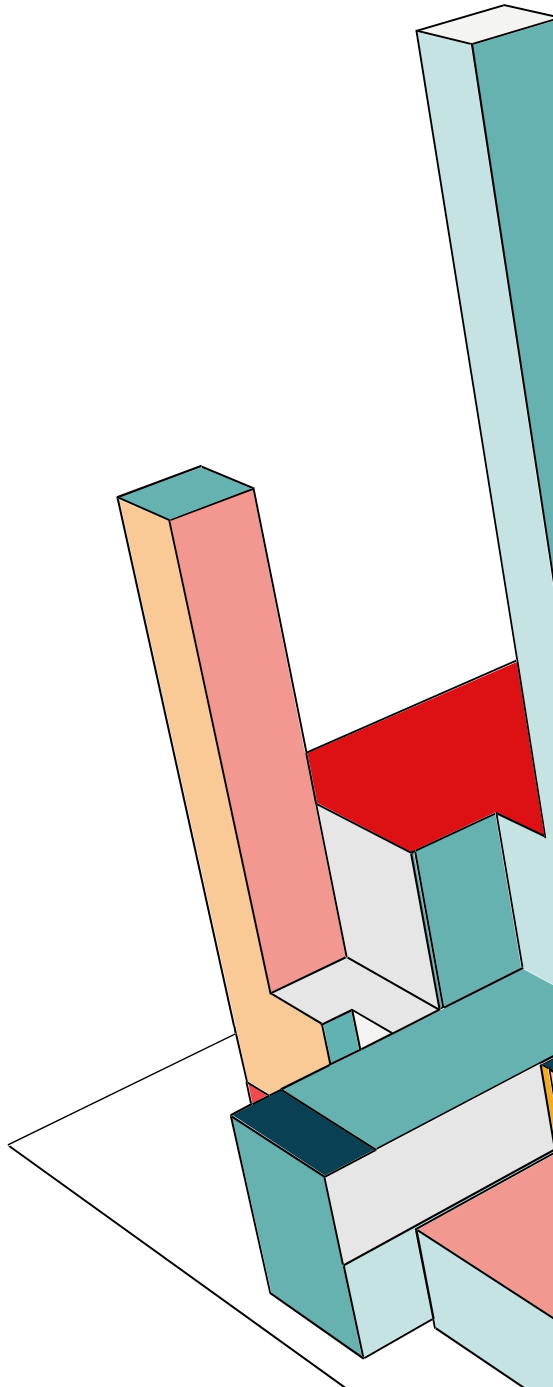
ЧТО ТЕСТИРОВАТЬ?

1. Простой код без зависимостей.
2. Сложный код с большим количеством зависимостей.
3. Сложный код без зависимостей.
4. Не очень сложный код с зависимостями.



РАССМОТРИМ ЭКСТРЕМАЛЬНЫЕ СЛУЧАИ:

1. **Простой код без зависимостей.** Скорее всего и так все ясно. Его можно не тестировать.
2. **Сложный код с большим количеством зависимостей.** Хм, если у вас есть такой код, тут пахнет God Object’ом и сильной связностью. Скорее всего, неплохо будет провести рефакторинг. Мы **не станем покрывать** этот код **юнит-тестами**, потому что перепишем его.

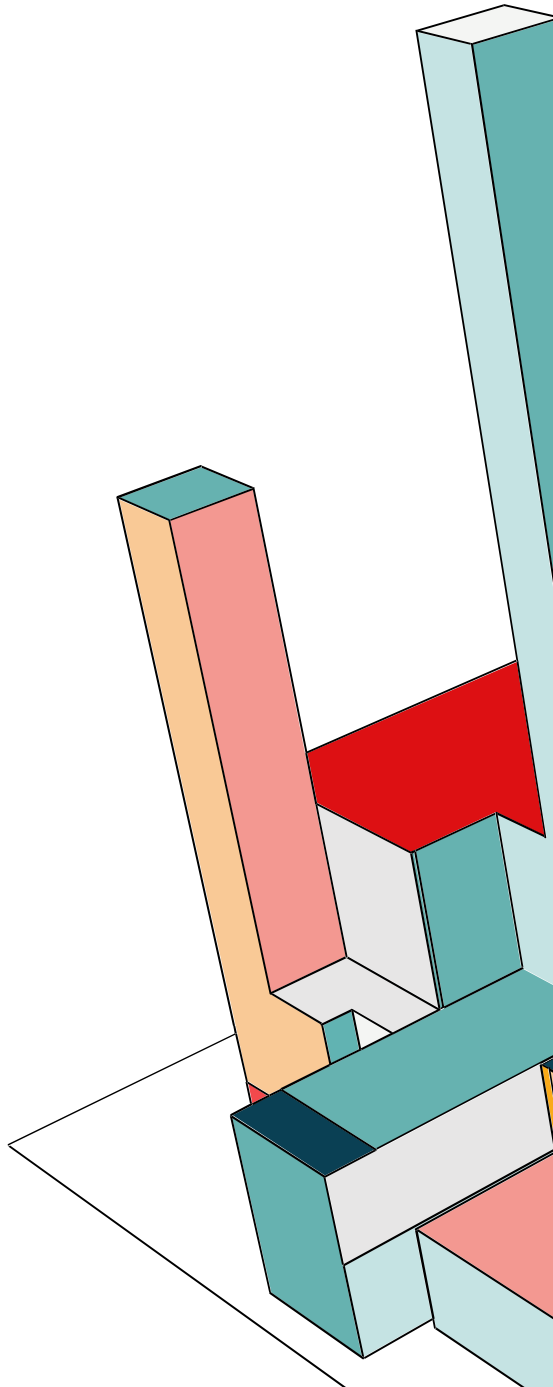


ЧТО У НАС ОСТАЕТСЯ:

3. Сложный код без зависимостей. Это некие алгоритмы или бизнес-логика. Отлично, это важные части системы, тестируем их.

4. Не очень сложный код с зависимостями. Этот код связывает между собой разные компоненты. Тесты важны, чтобы уточнить, как именно должно происходить взаимодействие.

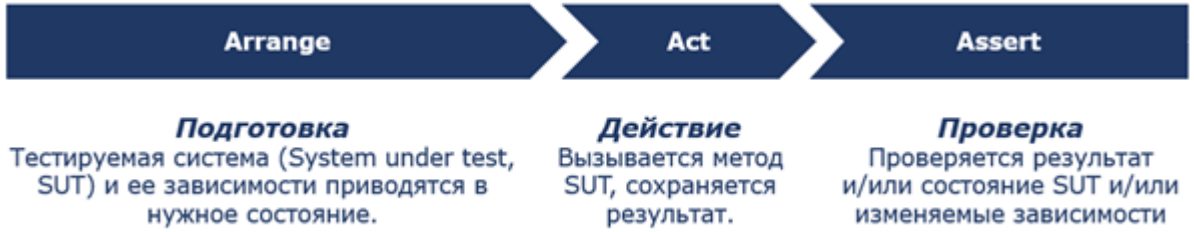
Причина потери Mars Climate Orbiter 23 сентября 1999 года заключалась в программно-человеческой ошибке: одно подразделение проекта считало «в дюймах», а другое – «в метрах», и прояснили это уже после потери аппарата. Результат мог быть другим, если бы команды протестировали «швы» приложения.



ПРИДЕРЖИВАЙТЕСЬ ЕДИНОГО СТИЛЯ

```
class CalculatorTests
{
    public void Sum_2Plus5_7Returned()
    {
        // arrange
        var calc = new Calculator();

        // act
        var res = calc.Sum(2,5);

        // assert
        Assert.AreEqual(7, res);
    }
}
```

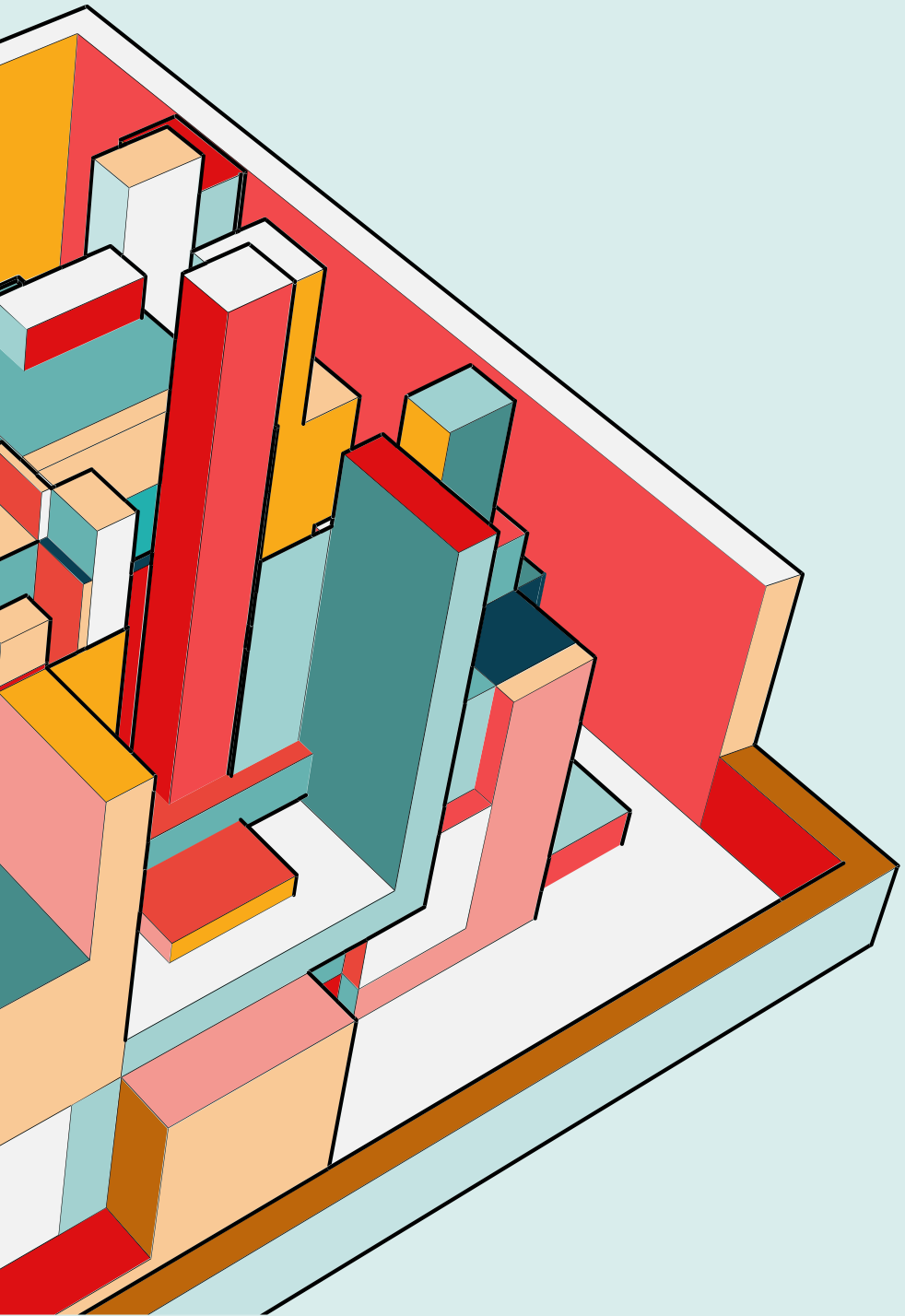
Отлично зарекомендовал себя подход




```
class CalculatorTests
{
    public void Sum_2Plus5_7Returned()
    {
        Assert.AreEqual(7, new Calculator().sum(2,5));
    }
}
```

ТЕСТИРУЙТЕ ОДНУ ВЕЩЬ ЗА ОДИН РАЗ

- **Каждый тест должен проверять только одну вещь.** Если процесс слишком сложен (например, покупка в интернет магазине), разделите его на несколько частей и протестируйте их отдельно.



СТИЛИ ЮНИТ-ТЕСТИРОВАНИЯ