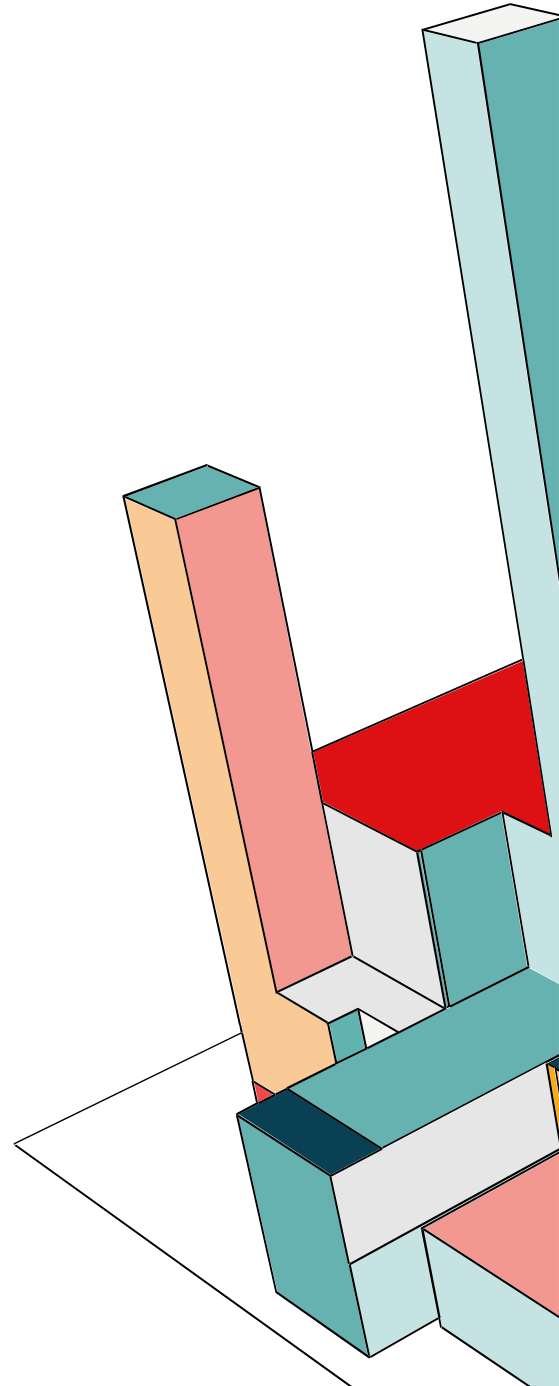




КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ПЛАН ЛЕКЦИИ № 4

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

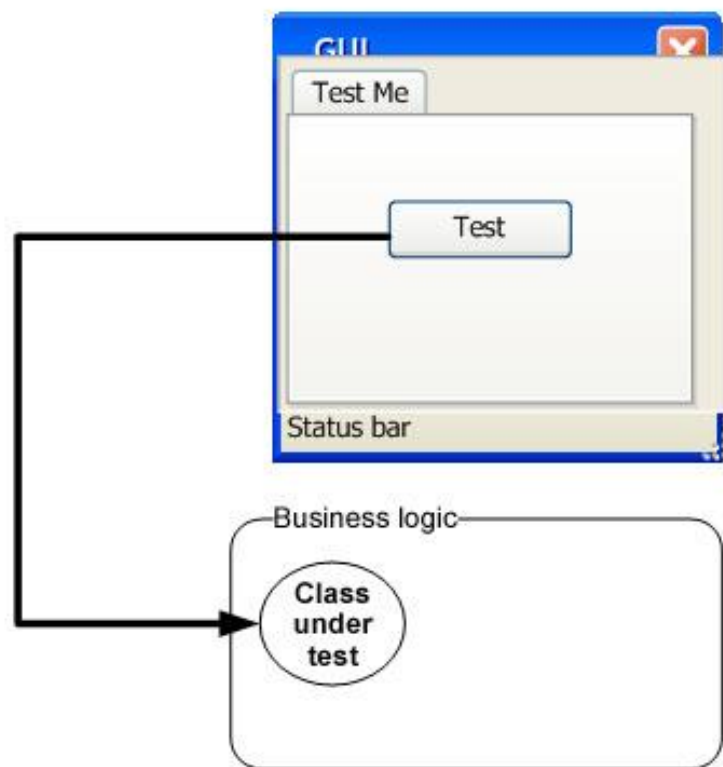


Тестирование показывает, соответствует ли ПО ожиданиям разработчиков.

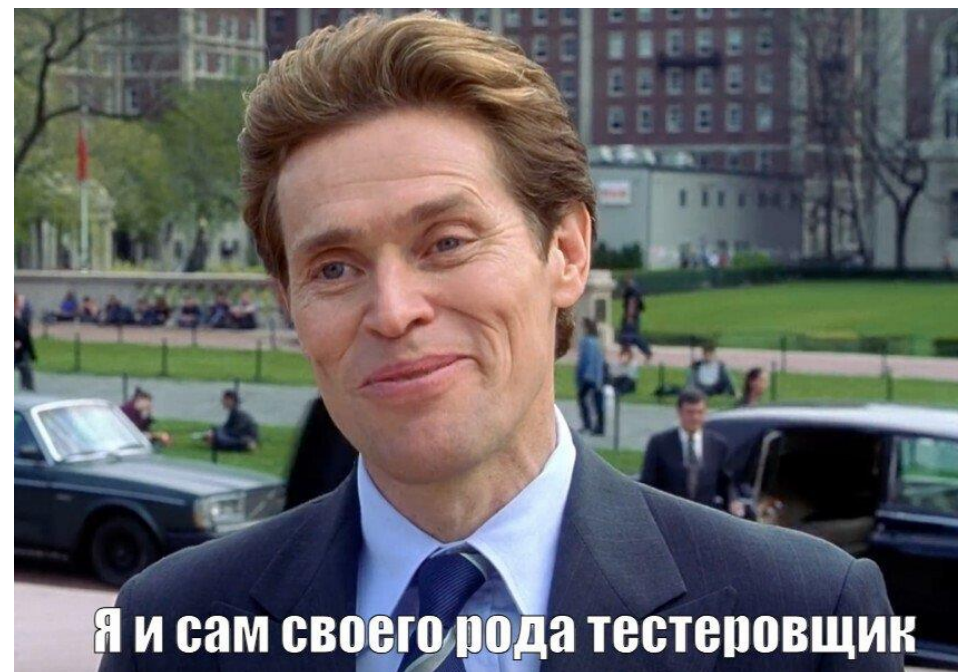
Тестирование проводят тестировщики — они отвечают за то, что продукт соответствует всем заданным требованиям.



ВСЕ МЫ НЕМНОГО... ТЕСТЕРЫ



Добавляете кнопку и проверяете, соответствует ли полученный результат вашим ожиданиям



ЭТАПЫ РАЗВИТИЯ ТЕСТИРОВАНИЯ

2000-е

оптимизация
бизнес-
технологий

1990-е

обеспечение
качества

1980-е

предупреждение
дефектов

1970-е

поиск дефектов

1960-е

исчерпывающее тестирование

1945

- первый случай, когда был найден баг

Photo # NH 96566-KN (Color) First Computer "Bug", 1947

0800 Antarm started
1000 " stopped - antarm ✓
1300 (032) MP-MC { 1.2700 9.037 847 025
033 PRO 2 2.130476415 9.037 846 995
conv 2.130476415 4.615925

Relays 6-2 in 033 failed special speed to
in relay 10,000 test.

1100 Started Cosine Tapc (Sine check)
1525 Started Multi-Adder Test.

1545 Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1630 Antarm started.
1700 closed down.

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)



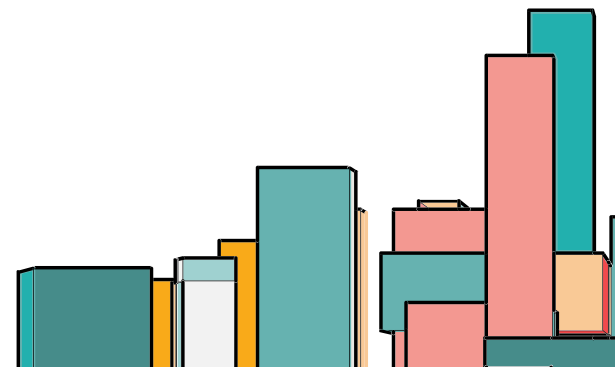
80-Е: ПРЕДУПРЕЖДЕНИЕ ДЕФЕКТОВ

- 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.



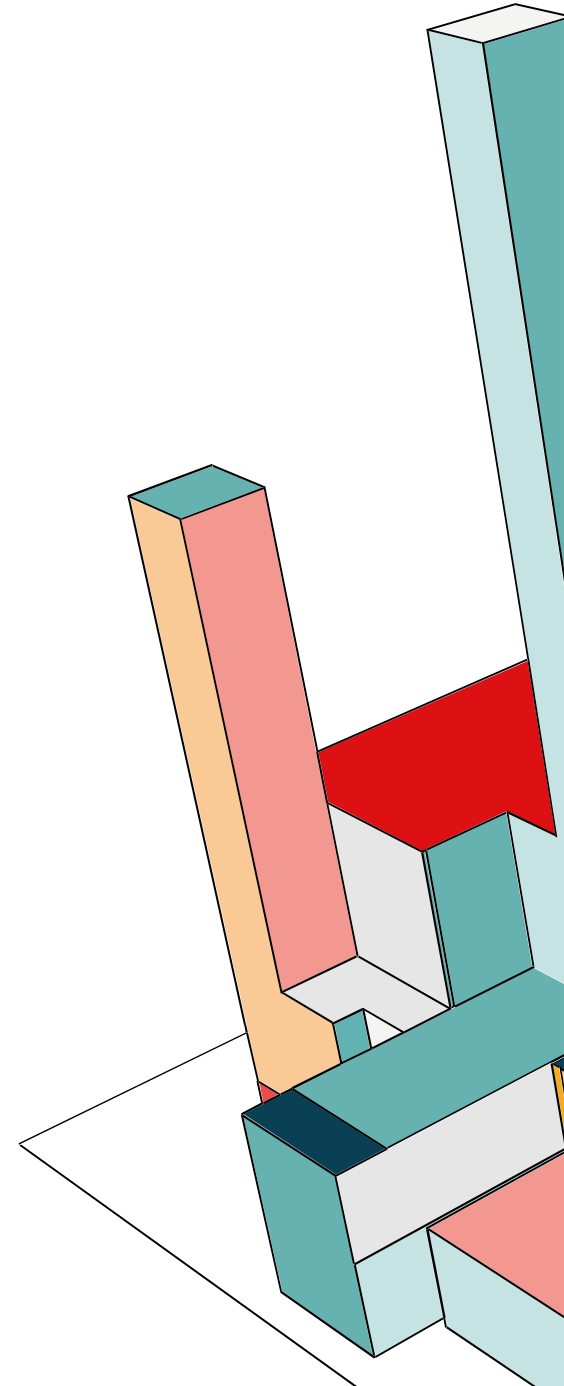
2000-Е: ТЕСТИРОВАНИЕ И БИЗНЕС

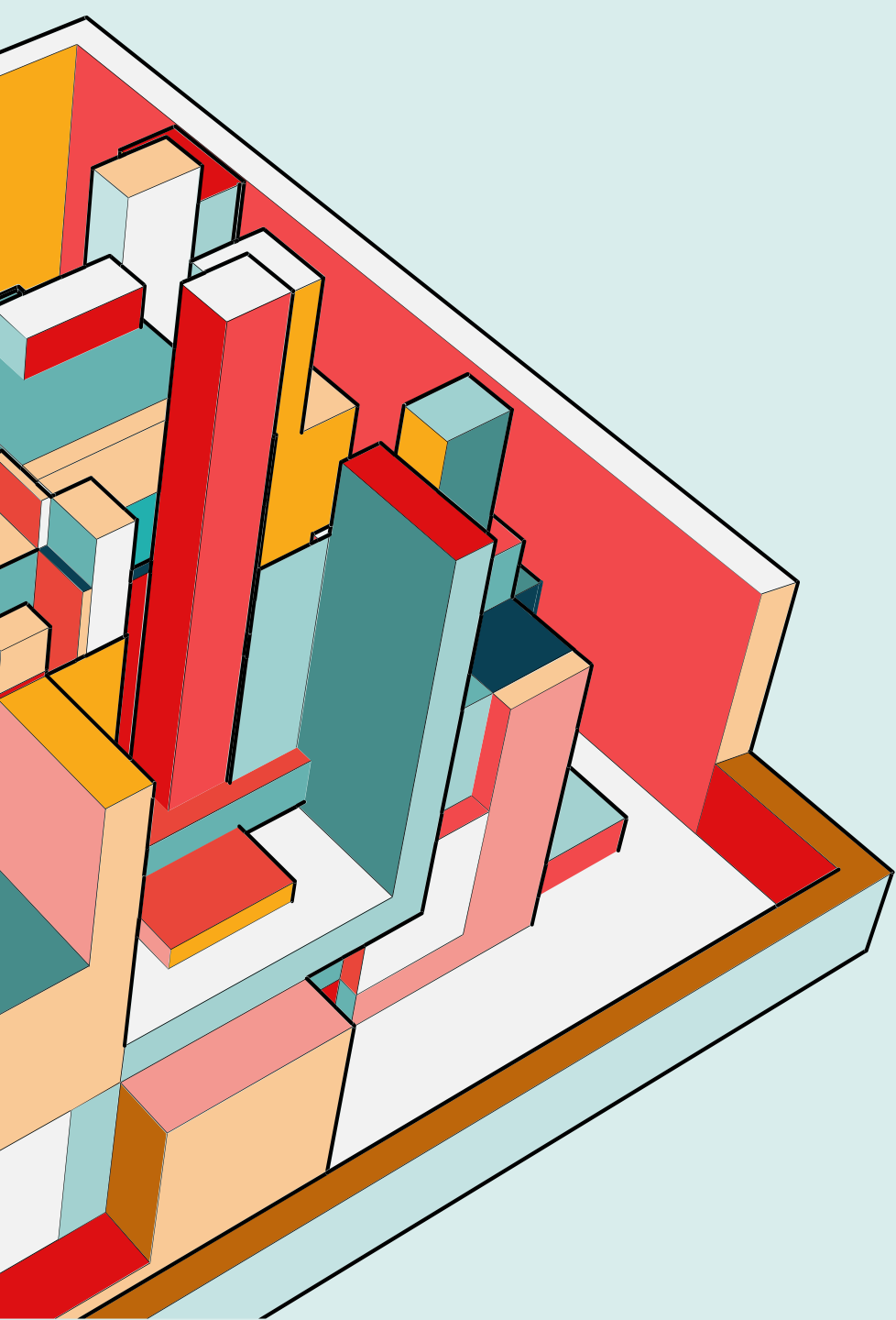
- В 2000-х появилось еще более широкое определение тестирования, когда в него было добавлено понятие «оптимизация бизнес-технологий»
- Основной подход заключается в оценке и максимизации значимости всех этапов жизненного цикла разработки ПО для достижения необходимого уровня качества, производительности, 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



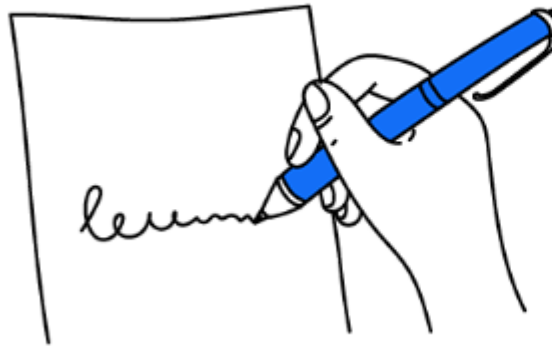


ВИДЫ ТЕСТИРОВАНИЯ

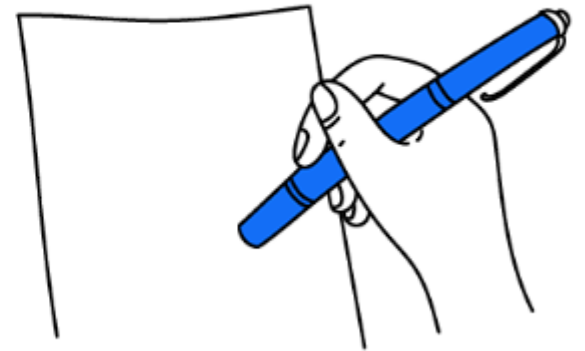
ПО ХАРАКТЕРУ СЦЕНАРИЕВ

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

Виды тестирования: по характеру сценариев



тестирование позитивных сценариев



тестирование негативных сценариев

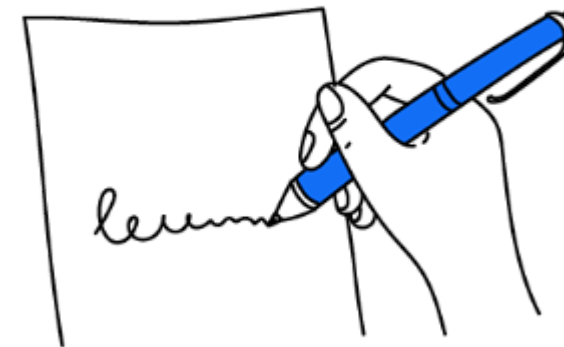
ПО КРИТЕРИЯМ ЗАПУСКА ПРОГРАММЫ ИЛИ КОДА

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

Виды тестирования: по критериям запуска программы или кода



статическое тестирование



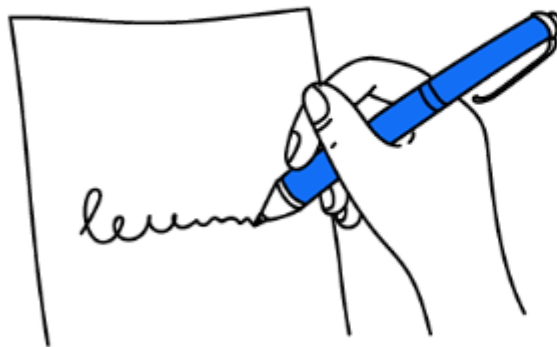
динамическое тестирование

ПО СТЕПЕНИ АВТОМАТИЗАЦИИ ТЕСТИРОВАНИЯ

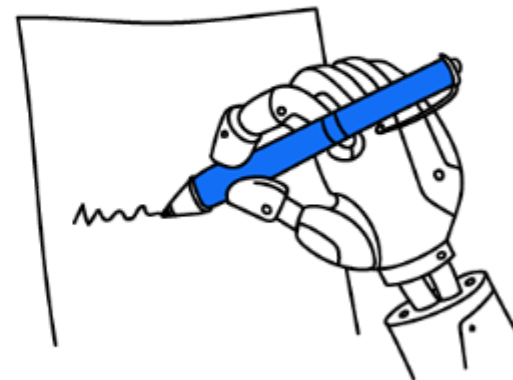
Ручное тестирование позволяет проверить различные аспекты программы: удобство использования, внешний интерфейс, а также воспроизводить нестандартные ситуации, которые может быть сложно автоматизировать.

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

Виды тестирования: по степени автоматизации



ручное тестирование



автоматизированное тестирование

ПО ОБЪЕКТАМ ТЕСТИРОВАНИЯ

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

Виды тестирования: по объектам тестирования



функциональное тестирование



тестирование интерфейса пользователя

ПО СТЕПЕНИ ЗНАНИЯ СИСТЕМЫ

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

Виды тестирования: по степени знания системы



тестирование черного ящика



тестирование белого ящика



тестирование серого ящика

ПО ВРЕМЕНИ ПРОВЕДЕНИЯ ТЕСТИРОВАНИЯ

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

Виды тестирования: по времени проведения





альфа-тестирование





бета-тестирование


ПРИМЕРЫ ТЕСТИРОВАНИЯ ПО

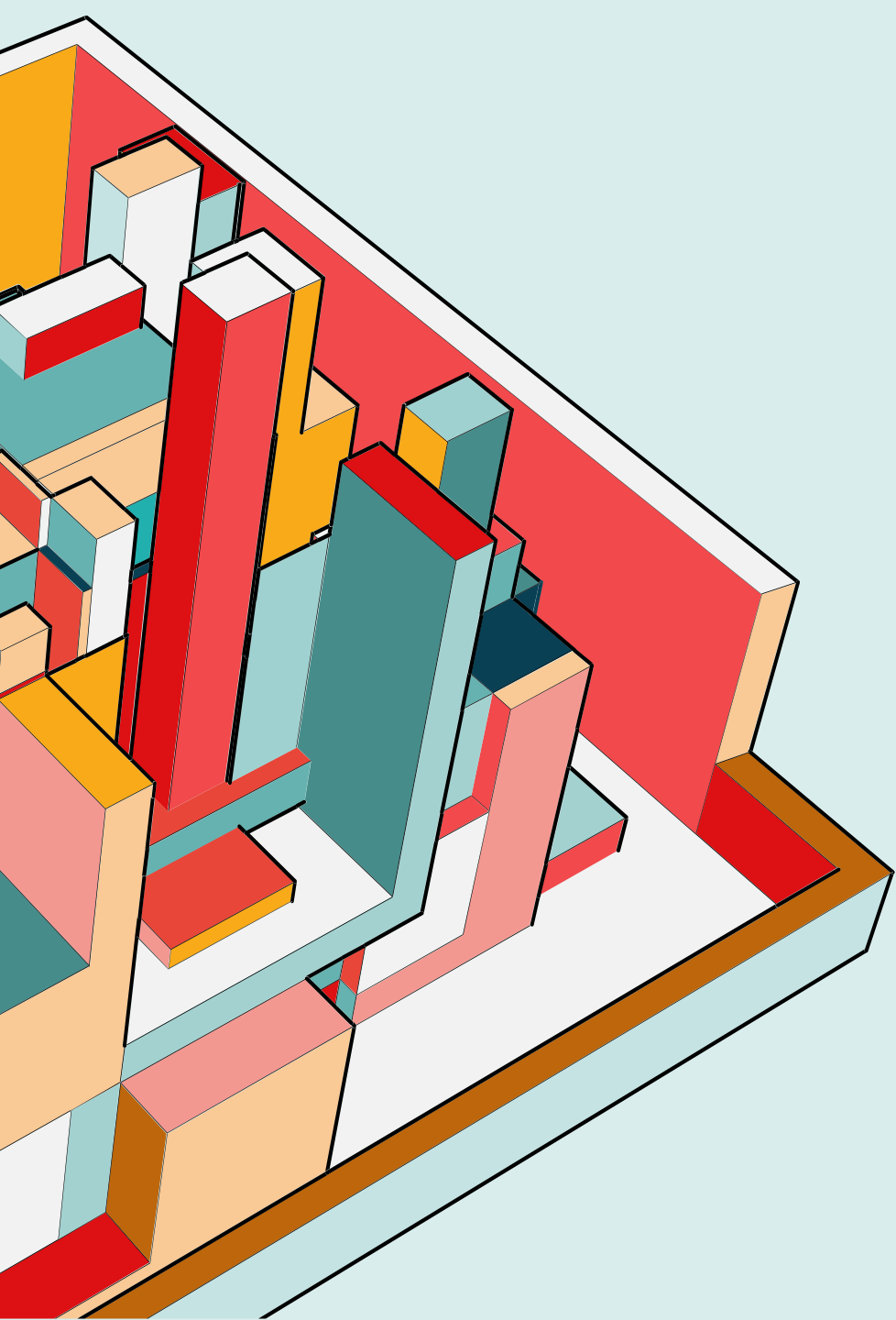
 **Пример функционального тестирования:** проверим, корректно ли работает функция вычисления квадратного корня. Введем число 9, ожидаемый результат — 3. Если результат соответствует ожидаемому, функция работает корректно.

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

 **Пример автоматизированного тестирования:** напомним 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.

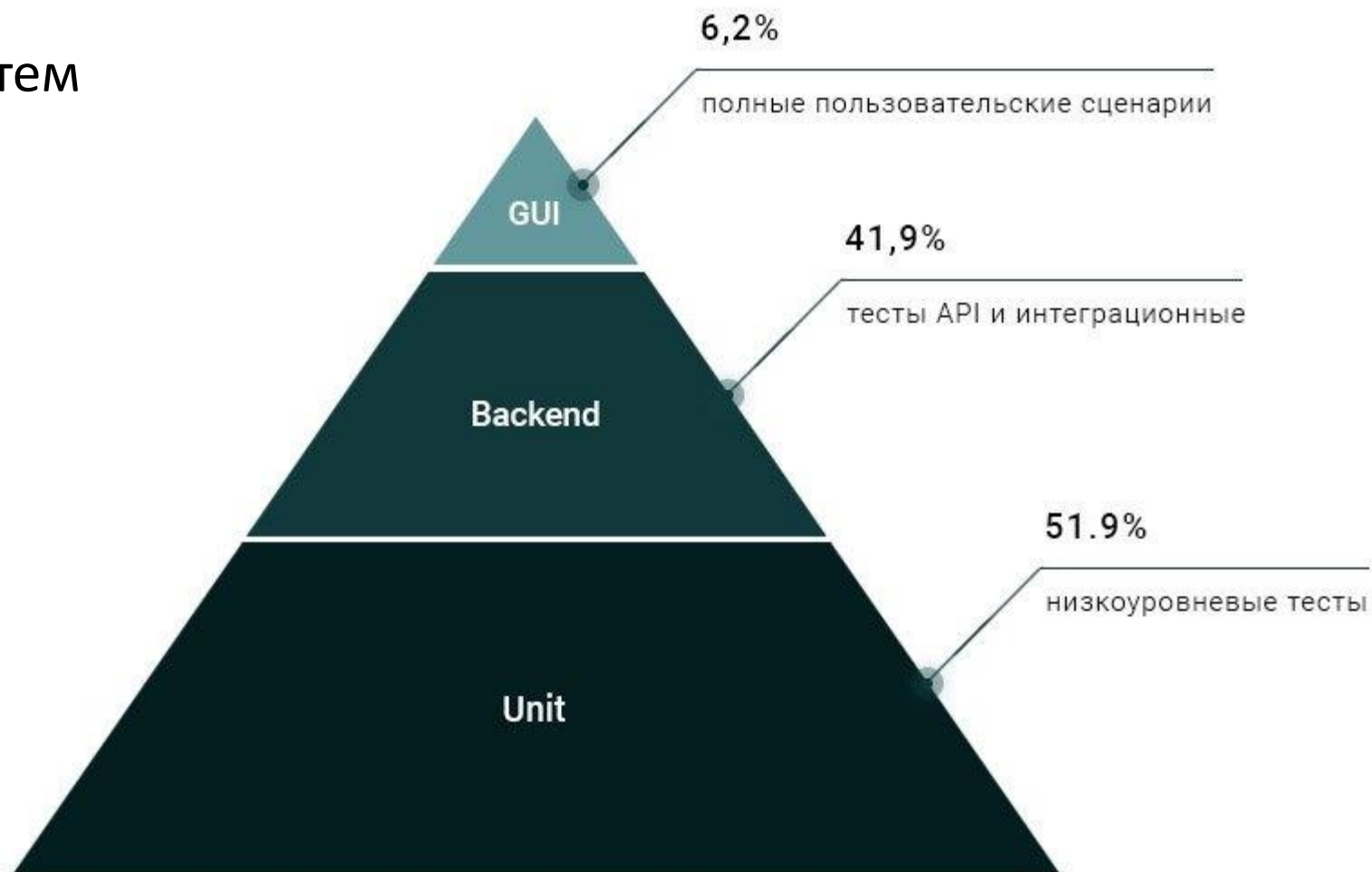


ЮНИТ- ТЕСТИРОВАНИЕ

Первый бастион
на борьбе с багами

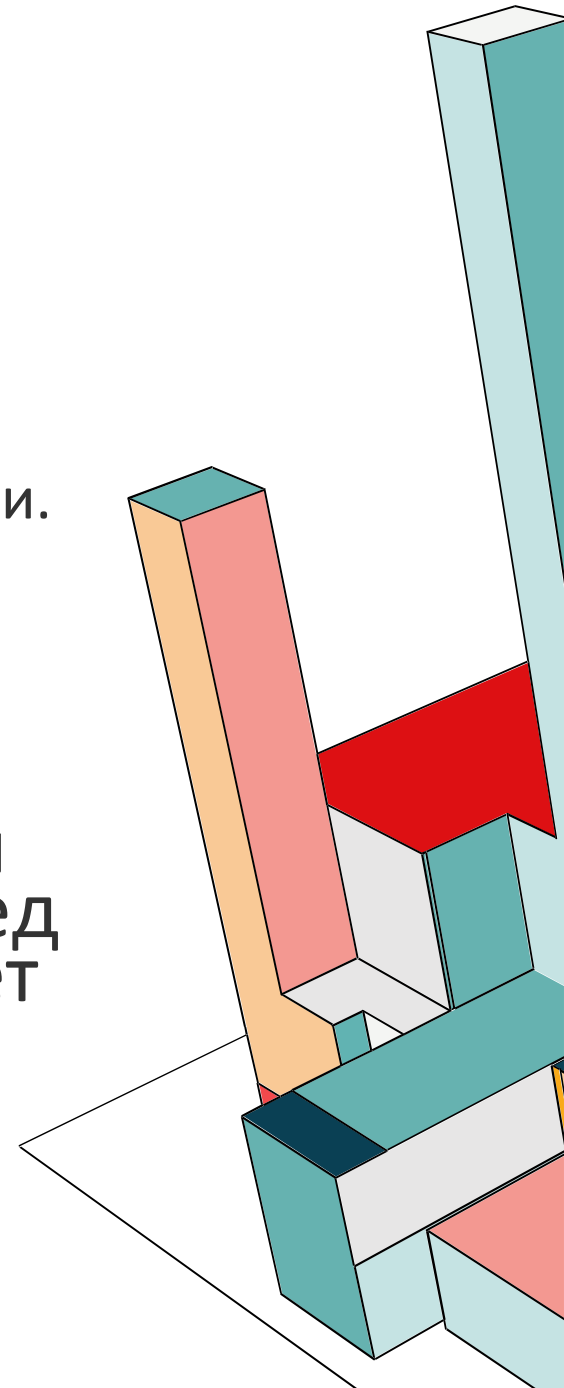
ПИРАМИДА МАЙКА КОНА:

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 статистики.
- 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 — `foo` не нужно реализовывать



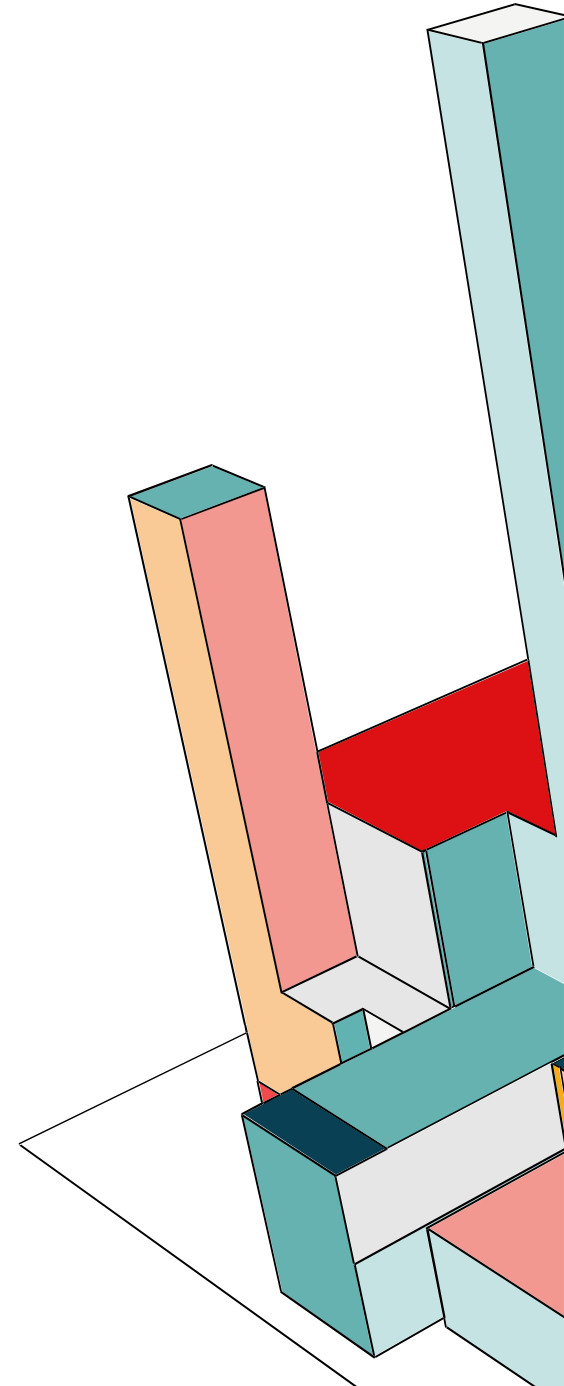
ОН НЕ ТЕСТИРУЕТ



КЛАССИФИКАЦИЯ ЛЕГАСИ ПРОЕКТОВ

1. Без покрытия тестами.
2. С тестами, которые никто не запускает и не поддерживает.
3. С серьезным покрытием. Все тесты проходят.

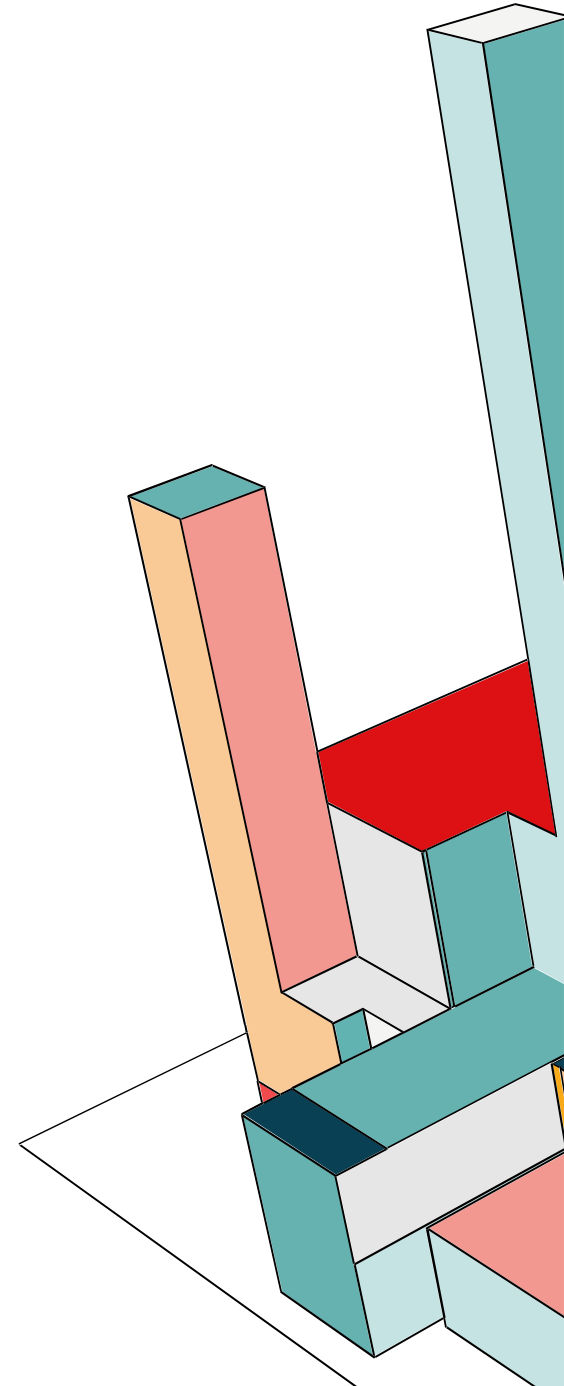
Я убежден, что бездумное написание тестов **не только не помогает, но вредит проекту.**



ВАШИ ТЕСТЫ ДОЛЖНЫ:

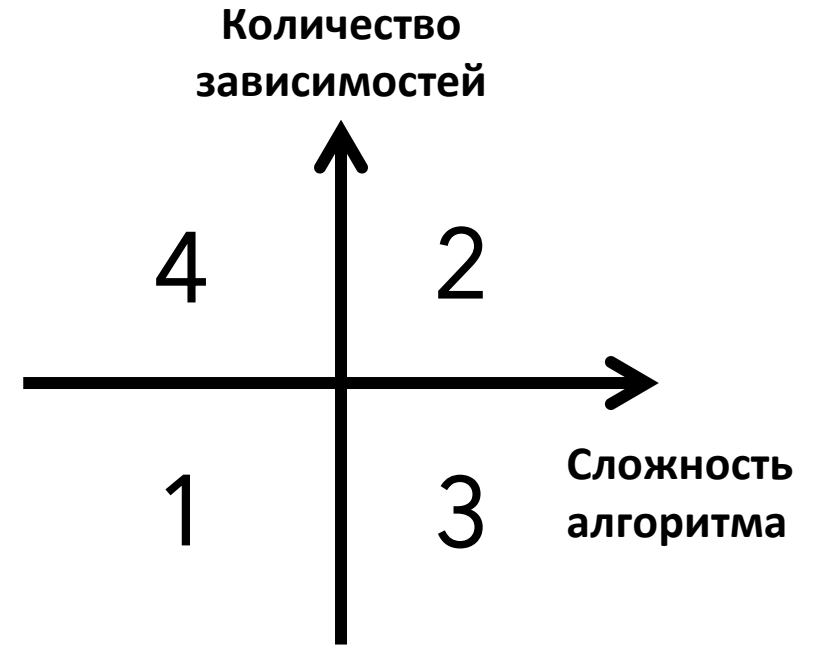
- Быть достоверными
- Не зависеть от окружения, на котором они выполняются
- Легко поддерживаться
- Легко читаться и быть простыми для понимания (даже новый разработчик должен понять что именно тестируется)
- Соблюдать единую конвенцию именования
- Запускаться регулярно в автоматическом режиме

Чтобы достичь выполнения этих пунктов, нужны терпение и воля.



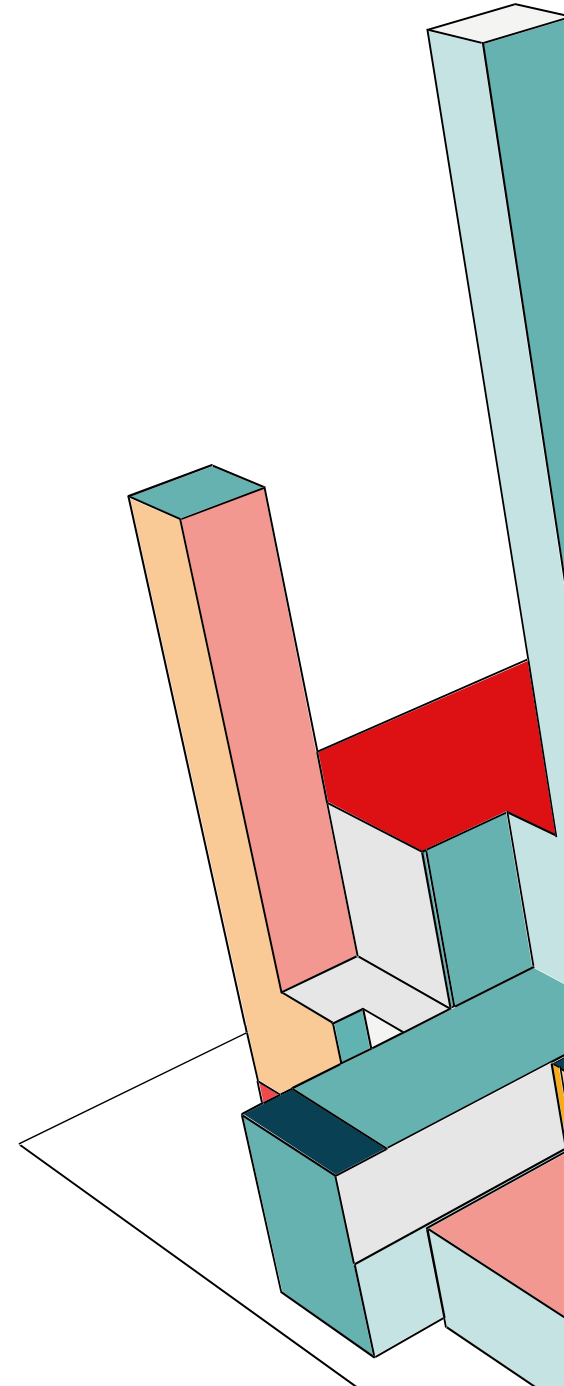
ЧТО ТЕСТИРОВАТЬ?

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.

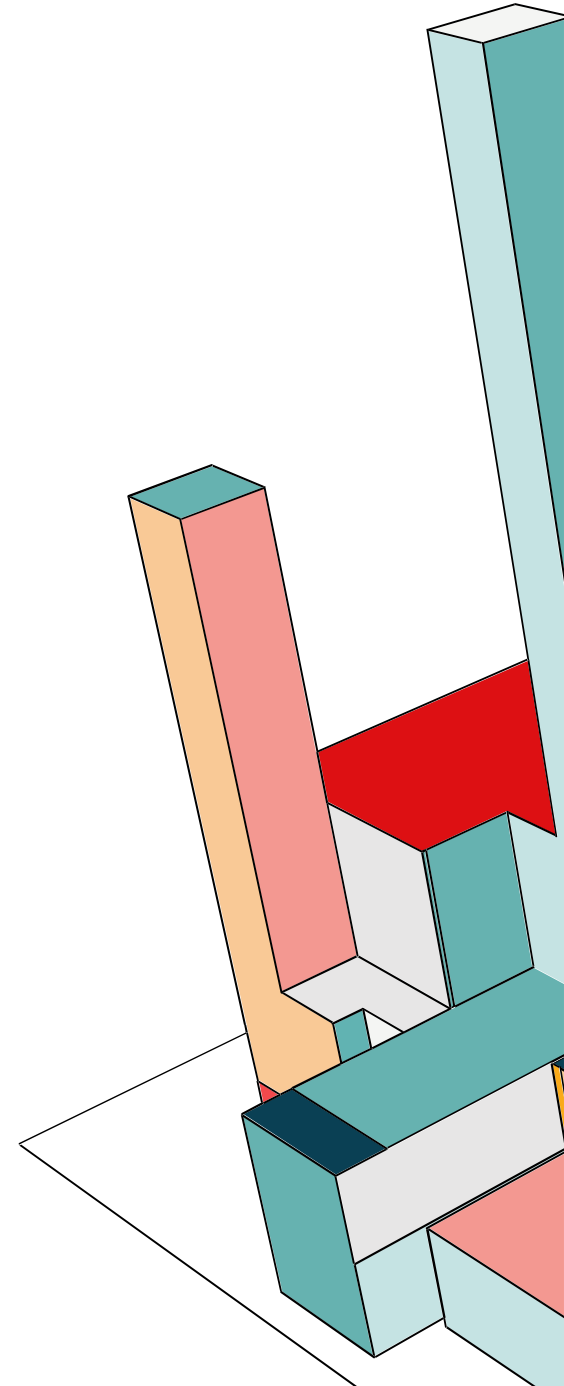


ЧТО У НАС ОСТАЕТСЯ:

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.



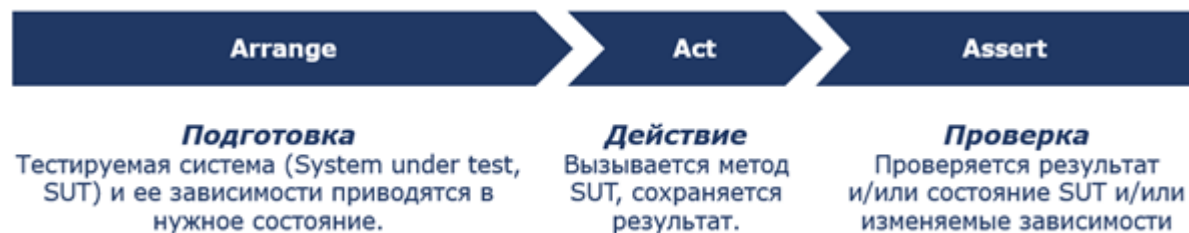
ПРИДЕРЖИВАЙТЕСЬ ЕДИНОГО СТИЛЯ

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

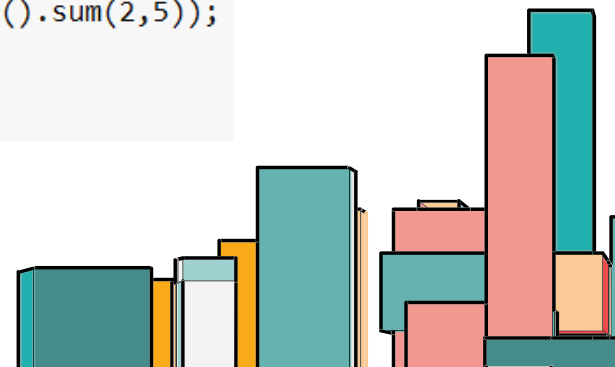
        // act
        var res = calc.Sum(2,5);

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

Отлично зарекомендовал себя подход

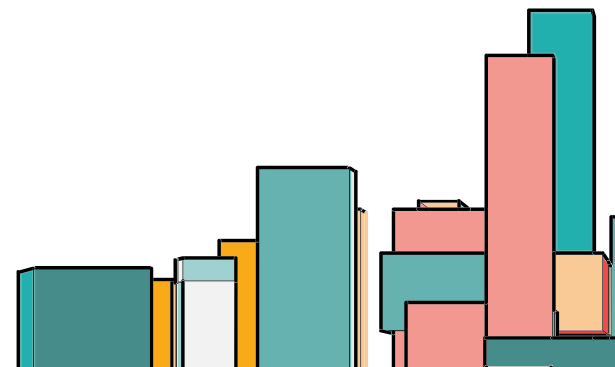


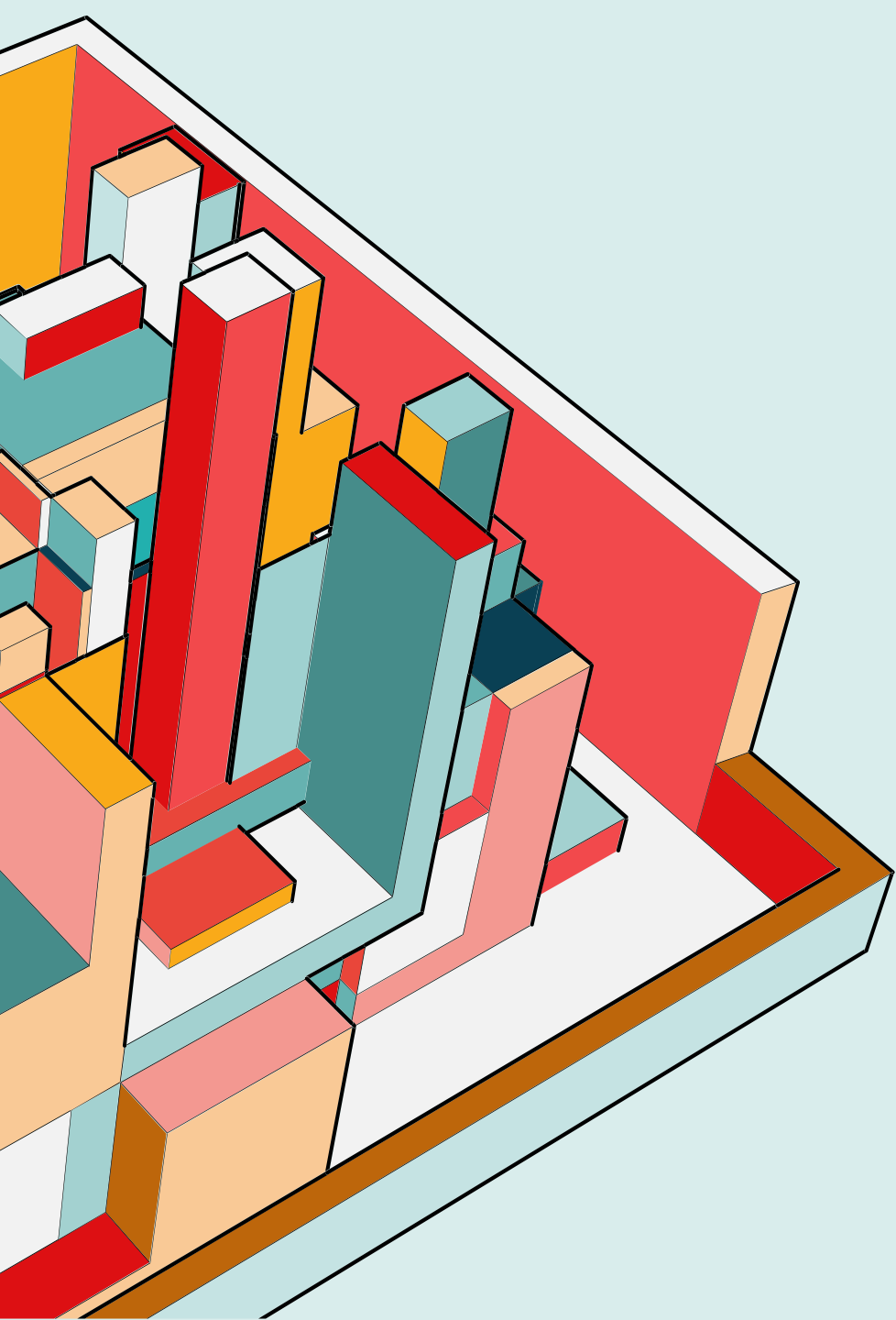
```
class CalculatorTests
{
    public void Sum_2Plus5_7Returned()
    {
        Assert.AreEqual(7, new Calculator().sum(2,5));
    }
}
```



ТЕСТИРУЙТЕ ОДНУ ВЕЩЬ ЗА ОДИН РАЗ

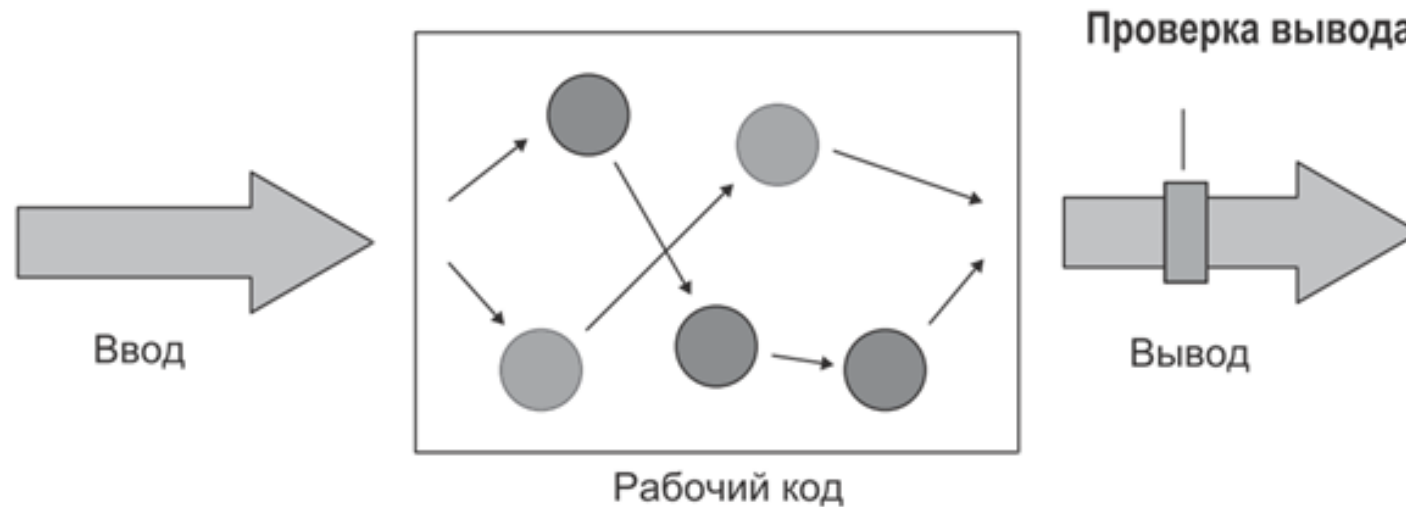
- Каждый тест должен проверять только одну вещь. Если процесс слишком сложен (например, покупка в интернет магазине), разделите его на несколько частей и протестируйте их отдельно.





СТИЛИ ЮНИТ- ТЕСТИРОВАНИЯ

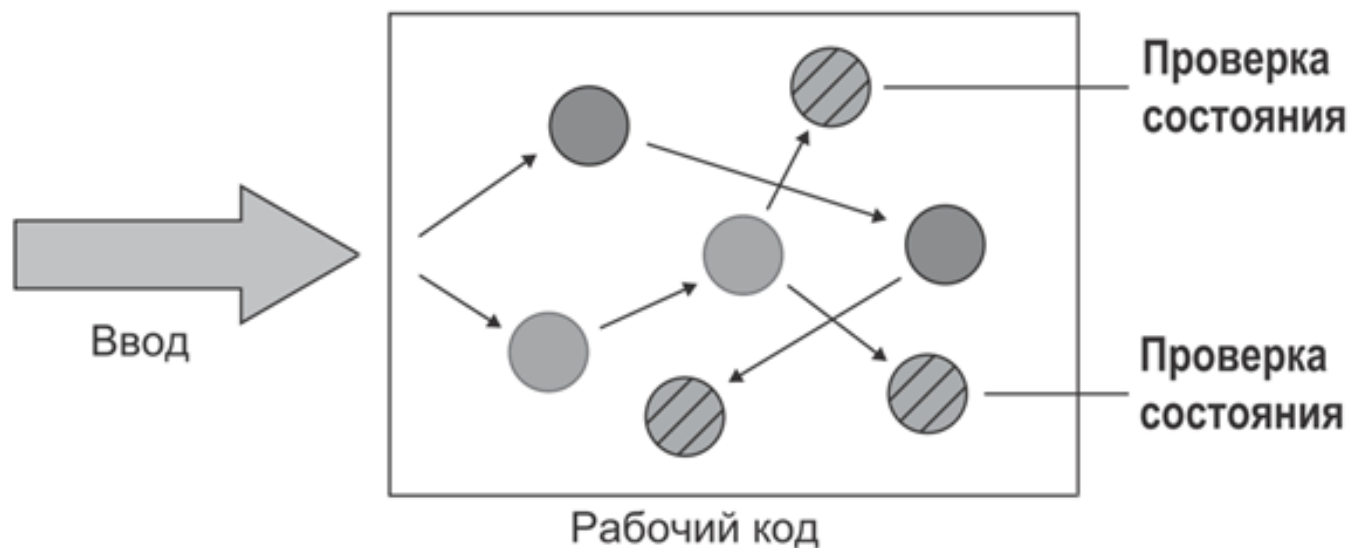
ПРОВЕРКА ВЫХОДНЫХ ДАННЫХ



Если значение, которое нам вернула тестируемая система и значение, которое мы ожидали, совпадают, значит, она работает корректно. Это очень простой стиль, но в то же время очень мощный. Увы, но этот стиль не всегда применим.



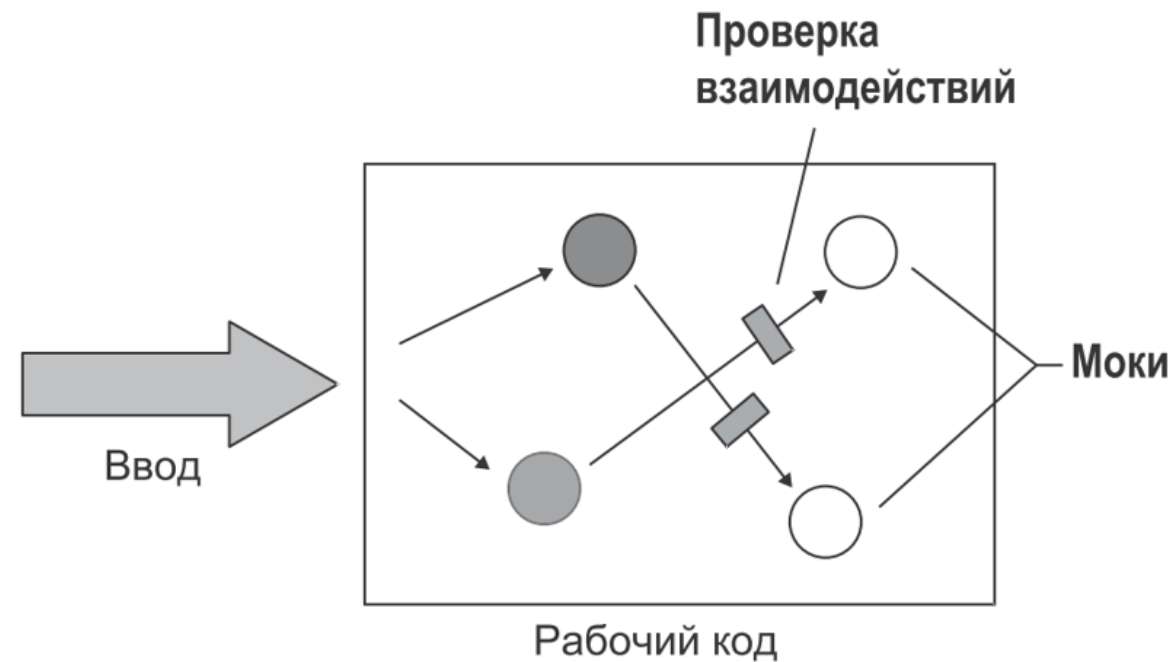
ПРОВЕРКА СОСТОЯНИЯ



Тест проверяет итоговое состояние тестируемой системы после выполнения операции, либо его изменяемых зависимостей.



ПРОВЕРКА ВЗАИМОДЕЙСТВИЯ



Такой стиль используется лондонской школой. Тест использует моки для проверки взаимодействий между тестируемой системой и ее изменяемыми зависимостями.



БОРЬБА С ЗАВИСИМОСТЯМИ

```
public class AccountManagementController : BaseAdministrationController
{
    #region Vars

    private readonly IOrderManager _orderManager;
    private readonly IAccountData _accountData;
    private readonly IUserManager _userManager;
    private readonly FilterParam _disabledAccountsFilter;

    #endregion

    public AccountManagementController()
    {
        _oms = OrderManagerFactory.GetOrderManager();
        _accountData = _orderManager.GetComponent<IAccountData>();
        _userManager = UserManagerFactory.Get();
        _disabledAccountsFilter = new FilterParam("Enabled", Expression.Eq, true);
    }
}
```

Фабрика в этом примере берет данные о конкретной реализации **AccountData**



ТЕСТИРУЕМ ТОЛЬКО КОНТРОЛЛЕР

Мы переписали класс и теперь можем подсунуть контроллеру другие реализации зависимостей, которые не станут лезть в базу.

```
public class AccountManagementController : BaseAdministrationController
{
    #region Vars

    private readonly IOrderManager _oms;
    private readonly IAccountData _accountData;
    private readonly IUserManager _userManager;
    private readonly FilterParam _disabledAccountsFilter;

    #endregion

    public AccountManagementController()
    {
        _oms = OrderManagerFactory.GetOrderManager();
        _accountData = _oms.GetComponent<IAccountData>();
        _userManager = UserManagerFactory.Get();
        _disabledAccountsFilter = new FilterParam("Enabled", Expression.Eq, true);
    }

    /// <summary>
    /// For testability
    /// </summary>
    /// <param name="accountData"></param>
    /// <param name="userManager"></param>
    public AccountManagementController(
        IAccountData accountData,
        IUserManager userManager)
    {
        _accountData = accountData;
        _userManager = userManager;
        _disabledAccountsFilter = new FilterParam("Enabled", Expression.Eq, true);
    }
}
```

ТЕСТОВЫЕ ДВОЙНИКИ



Помощниками в тестировании выступают «тестовые двойники»



ТЕСТИРОВАНИЕ СОСТОЯНИЯ И ТЕСТИРОВАНИЕ ПОВЕДЕНИЯ

Тестирование состояния

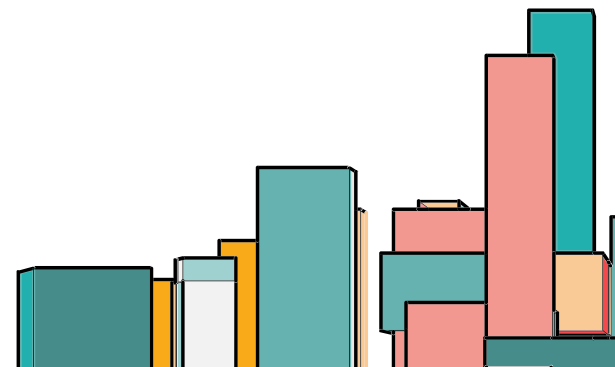
Запускаем цикл (12 часов). И через 12 часов проверяем, хорошо ли политы растения, достаточно ли воды, каково состояние почвы и т.д.

Тестирование взаимодействия

Установим датчики, которые будут засекать, когда полив начался и закончился, и сколько воды поступило из системы.

Стабы используются при тестировании состояния, а моки – взаимодействия.

Лучше использовать не более одного мока на тест. Иначе с высокой вероятностью вы нарушите принцип «тестировать только одну вещь».



[Test]

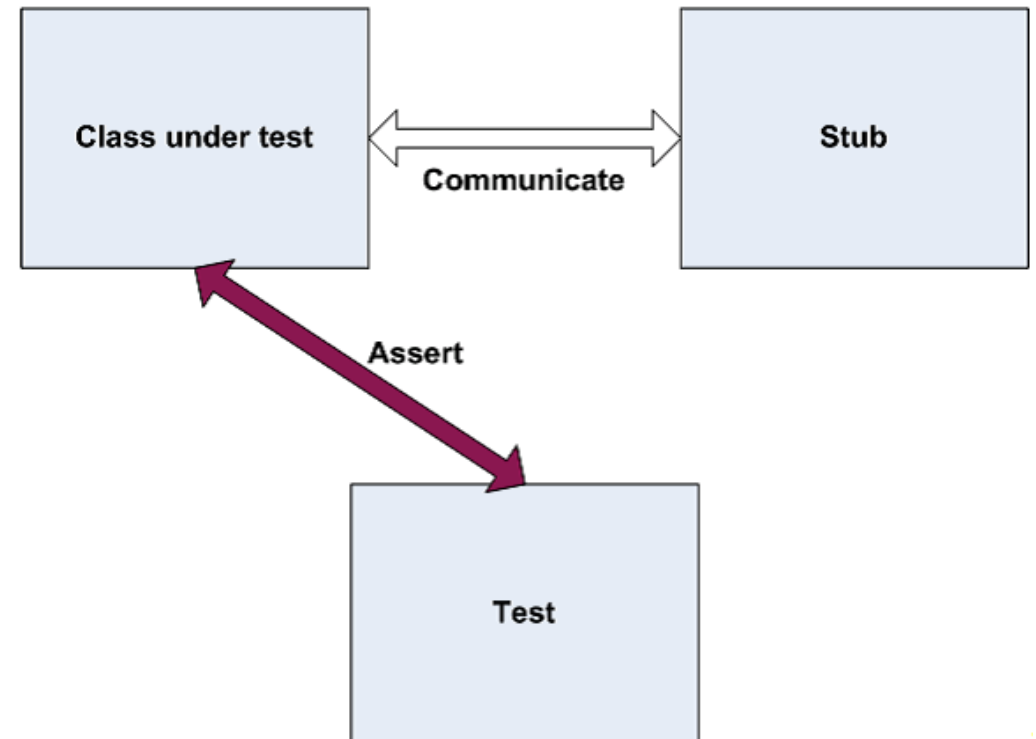
```
public void LogIn_ExistingUser_HashReturned()
{
    // Arrange
    OrderProcessor = Mock.Of<IOrderProcessor>();
    OrderData = Mock.Of<IOrderData>();
    LayoutManager = Mock.Of<ILayoutManager>();
    NewsProvider = Mock.Of<INewsProvider>();

    Service = new IosService(
        UserManager,
        AccountData,
        OrderProcessor,
        OrderData,
        LayoutManager,
        NewsProvider);

    // Act
    var hash = Service.LogIn("ValidUser", "Password");

    // Assert
    Assert.That(!string.IsNullOrEmpty(hash));
}
```

Заглушка (Stub) - имеет заранее подготовленные ответы на вызовы методов. Практически не имеет логики.

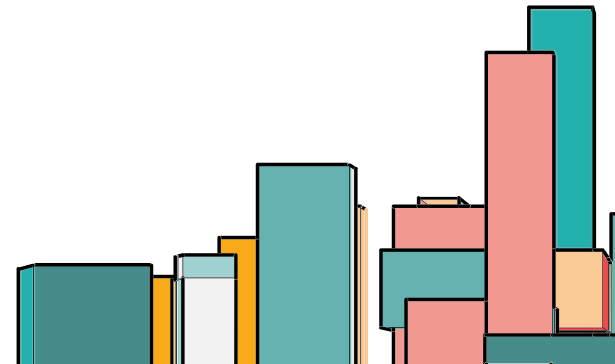
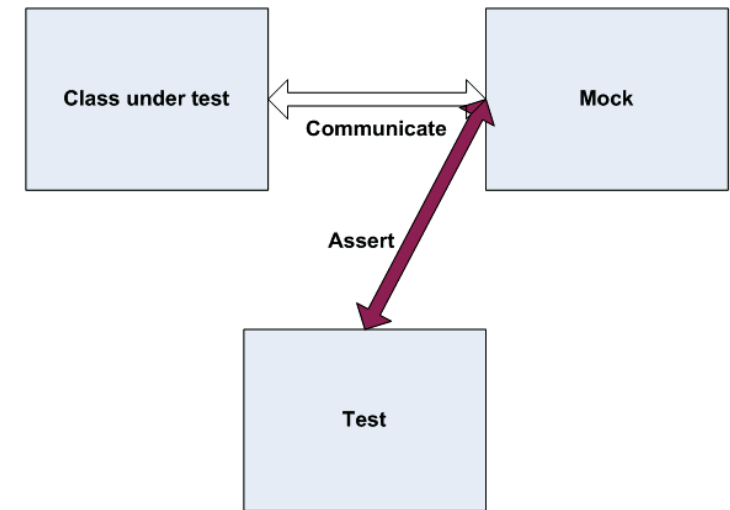


Мок (Mock) может иметь сложную логику ответов, зависящих от параметров вызовов и имеет другой, полезный для тестирования функционал.

```
[Test]
public void Create_AddAccountToSpecificUser_AccountCreatedAndAddedToUser()
{
    // Arrange
    var account = Mock.Of<AccountViewModel>();

    // Act
    _controller.Create(1, account);

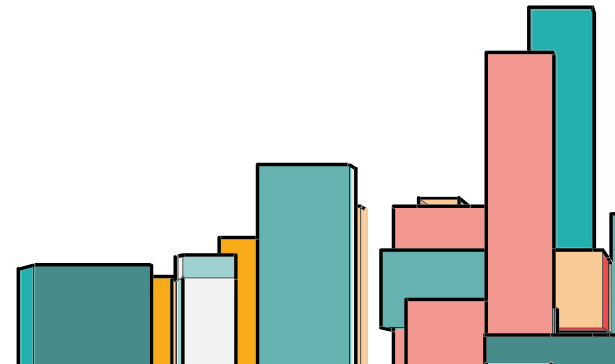
    // Assert
    _accountData.Verify(m => m.CreateAccount(It.IsAny<IAccount>()), Times.Exactly(1));
    _accountData.Verify(m => m.AddAccountToUser(It.IsAny<int>(), It.IsAny<int>()), Times.Once());
}
```



ТЕСТИРУЕМАЯ АРХИТЕКТУРА

Здесь мы отделались
«малой кровью».

```
public AccountManagementController(  
    IAccountData accountData,  
    IUserManager userManager)  
{  
    _accountData = accountData;  
    _userManager = userManager;  
    _disabledAccountsFilter = new FilterParam("Enabled", Expression.Eq, true);  
}
```



Переопределение локального фабричного метода

```
public class Calculator
{
    public double Multiply(double a, double b)
    {
        var multiplier = new Multiplier();
        return multiplier.Execute(a, b);
    }
}

public interface IArithmetic
{
    double Execute(double a, double b);
}

public class Multiplier : IArithmetic
{
    public double Execute(double a, double b)
    {
        return a * b;
    }
}
```

```
public class Calculator
{
    public double Multiply(double a, double b)
    {
        var multiplier = CreateMultiplier();
        return multiplier.Execute(a, b);
    }

    protected virtual IArithmetic CreateMultiplier()
    {
        var multiplier = new Multiplier();
        return multiplier;
    }
}

public class CalculatorUnderTest : Calculator
{
    protected override IArithmetic CreateMultiplier()
    {
        return new FakeMultiplier();
    }
}

public class FakeMultiplier : IArithmetic
{
    public double Execute(double a, double b)
    {
        return 5;
    }
}
```

Если зависимости
инстанцируются прямо в коде
явным образом, то самый
простой путь – выделить
фабричный protected-метод
CreateObjectName() и
переопределить его в классе-
наследнике.

После этого тестируйте класс-
наследник, а не ваш
первоначально тестируемый
класс.



ТЕСТИРУЕМАЯ АРХИТЕКТУРА VS OOP

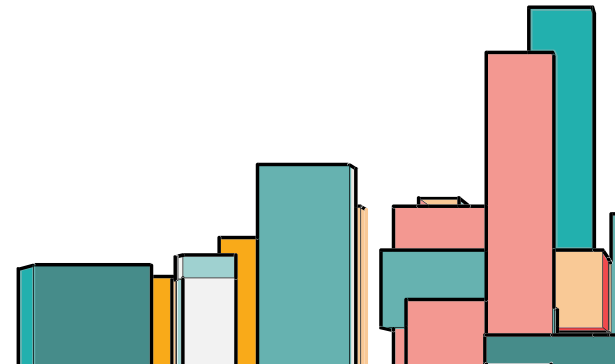
Серьезные требования к безопасности

Это значит, что у вас серьезная криптография, бинарники упакованы, и все обвешано сертификатами.

«Этот ваш тестируемый дизайн»
нарушает инкапсуляцию!

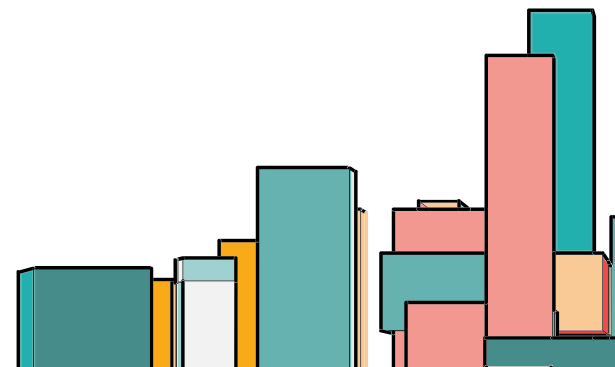
Производительность

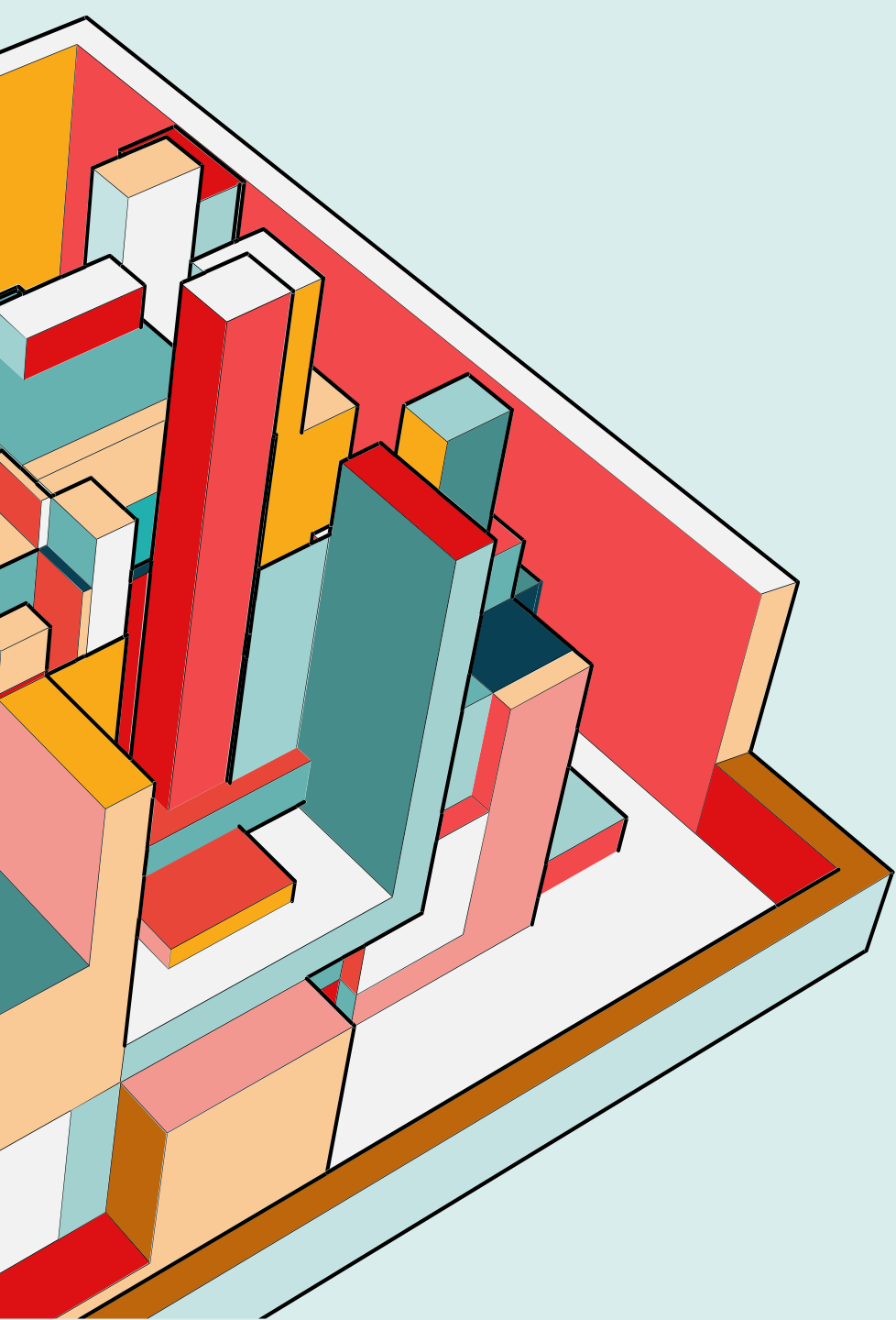
Существует ряд задач, когда архитектурой приходится жертвовать в угоду производительности, и для кого-то это становится поводом отказаться от тестирования



НЕСКОЛЬКО ПРИНЦИПОВ, КОТОРЫЕ ПОМОГАЮТ ПИСАТЬ ТЕСТИРУЕМЫЙ КОД

- Мыслите интерфейсами, а не классами, тогда вы всегда сможете легко подменять настоящие реализации подделками в тестовом коде
- Избегайте прямого инстанцирования объектов внутри методов с логикой. Используйте фабрики или *dependency injection*.
- Избегайте конструкторов, которые содержат логику: вам сложно будет это протестировать.
- Не относитесь к своим тестам как к второсортному коду. Многие начинающие разработчики ошибочно полагают, что DRY, KISS и все остальное – это для продакшна. А в тестах допустимо все. **Это не верно. Тесты – такой-же код.**





ЭФФЕКТИВНОСТЬ ЮНИТ-ТЕСТОВ

Первый бастион
на борьбе с багами

КАК ИЗМЕРИТЬ ЭФФЕКТИВНОСТЬ ЮНИТ ТЕСТОВ



Автоматического способа увы нет



Test coverage?



Хороший негативный индикатор



Плохой позитивный индикатор



Каждый тест необходимо оценивать
отдельно

КАК ИЗМЕРИТЬ ЭФФЕКТИВНОСТЬ ЮНИТ ТЕСТОВ

1. Защита от багов
(protection against bugs)

2. Устойчивость к
рефакторингу
(resilience to refactoring)

3. Скорость обратной связи
(fast feedback)

4. Простота поддержки
(maintainability)

СКОРОСТЬ ОБРАТНОЙ СВЯЗИ



Чем быстрее тест, тем меньше
времени тратится на устранение багов

ПРОСТОТА ПОДДЕРЖКИ

Простота поддержки



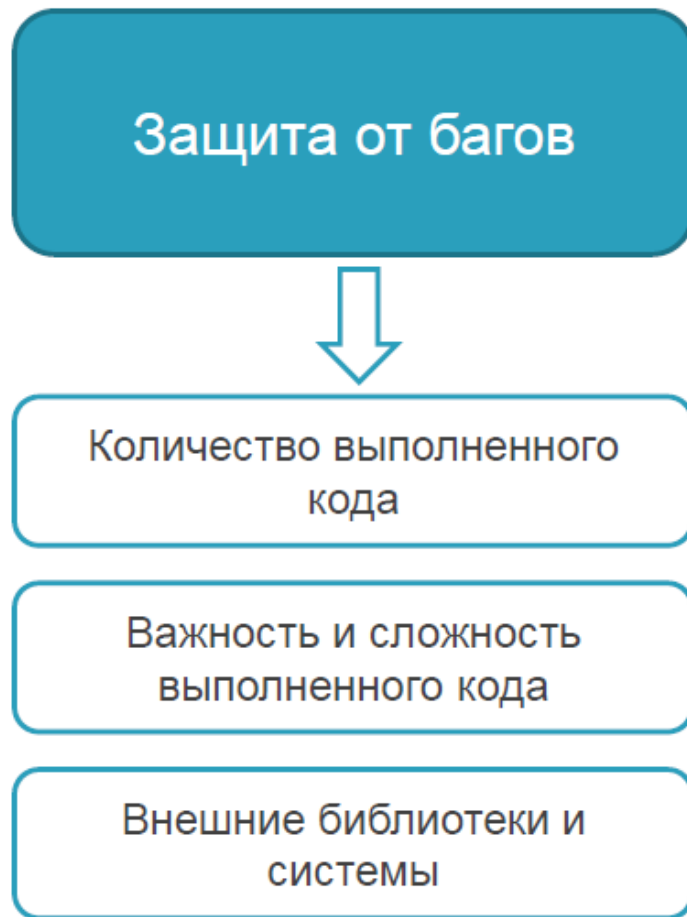
Насколько сложно понять
тест

- Размер и простота теста

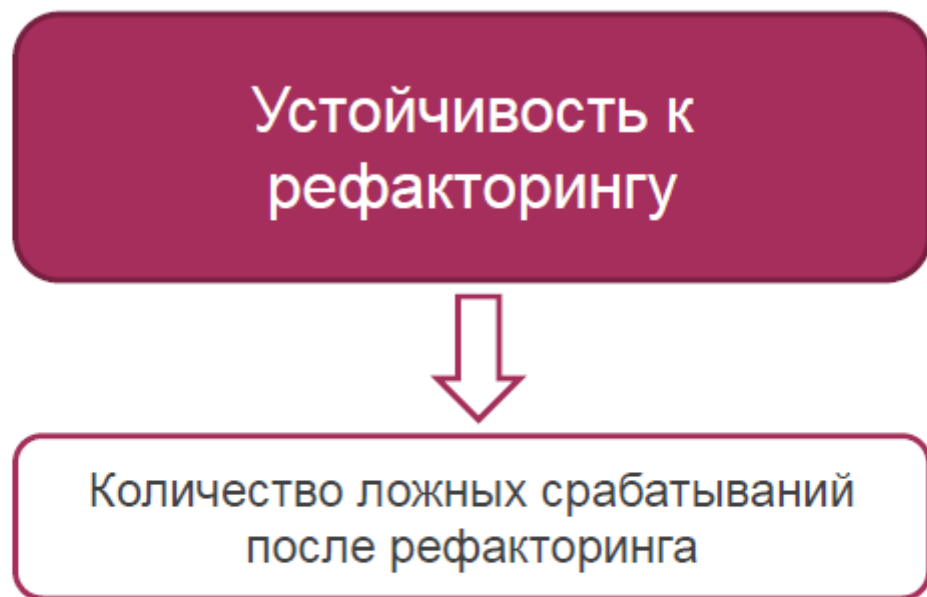
Насколько сложно этот тест
запускать

- Количество внешних зависимостей

ЗАЩИТА ОТ БАГОВ

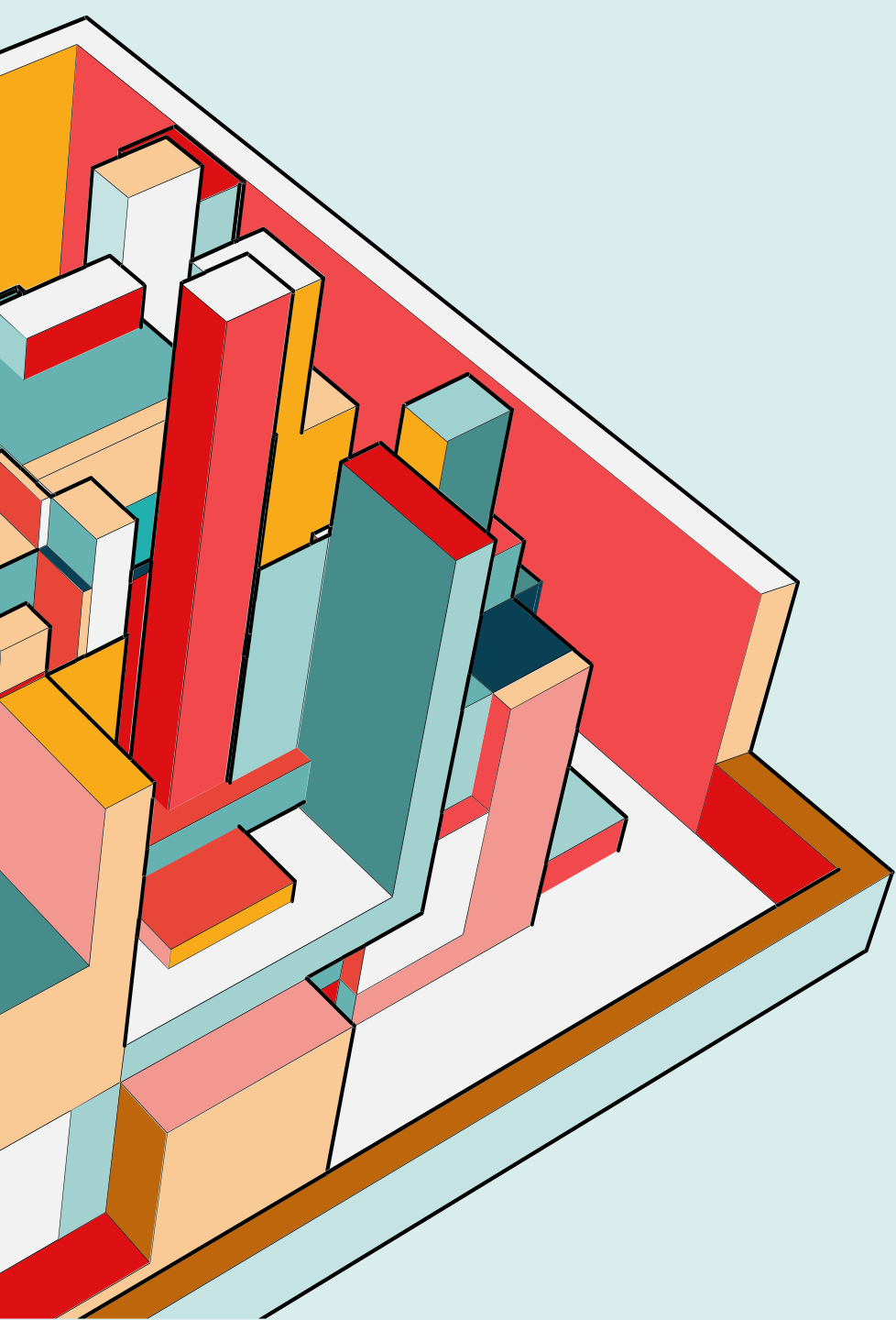


УСТОЙЧИВОСТЬ К РЕФАКТОРИНГУ



Ложные срабатывания возникают из-за привязки тестов к деталям реализации кода

Ложное срабатывание = False positive



ЛОЖНЫЕ СРАБАТЫВАНИЯ ЮНИТ-ТЕСТОВ

пример

```

public class FooterRenderer : IRenderer
{
    public string Render(Message message)
    {
        return $"<i>{message.Footer}</i>";
    }
}

public class BodyRenderer : IRenderer
{
    public string Render(Message message)
    {
        return $"<b>{message.Body}</b>";
    }
}

public class HeaderRenderer : IRenderer
{
    public string Render(Message message)
    {
        return $"<h1>{message.Header}</h1>";
    }
}

```

```

public class Message {
    public string Header { get; set; }
    public string Body { get; set; }
    public string Footer { get; set; }
}

public interface IRenderer {
    string Render(Message message);
}

public class MessageRenderer : IRenderer {
    public IReadOnlyList<IRenderer> SubRenderers { get; }

    public MessageRenderer() {
        SubRenderers = new List<IRenderer>
        {
            new HeaderRenderer(),
            new BodyRenderer(),
            new FooterRenderer()
        };
    }

    public string Render(Message message)
    {
        return SubRenderers
            .Select(x => x.Render(message))
            .Aggregate("", (str1, str2) => str1 + str2);
    }
}

```




```
public class Message {  
    public string Header { get; set; }  
    public string Body { get; set; }  
    public string Footer { get; set; }  
}
```

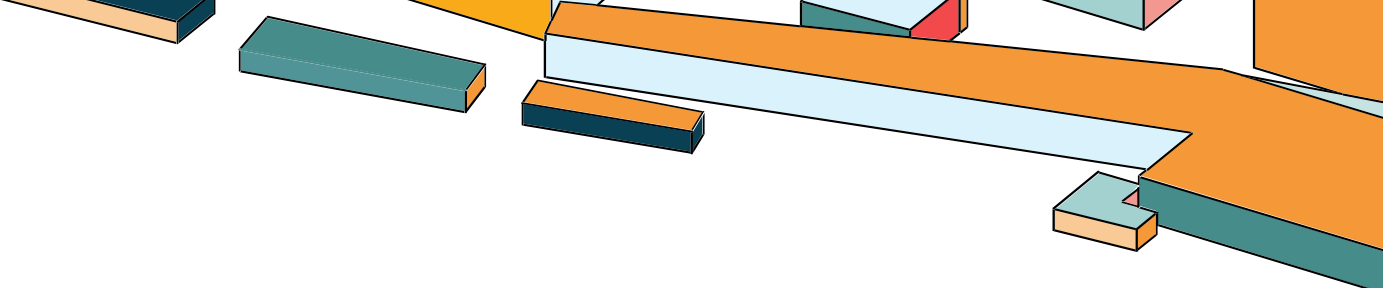
```
public interface IRenderer {  
    string Render(Message message);  
}
```

```
public class MessageRenderer : IRenderer {  
    public IReadOnlyList<IRenderer> SubRenderers { get; }
```

Способ 1: проверить
правильность заполнения
коллекции



```
        public MessageRenderer() {  
            SubRenderers = new List<IRenderer>  
            {  
                new HeaderRenderer(),  
                new BodyRenderer(),  
                new FooterRenderer()  
            };  
        }  
  
        public string Render(Message message)  
        {  
            return SubRenderers  
                .Select(x => x.Render(message))  
                .Aggregate("", (str1, str2) => str1 + str2);  
        }  
    }  
}
```



```
[Fact]
public void MessageRenderer_uses_correct_sub_renderers()
{
    var sut = new MessageRenderer();

    IReadOnlyList<IRenderer> renderers = sut.SubRenderers;

    Assert.Equal(3, renderers.Count);
    Assert.IsAssignableFrom<HeaderRenderer>(renderers[0]);
    Assert.IsAssignableFrom<BodyRenderer>(renderers[1]);
    Assert.IsAssignableFrom<FooterRenderer>(renderers[2]);
}
```



Structural inspection

```

public void MessageRenderer_is_implemented_correctly()
{
    string sourceCode = File.ReadAllText(@"<project path>\MessageRenderer.cs");

    Assert.Equal(
        @"
public class MessageRenderer : IRenderer
{
    public IReadOnlyList<IRenderer> SubRenderers { get; }

    public MessageRenderer()
    {
        SubRenderers = new List<IRenderer>
        {
            new HeaderRenderer(),
            new BodyRenderer(),
            new FooterRenderer()
        };
    }

    public string Render(Message message)
    {
        return SubRenderers
            .Select(x => x.Render(message))
            .Aggregate("", (str1, str2) => str1 + str2);
    }
}", sourceCode);
}

```



Source code inspection

```

public class Message {
    public string Header { get; set; }
    public string Body { get; set; }
    public string Footer { get; set; }
}

public interface IRenderer {
    string Render(Message message);
}

public class MessageRenderer : IRenderer {
    public IReadOnlyList<IRenderer> SubRenderers { get; }

    public MessageRenderer() {
        SubRenderers = new List<IRenderer>
        {
            new HeaderRenderer(),
            new BodyRenderer(),
            new FooterRenderer()
        };
    }

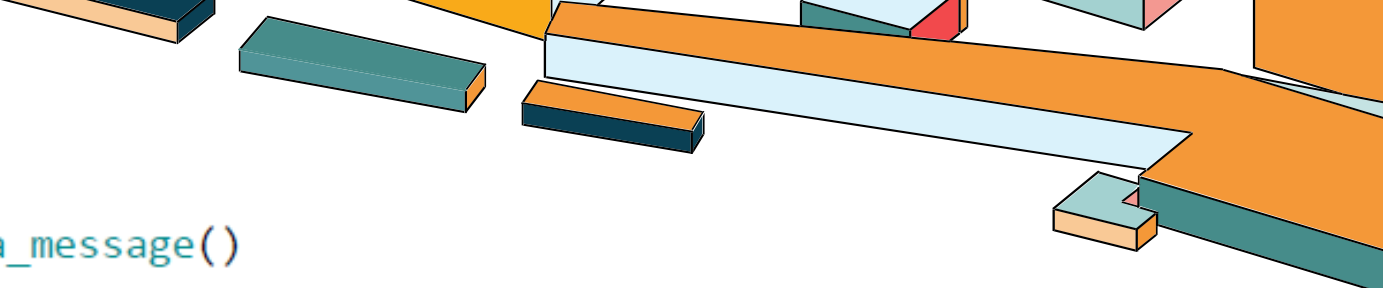
    public string Render(Message message)
    {
        return SubRenderers
            .Select(x => x.Render(message))
            .Aggregate("", (str1, str2) => str1 + str2);
    }
}

```

Способ 2: проверить конечный
результат



Тест должен привязываться к
конечному результату, а не
деталю имплементации



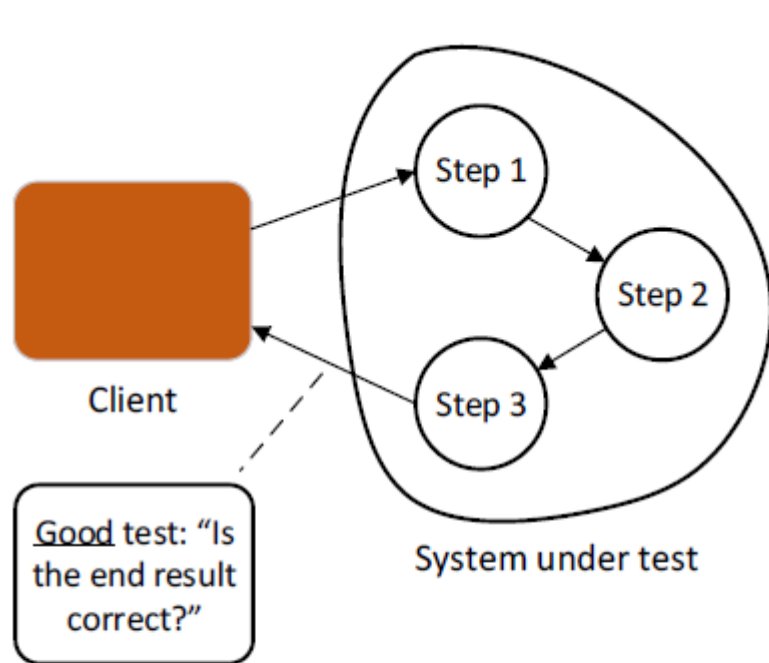
```
[Fact]
public void Rendering_a_message()
{
    var sut = new MessageRenderer();
    var message = new Message
    {
        Header = "h",
        Body = "b",
        Footer = "f"
    };

    string html = sut.Render(message);

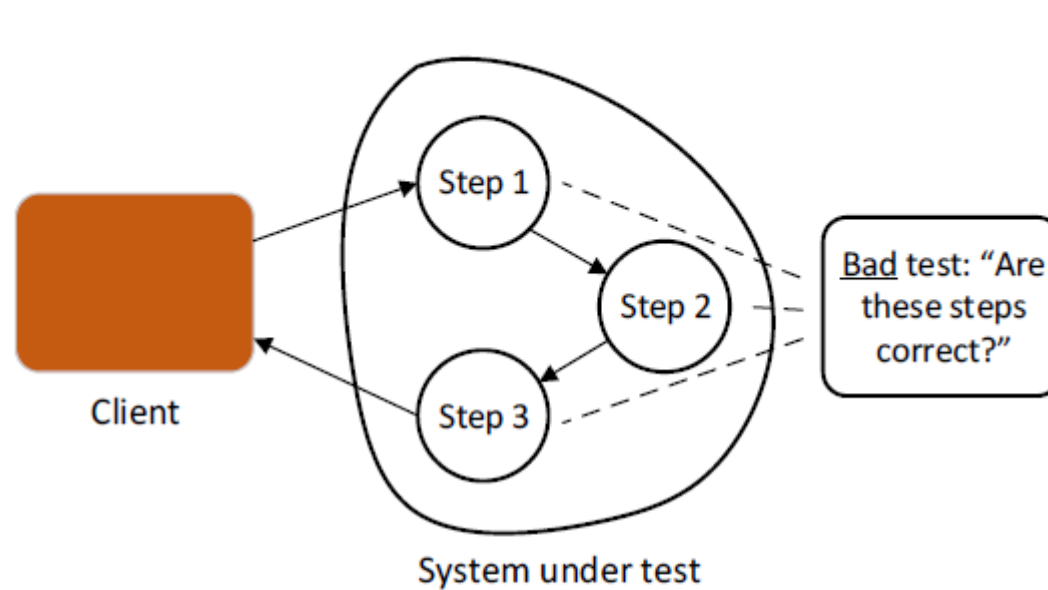
    Assert.Equal("<h1>h</h1><b>b</b><i>f</i>", html);
}
```



Тест проверяет конечный результат, а не детали имплементации



Хороший тест отвечает на вопрос: «Верен ли конечный результат?»



Плохой тест отвечает на вопрос: «Верен ли процесс?»

КАК ИЗМЕРИТЬ ЭФФЕКТИВНОСТЬ ЮНИТ ТЕСТОВ

1. Защита от багов
(protection against bugs)

2. Устойчивость к
рефакторингу
(resilience to refactoring)

3. Скорость обратной связи
(fast feedback)

4. Простота поддержки
(maintainability)

СВЯЗЬ МЕЖДУ ПЕРВЫМИ ДВУМЯ ПАРАМЕТРАМИ

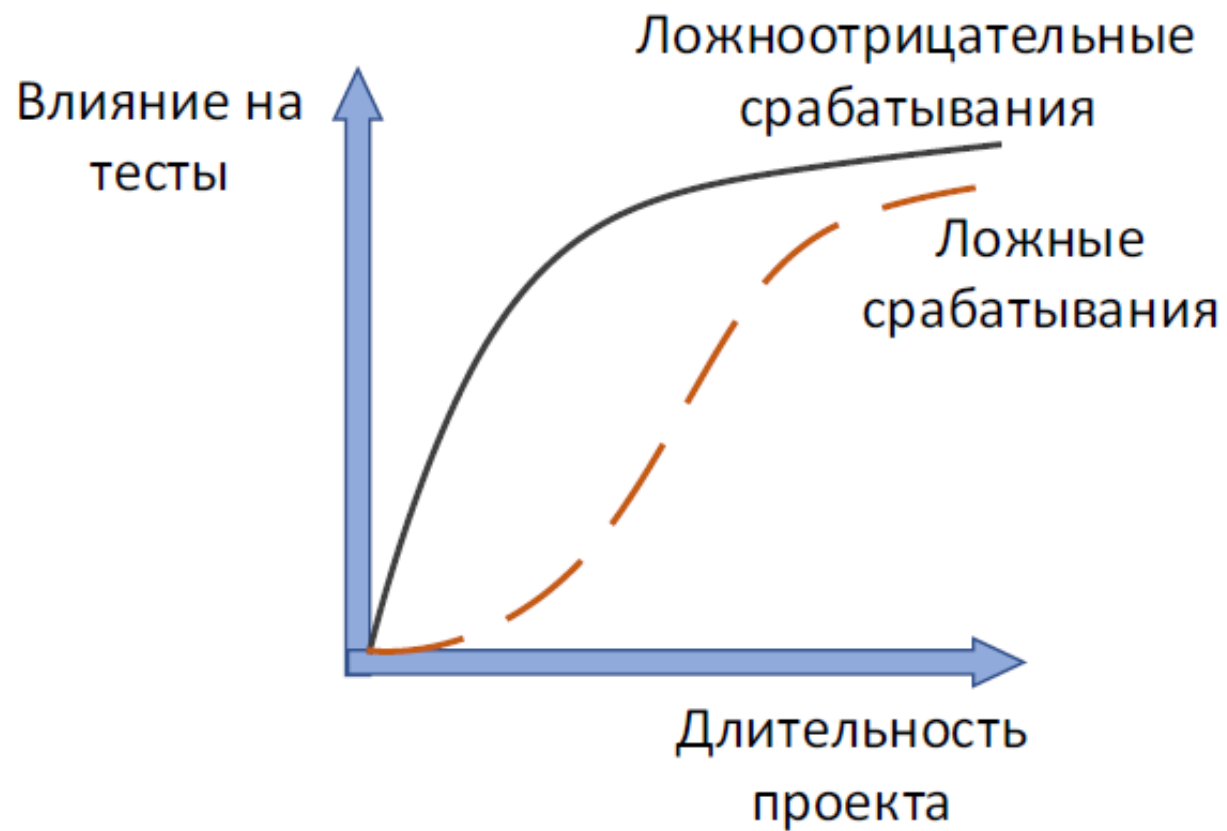
Точность теста = $\frac{\text{Сигнал (кол-во найденных багов)}}{\text{Шум (кол-во ложных срабатываний)}}$

Защита от багов

Устойчивость к рефакторингу

The diagram illustrates the relationship between test accuracy and two parameters. The formula for test accuracy is shown as a fraction: the numerator is 'Сигнал (кол-во найденных багов)' (Signal (number of found bugs)) and the denominator is 'Шум (кол-во ложных срабатываний)' (Noise (number of false alarms)). A red line connects the numerator to the text 'Защита от багов' (Protection from bugs), and another red line connects the denominator to the text 'Устойчивость к рефакторингу' (Stability to refactoring). The entire formula is enclosed in a red rectangular border.

СВЯЗЬ МЕЖДУ ПЕРВЫМИ ДВУМЯ ПАРАМЕТРАМИ



ФУНКЦИОНАЛЬНЫЕ ТЕСТЫ



Наилучшая защита от багов

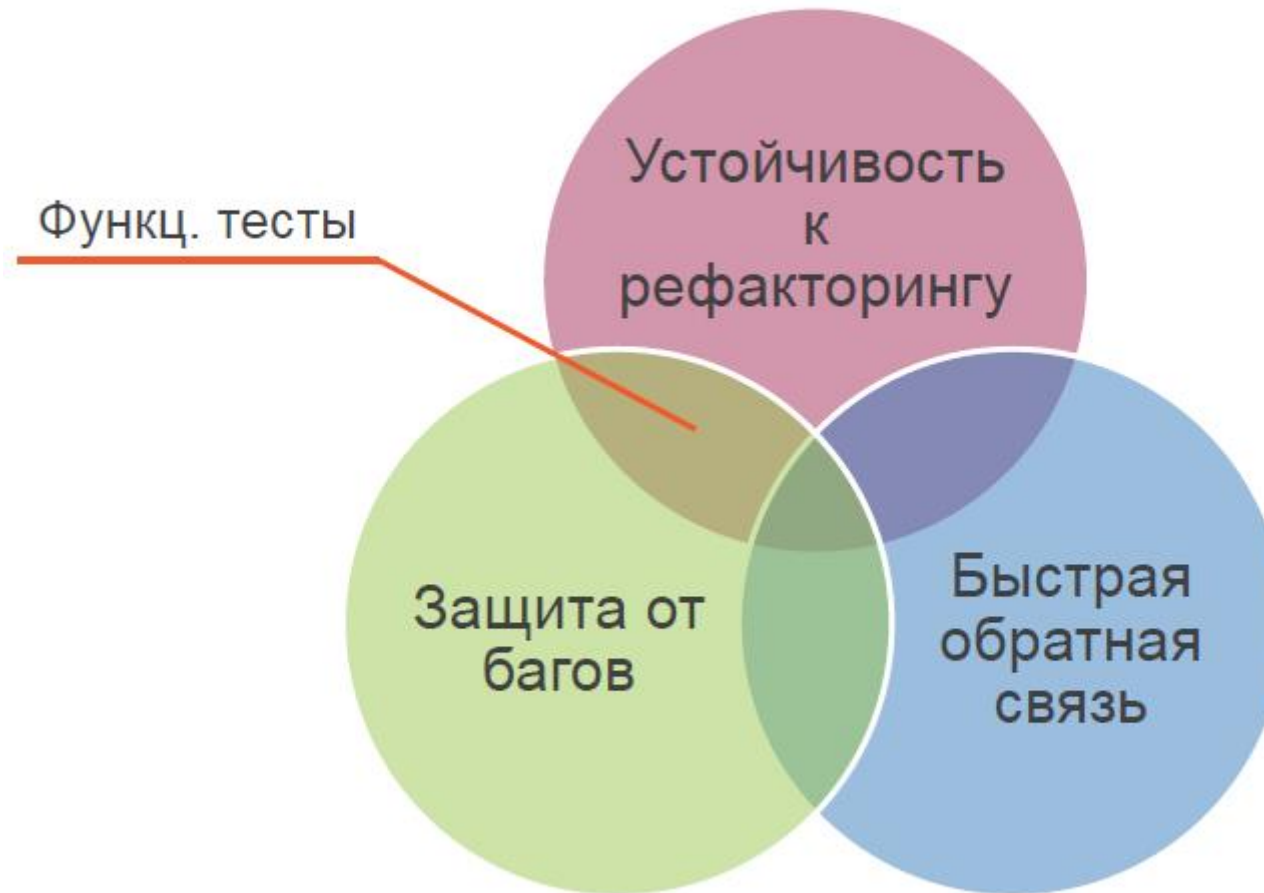


Наилучшая устойчивость к рефакторингу



Медленная обратная связь

ФУНКЦИОНАЛЬНЫЕ ТЕСТЫ



ТРИВИАЛЬНЫЕ ТЕСТЫ

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

```
[Fact]
public void Test()
{
    var user = new User();

    user.Name = "John Smith";

    Assert.Equal("John Smith", user.Name);
}
```



Быстрая обратная связь

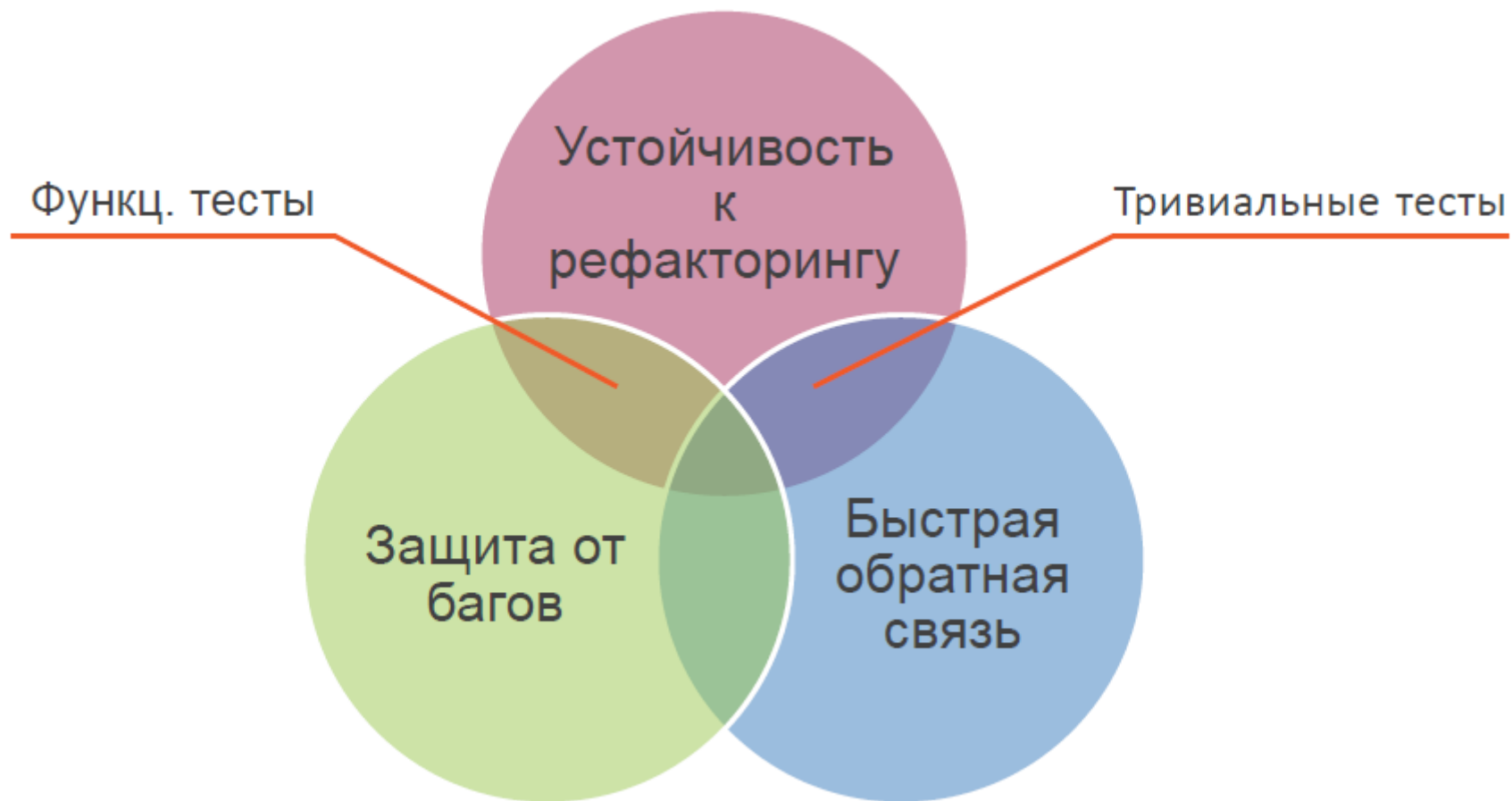


Хорошая устойчивость к рефакторингу



Плохая защита от багов

ТРИВИАЛЬНЫЕ ТЕСТЫ



ХРУПКИЕ ТЕСТЫ

```
public class UserRepository
{
    public User GetById(int id)
    {
        /* ... */
    }

    public string LastExecutedSql
    { get; private set; }
}
```

```
[Fact]
public void GetById_executes_correct_SQL_code()
{
    var repository = new UserRepository();

    User = repository.GetById(5);

    Assert.Equal(
        "SELECT * FROM dbo.[User] WHERE UserID = 5",
        repository.LastExecutedSqlStatement);
}
```

```
SELECT * FROM dbo.[User] WHERE UserID = 5
SELECT * FROM dbo.User WHERE UserID = 5
SELECT UserID, Name, Email FROM dbo.[User] WHERE UserID = 5
SELECT * FROM dbo.[User] WHERE UserID = @UserID
```



Быстрая обратная связь



Плохая устойчивость к рефакторингу



Хорошая защита от багов

ХРУПКИЕ ТЕСТЫ



ИДЕАЛЬНЫЙ ТЕСТ



НАПИСАНИЕ ЭФФЕКТИВНЫХ ТЕСТОВ



Эффективное юнит тестирование
требует рефакторинга кода

НАПИСАНИЕ ЭФФЕКТИВНЫХ ТЕСТОВ

Сложность, важность

Количество зависимостей-
собеседников (collaborators)



Хорошая защита от багов



Большая стоимость
поддержки

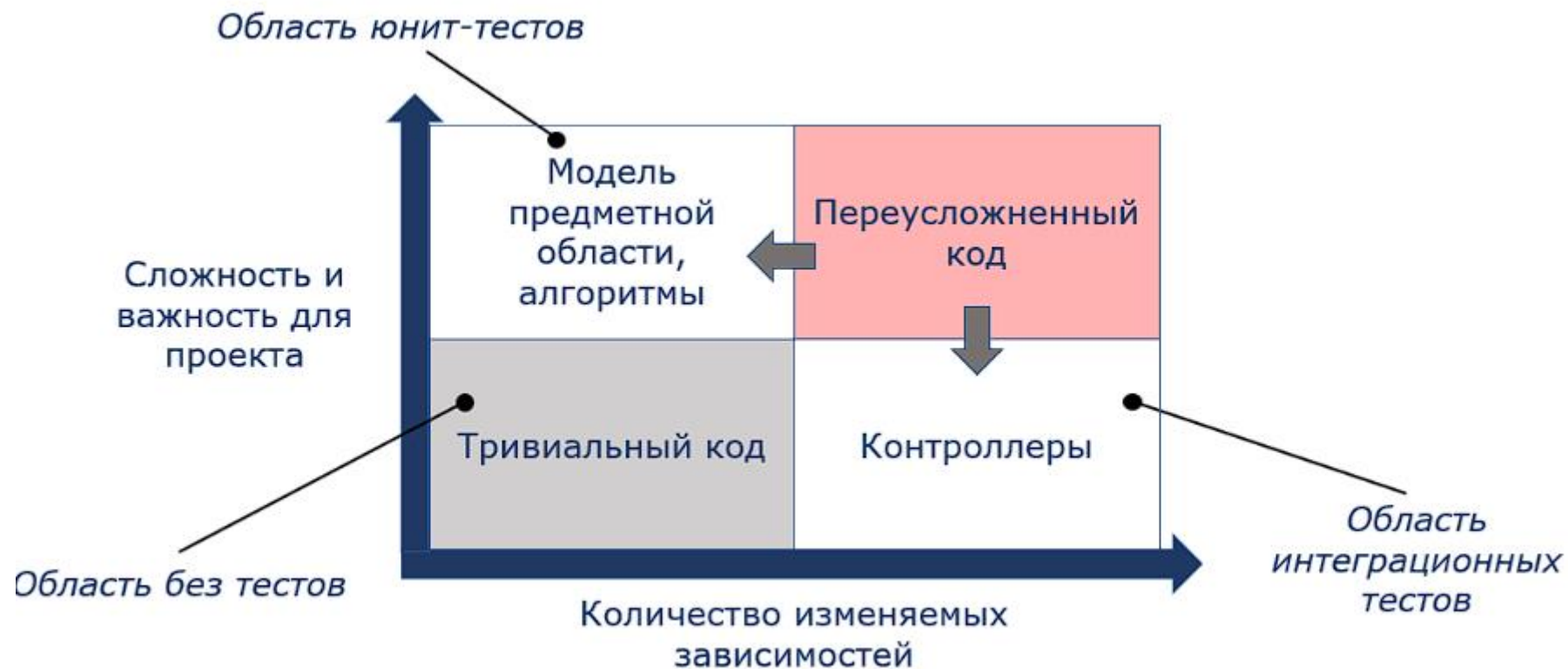
НАПИСАНИЕ ЭФФЕКТИВНЫХ ТЕСТОВ

Сложность и
важность для
проекта

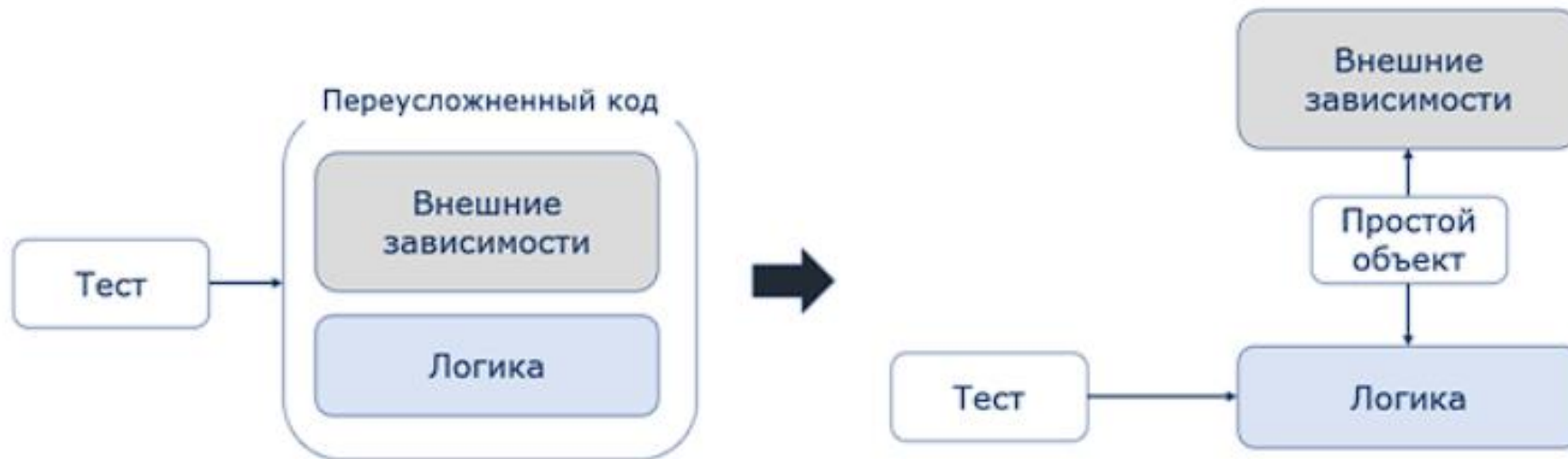


- A. Область Юнит тестов
- B. Область интеграционных тестов
- C. Область где тесты не нужны
- D. Область где не реально написать тест

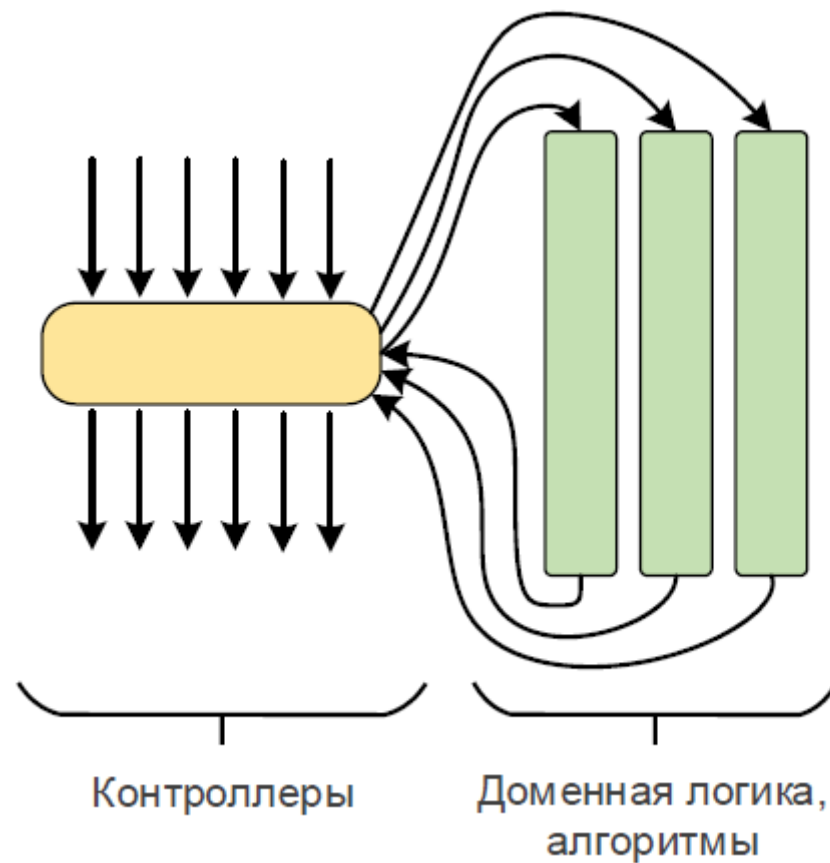
НАПИСАНИЕ ЭФФЕКТИВНЫХ ТЕСТОВ



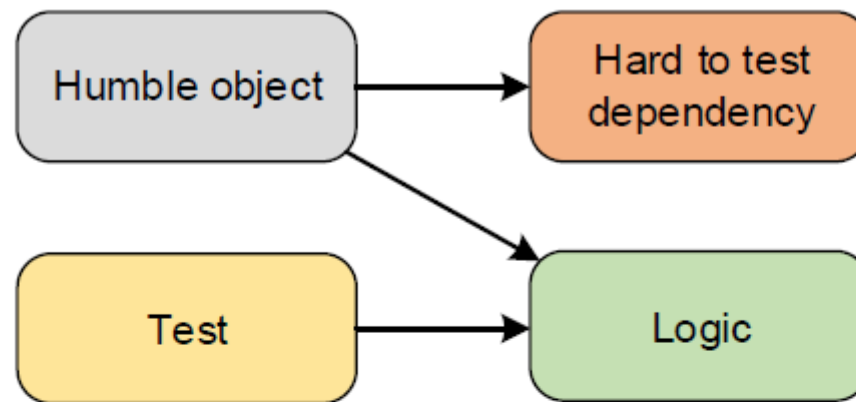
ПРИМЕНЕНИЕ HUMBLE OBJECT ПАТТЕРНА



ПРИМЕНЕНИЕ HUMBLE OBJECT ПАТТЕРНА



ПРИМЕНЕНИЕ HUMBLE OBJECT ПАТТЕРНА



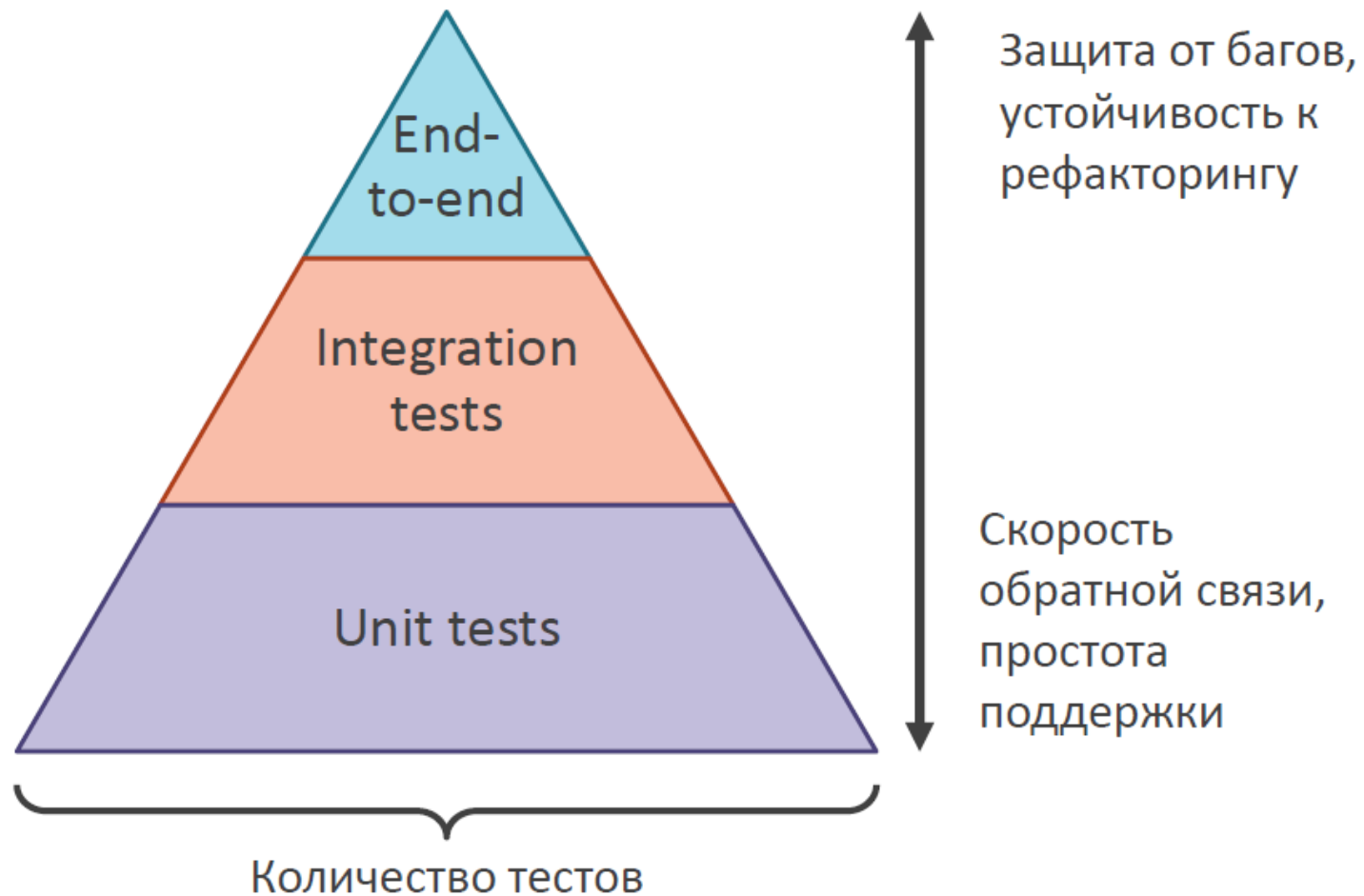
MVC (Model-View-Controller)

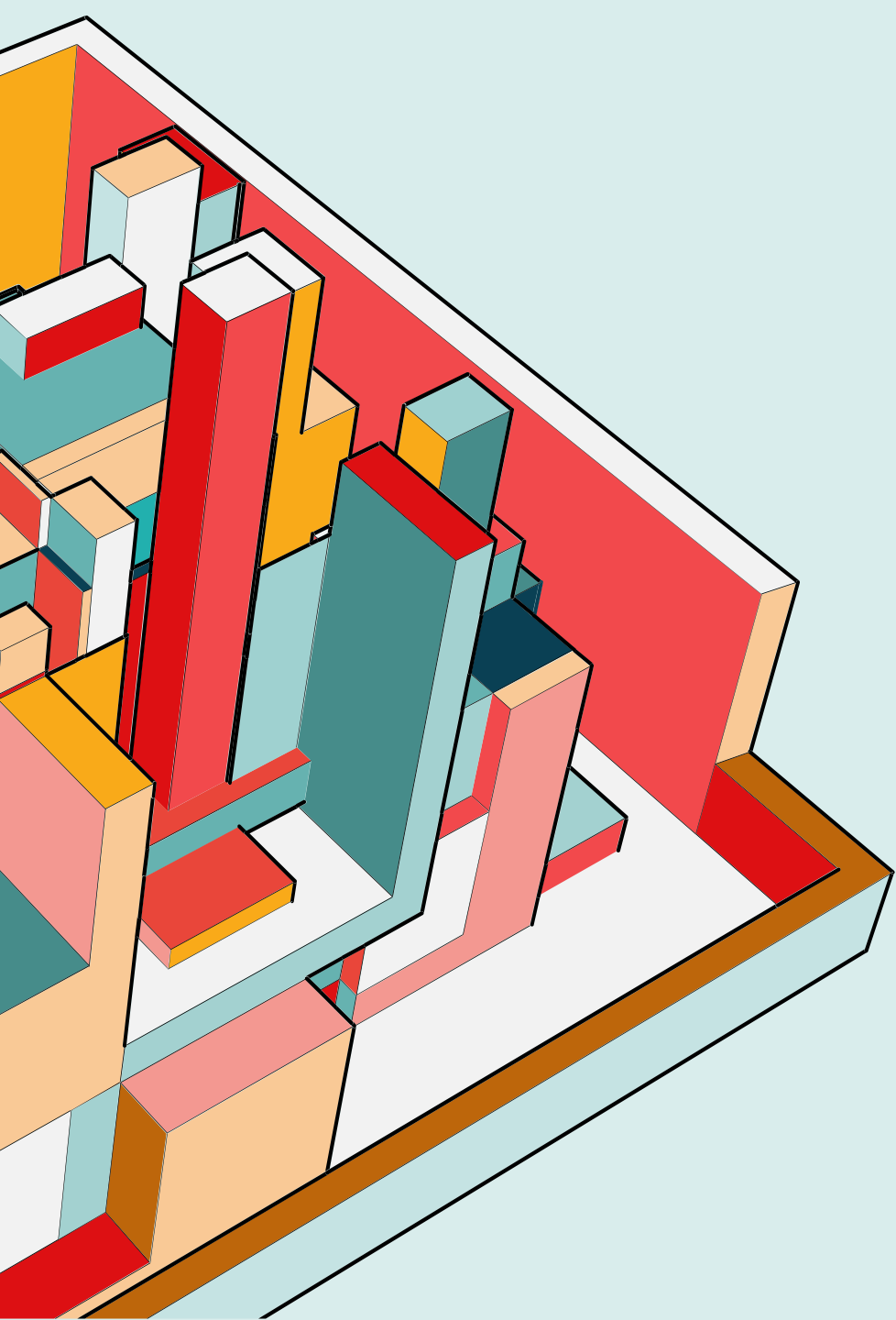
Model = Logic

View = Hard to test dependency

Controller = Humble object

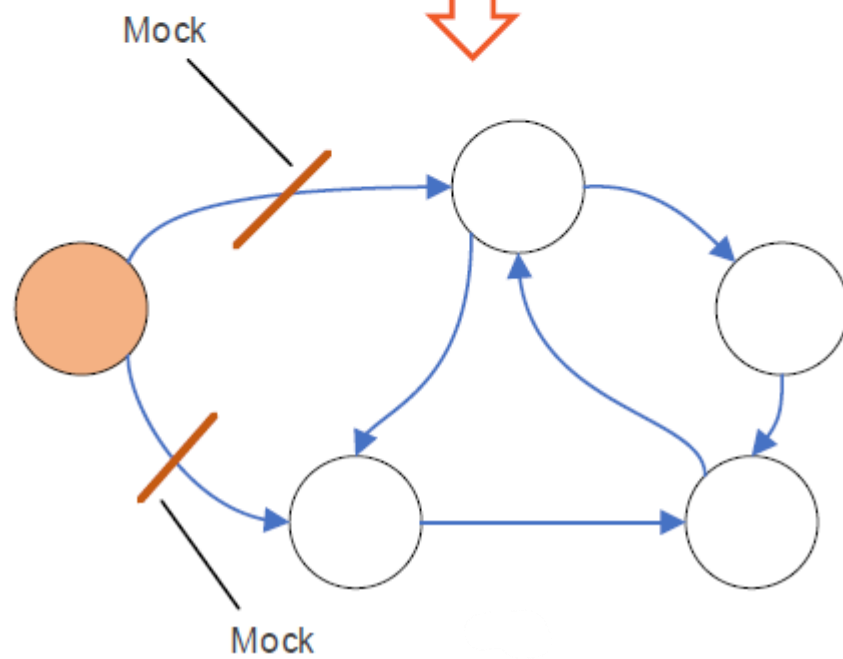
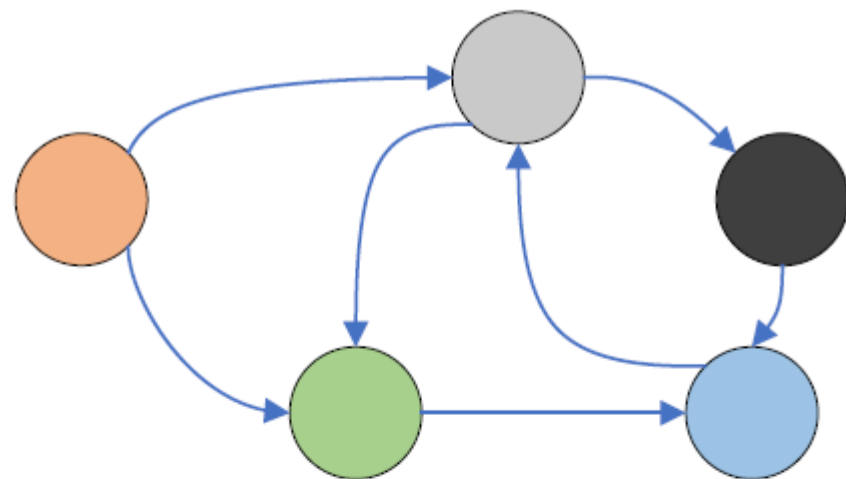
ПРИМЕНЕНИЕ HUMBLE OBJECT ПАТТЕРНА



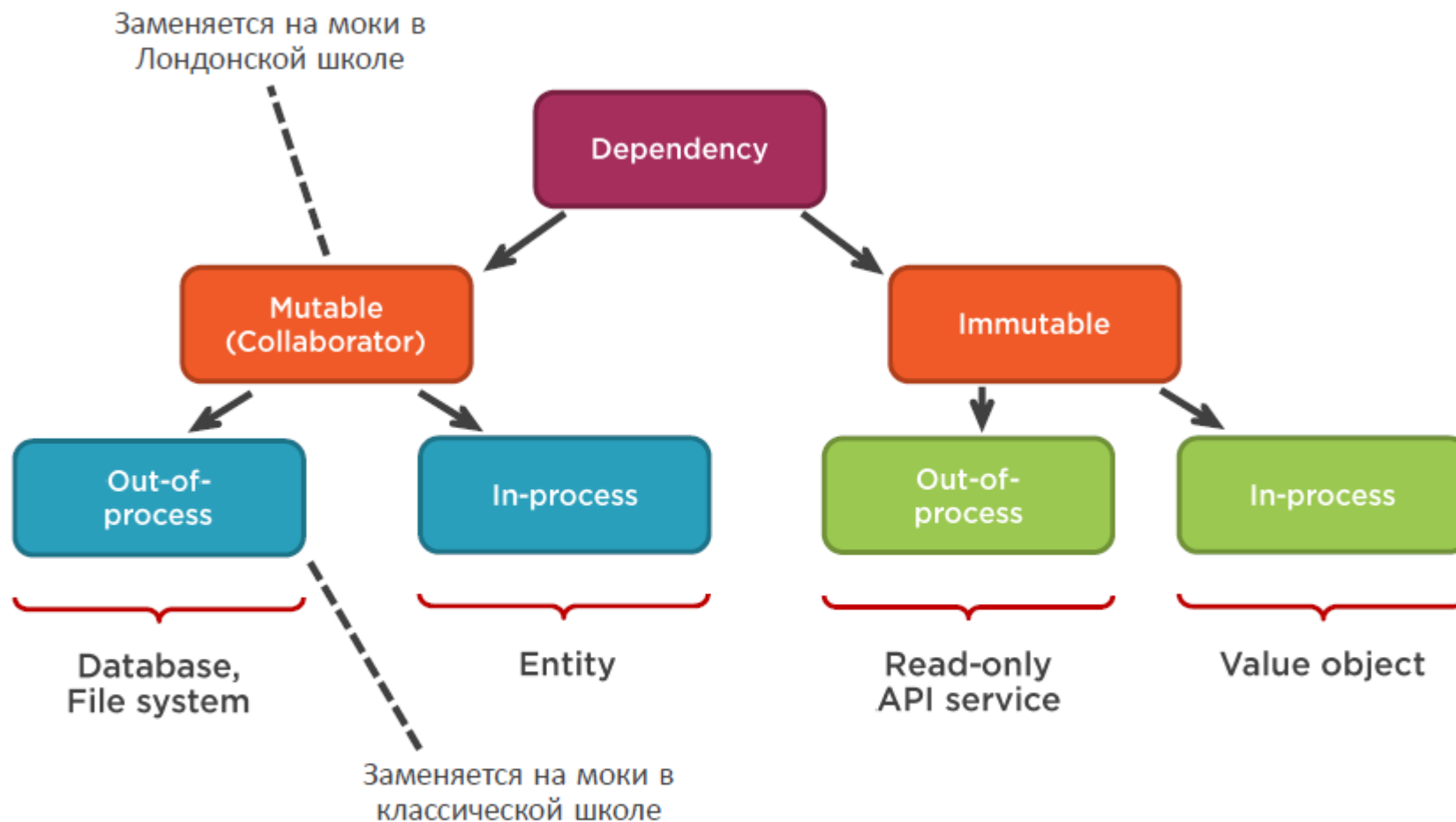


**КОГДА НУЖНО
ИСПОЛЬЗОВАТЬ
МОКИ**

МОКИ

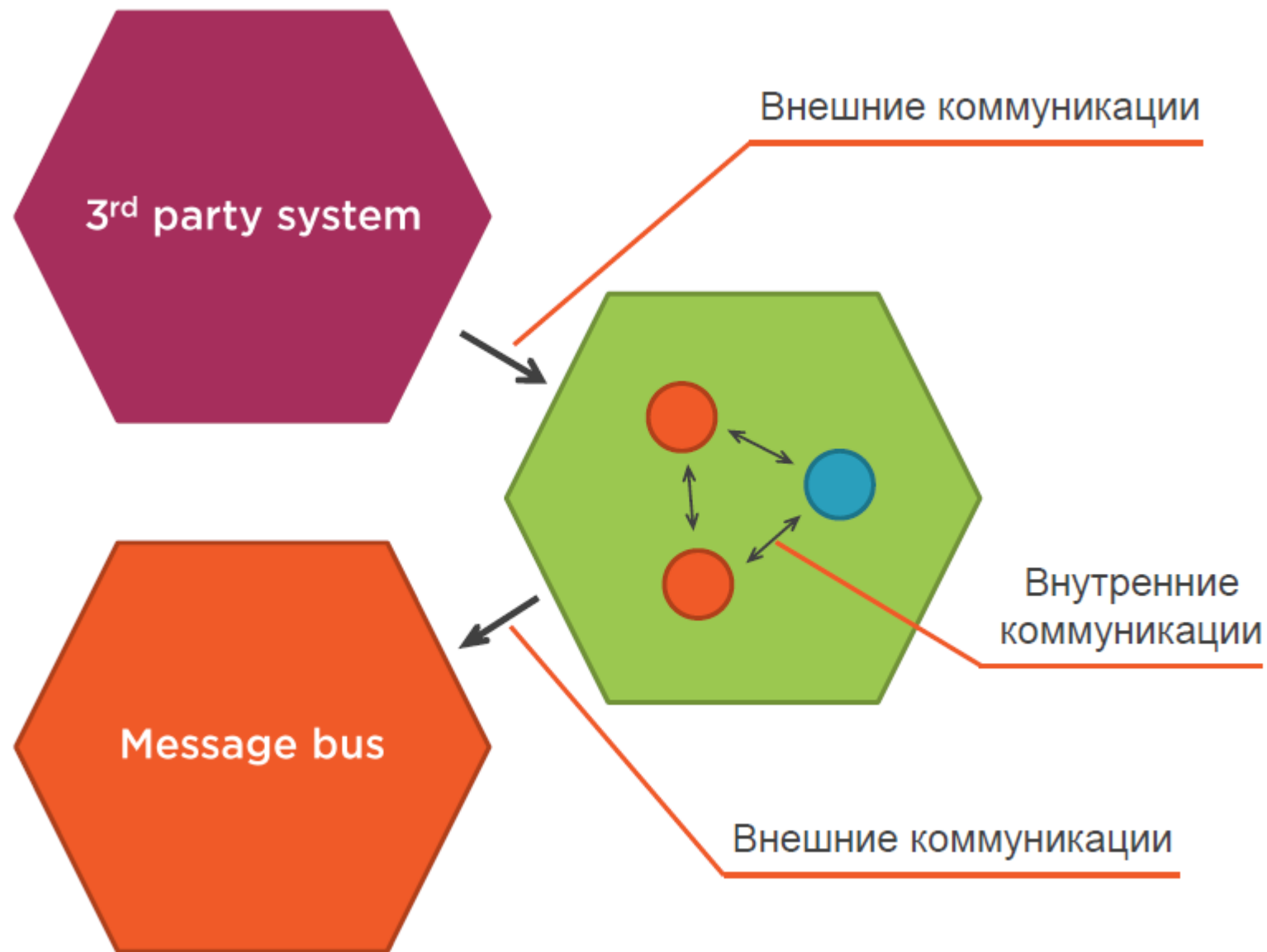


МОКИ

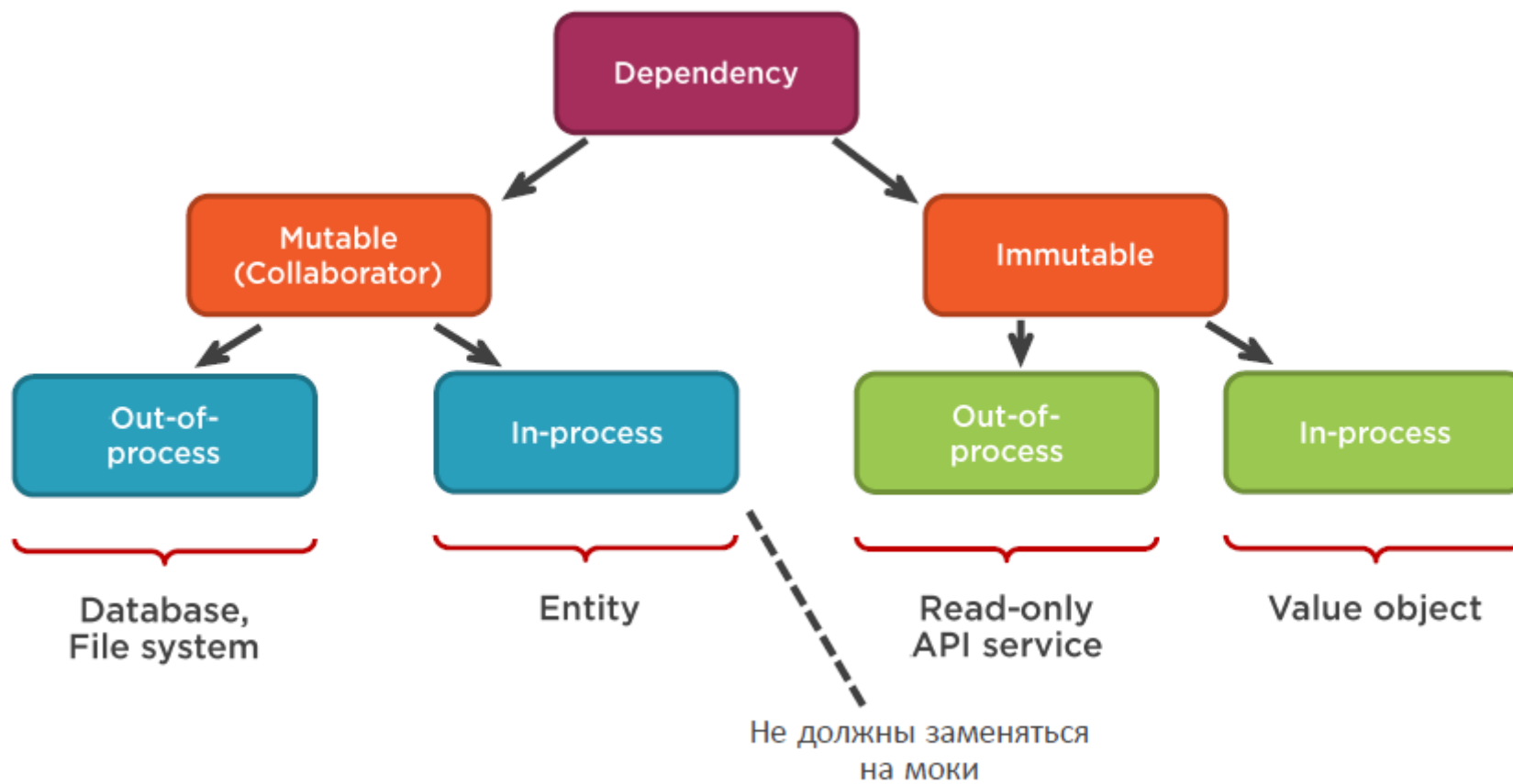


Обе школы неверны

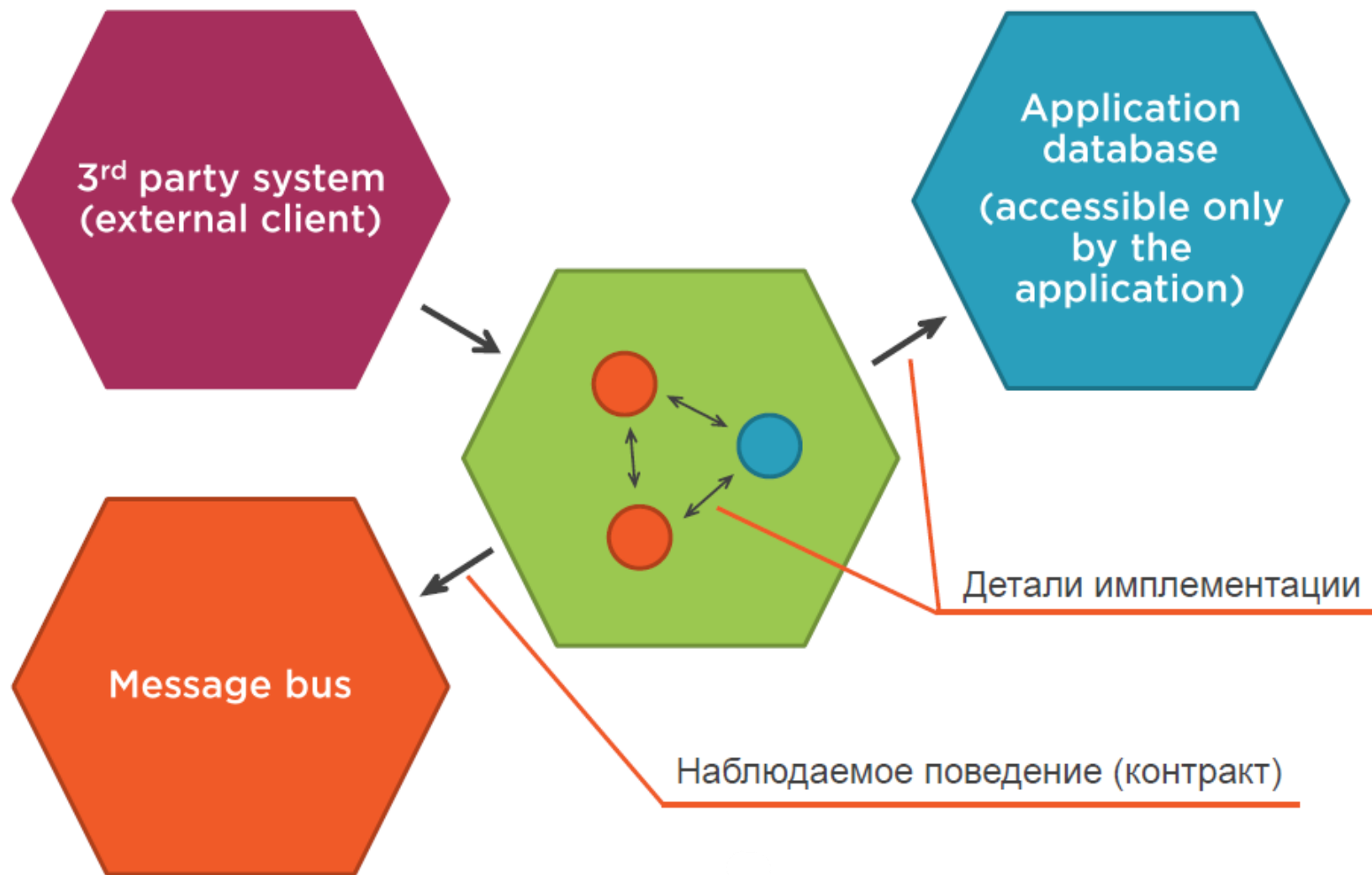
МОКИ



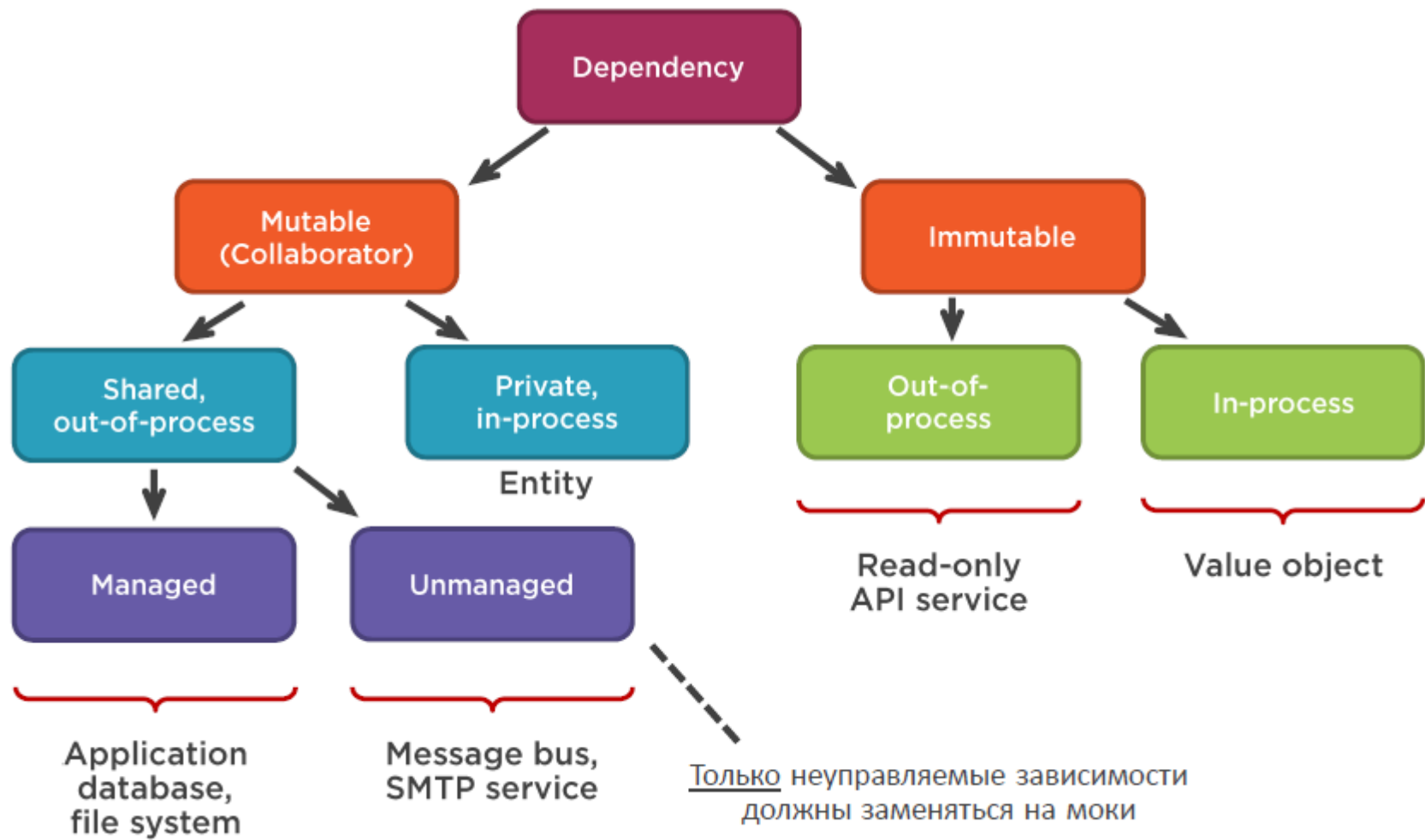
МОКИ



МОКИ



МОКИ



СПАСИБО!

Виденин Сергей

@videninserg

