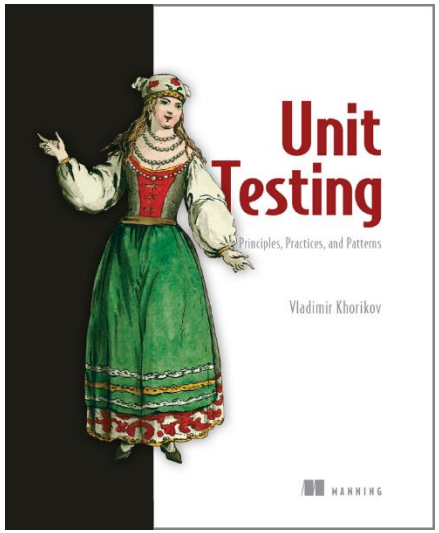
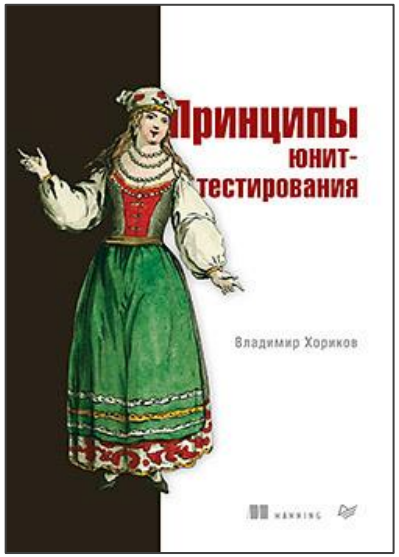
# Принципы юнит-тестирования

Владимир Хориков

@vkhorikov www.enterprisecraftsmanship.com

# About



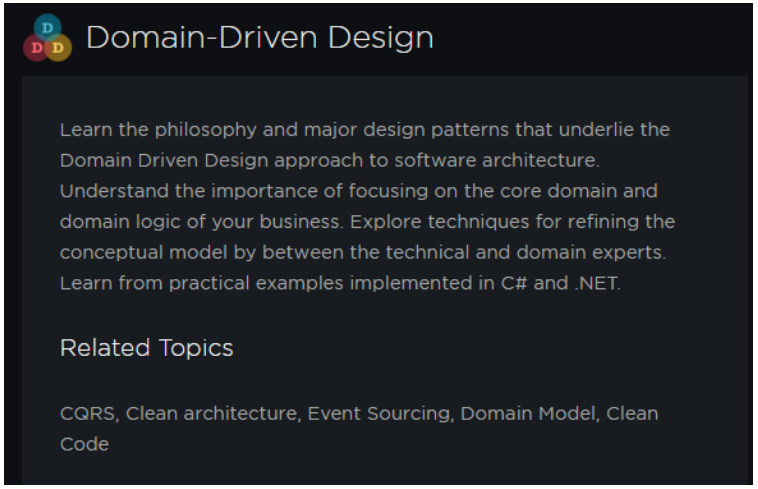




**Unit Testing Principles, Practices, and Patterns**

http://bit.ly/testing-book

**Принципы юнит-тестирования**

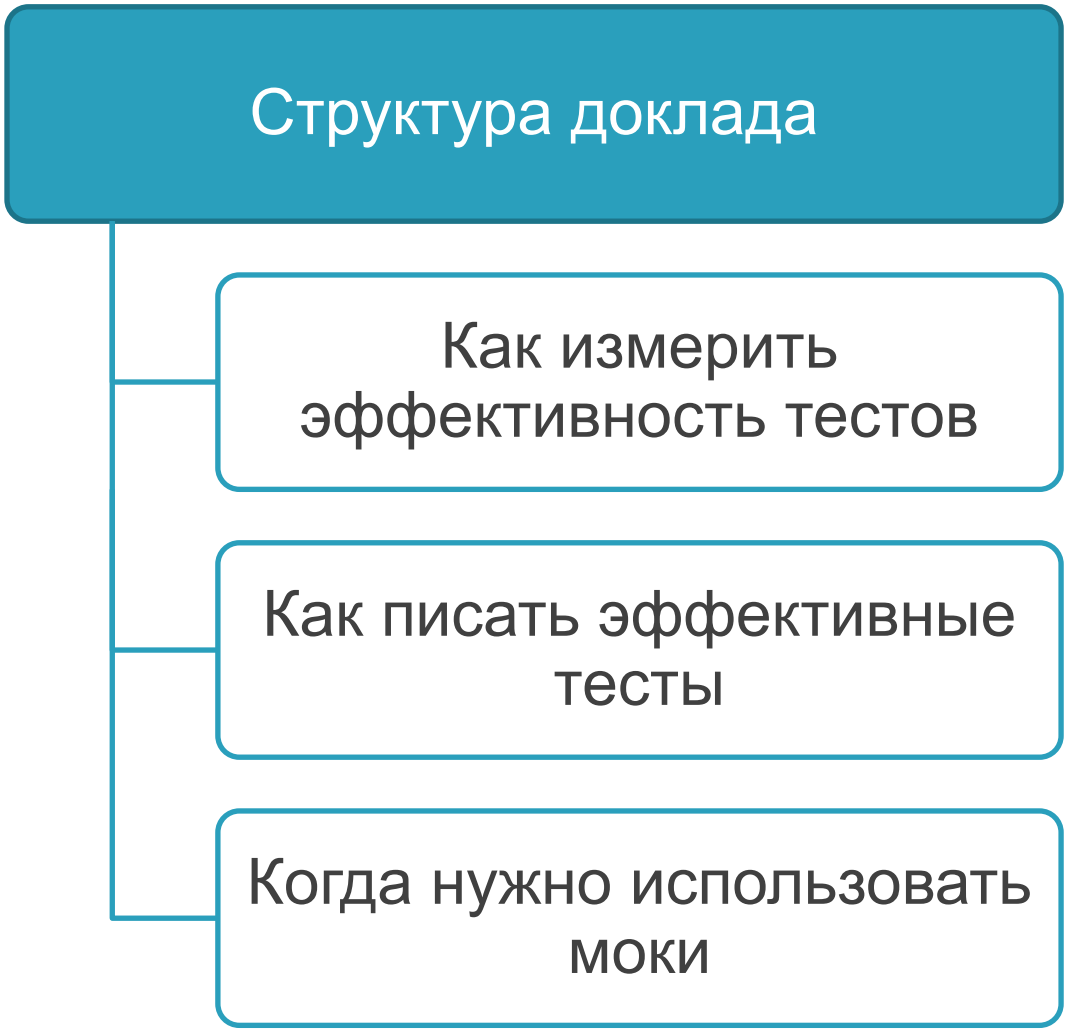
http://bit.ly/testing-book-ru




**Pluralsight Author**

https://bit.ly/ps-all

# План доклада



# 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

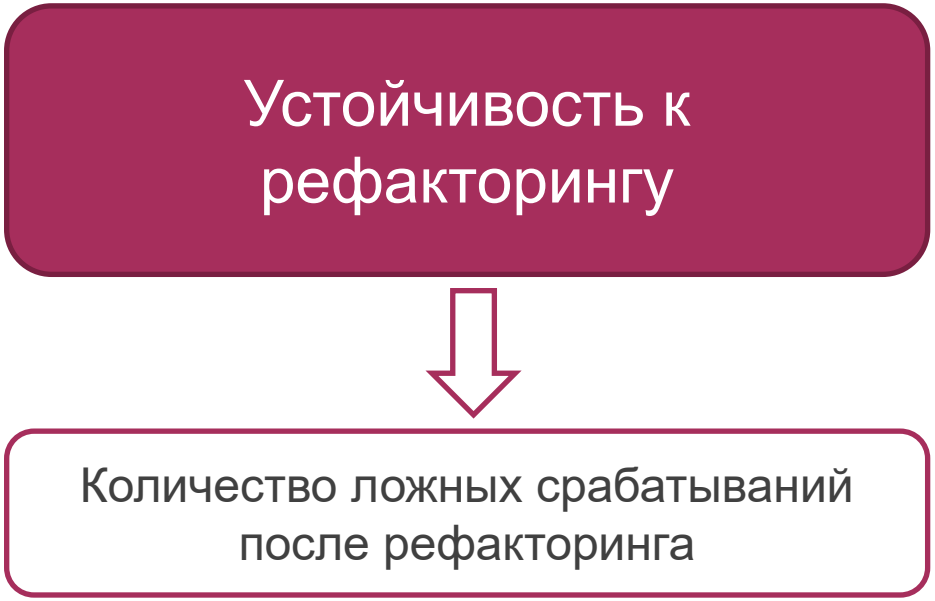
Чем быстрее тест, тем меньше
времени тратится на устранение багов

# Простота поддержки


- Размер и простота теста
- Количество внешних зависимостей

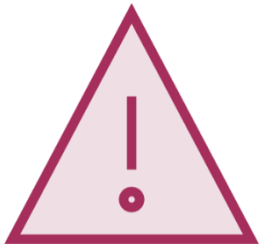
# Защита от багов

# Устойчивость к рефакторингу




Ложное срабатывание = False positive

# Устойчивость к рефакторингу



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

# Ложные срабатывания: пример

```
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);
    }
}
```

# Ложные срабатывания: 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>";
    }
}
```

# Ложные срабатывания: пример

```
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 1: проверить правильность заполнения коллекции

# Ложные срабатывания: пример

```
[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

# Ложные срабатывания: пример

```
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

# Ложные срабатывания: пример

Тест должен привязываться к конечному результату, а не деталям имплементации

# Ложные срабатывания: пример

```
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);
    }
}
```

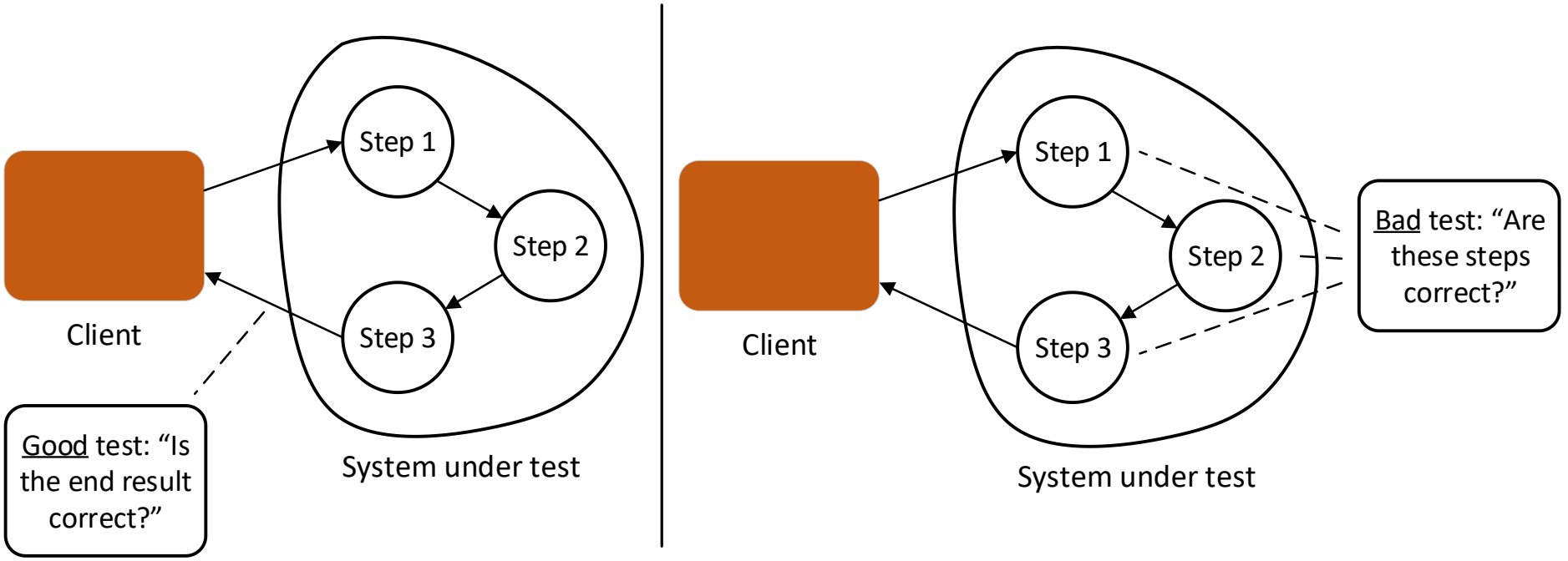
Способ 2: проверить конечный результат

# Ложные срабатывания: пример

```
[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

# Ложные срабатывания: пример




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

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

# Как измерить эффективность юнит тестов?

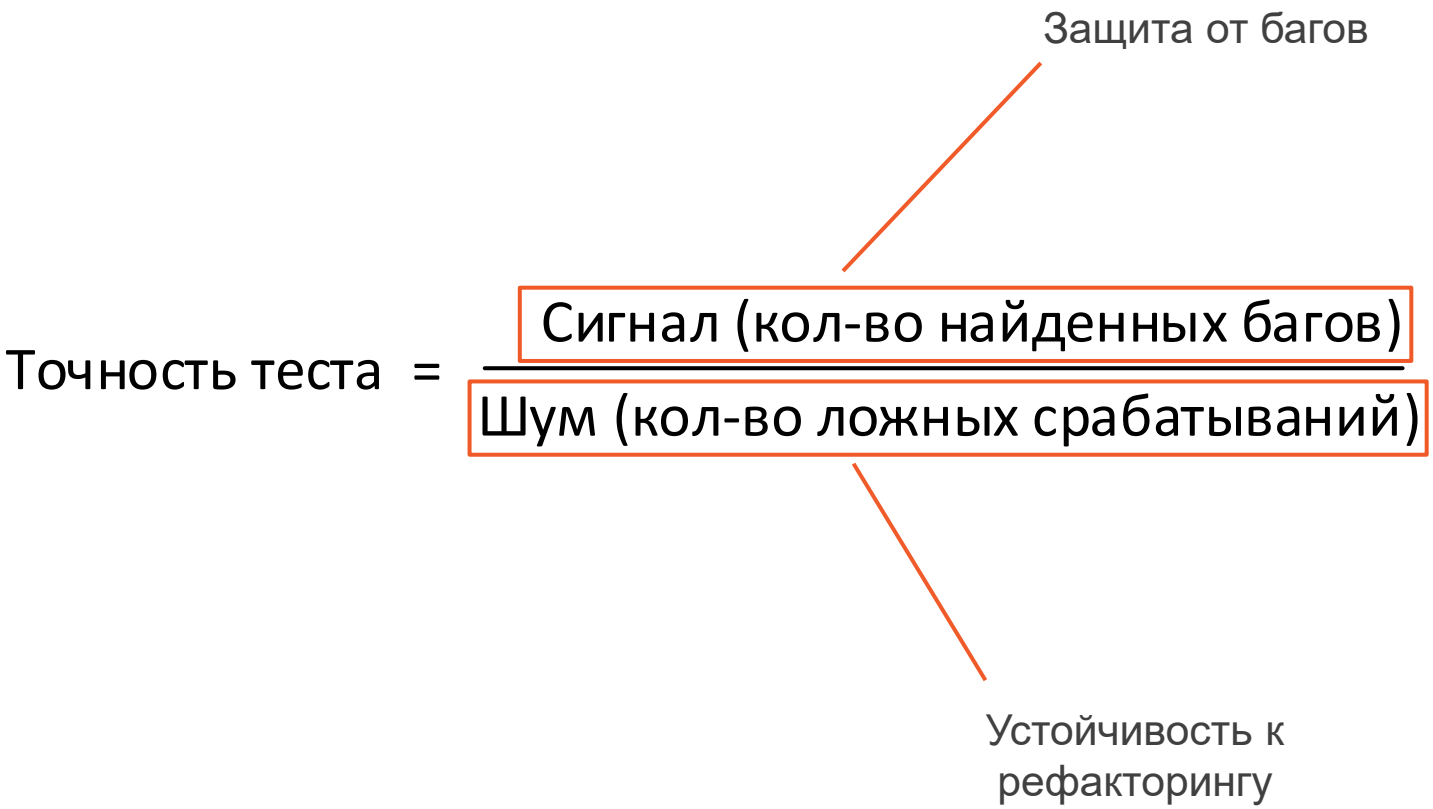
1. Защита от багов
(protection against bugs)

2. Устойчивость к
рефакторингу
(resilience to refactoring)

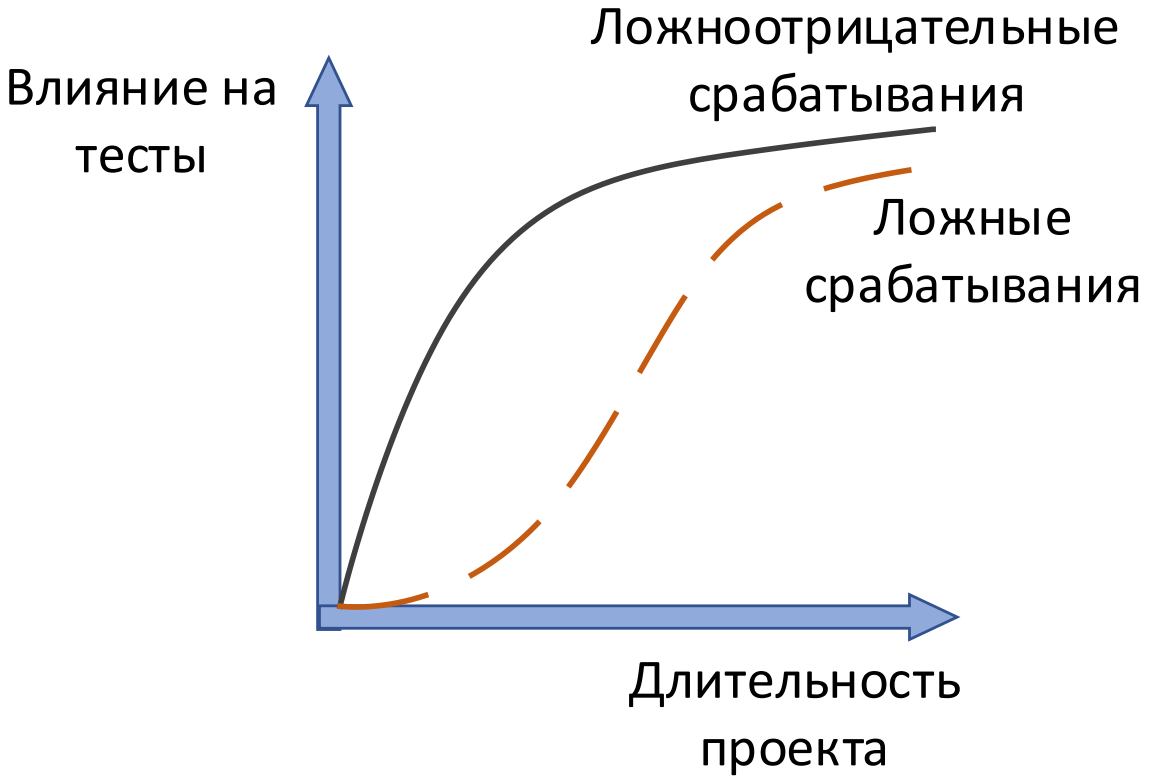
3. Скорость обратной связи
(fast feedback)

4. Простота поддержки
(maintainability)

# Связь между первыми двумя параметрами



# Связь между первыми двумя параметрами



# Как измерить эффективность юнит тестов?

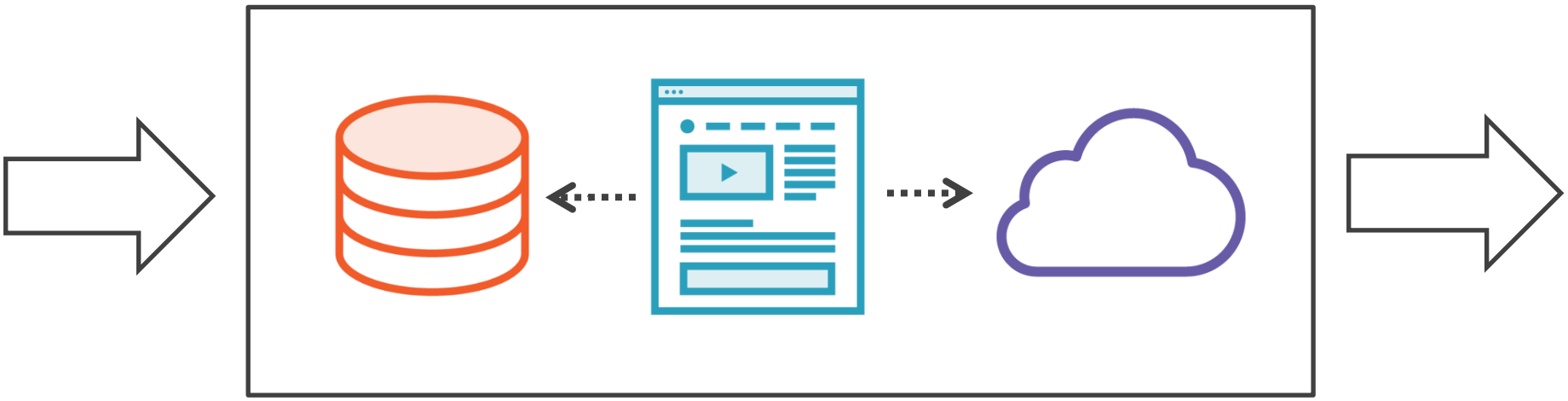
1. Защита от багов
(protection against bugs)

2. Устойчивость к
рефакторингу
(resilience to refactoring)

3. Скорость обратной связи
(fast feedback)

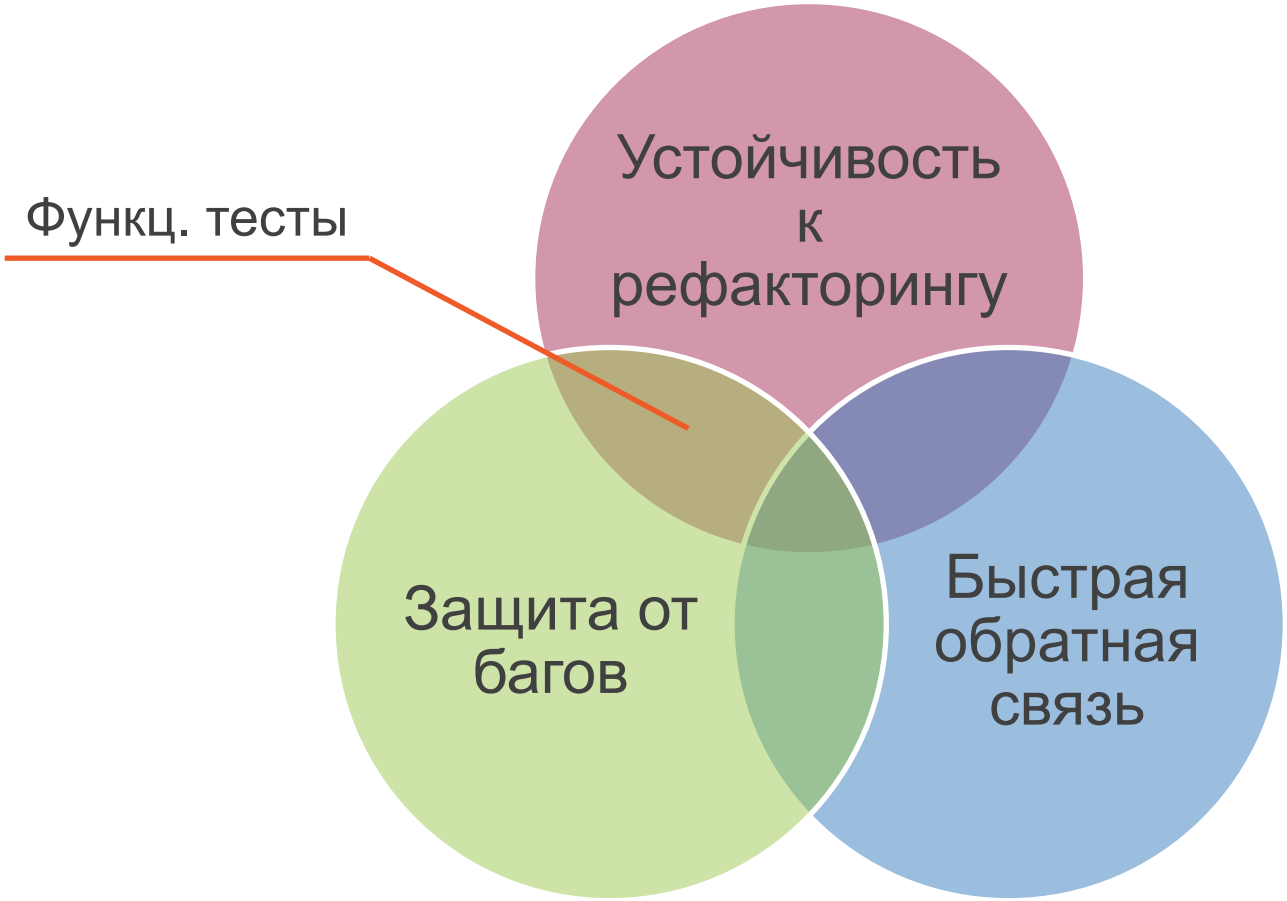
4. Простота поддержки
(maintainability)

# Функциональные (end-to-end) тесты



- Наилучшая защита от багов
- Наилучшая устойчивость к рефакторингу
- Медленная обратная связь

# Функциональные (end-to-end) тесты



# Тривиальные тесты

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

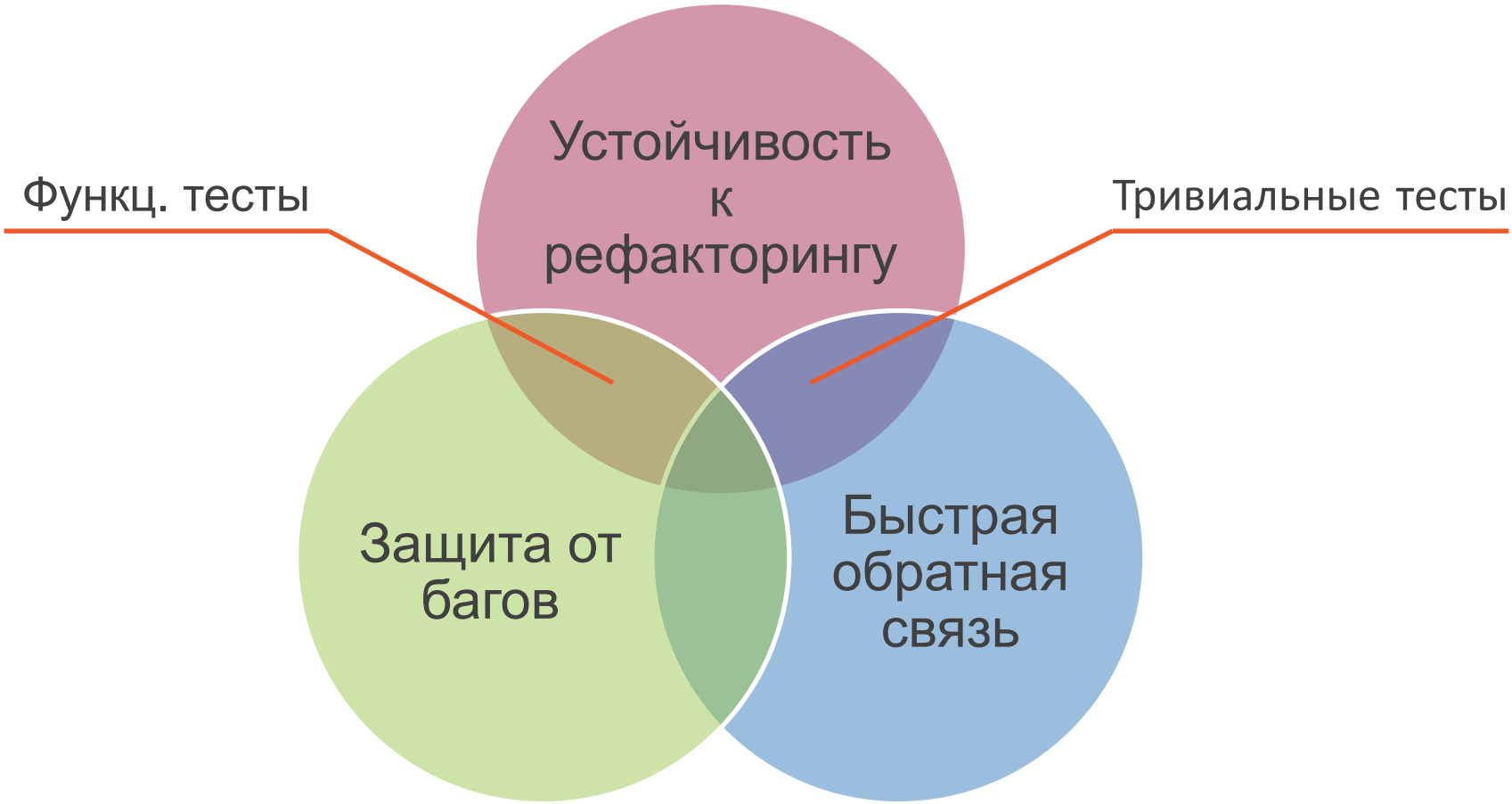
```
[Fact]
public void Test()
{
    var user = new User();

    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);
}
```

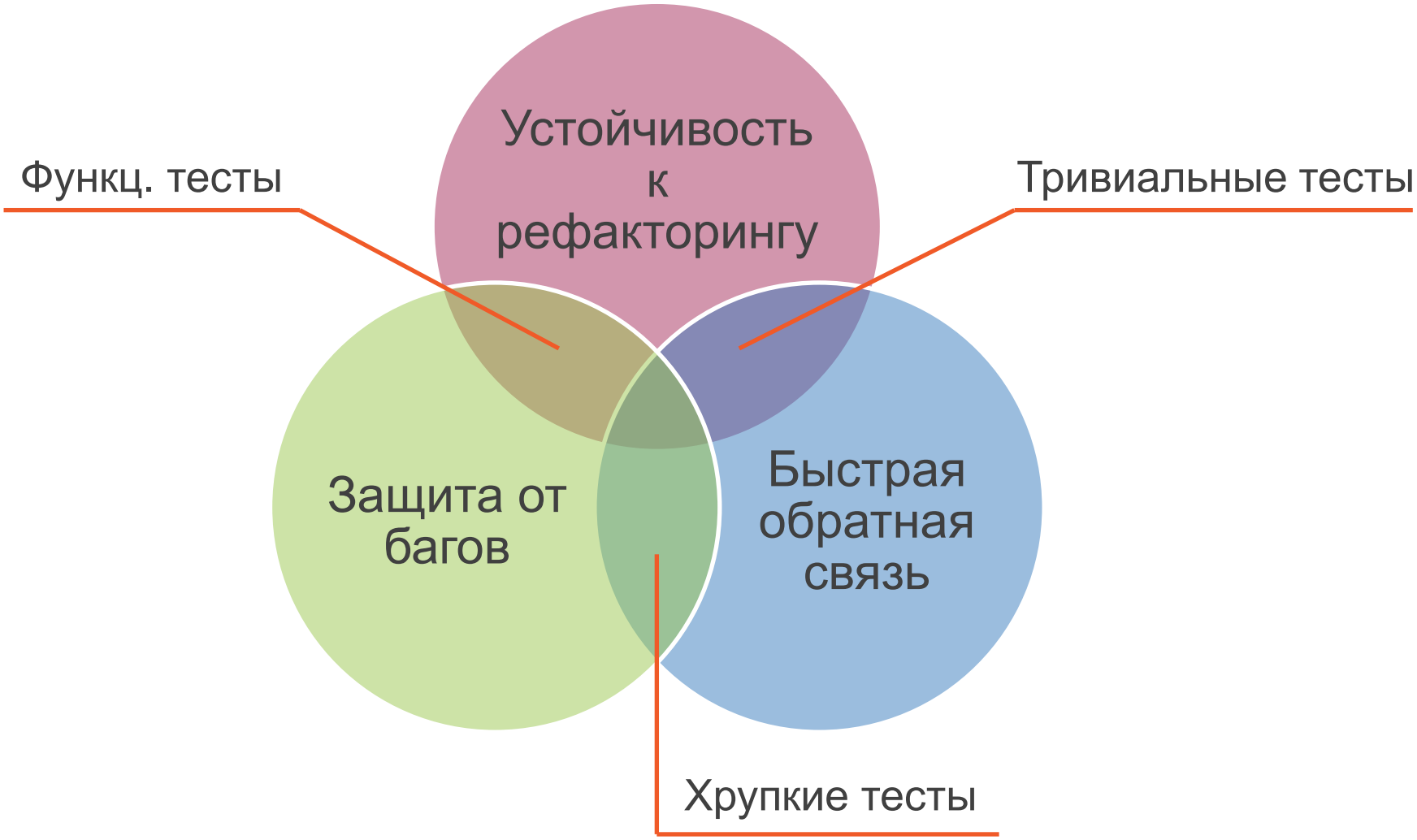
```
SELECT * FROM dbo.[User] WHERE UserID = 5
SELECT * FROM dbo.User WHERE UserID = 5
SELECT UserID, Name, Email FROM dbo.[User] WHERE UserID = 5
SELECT * FROM dbo.[User] WHERE UserID = @UserID
```

Быстрая обратная связь

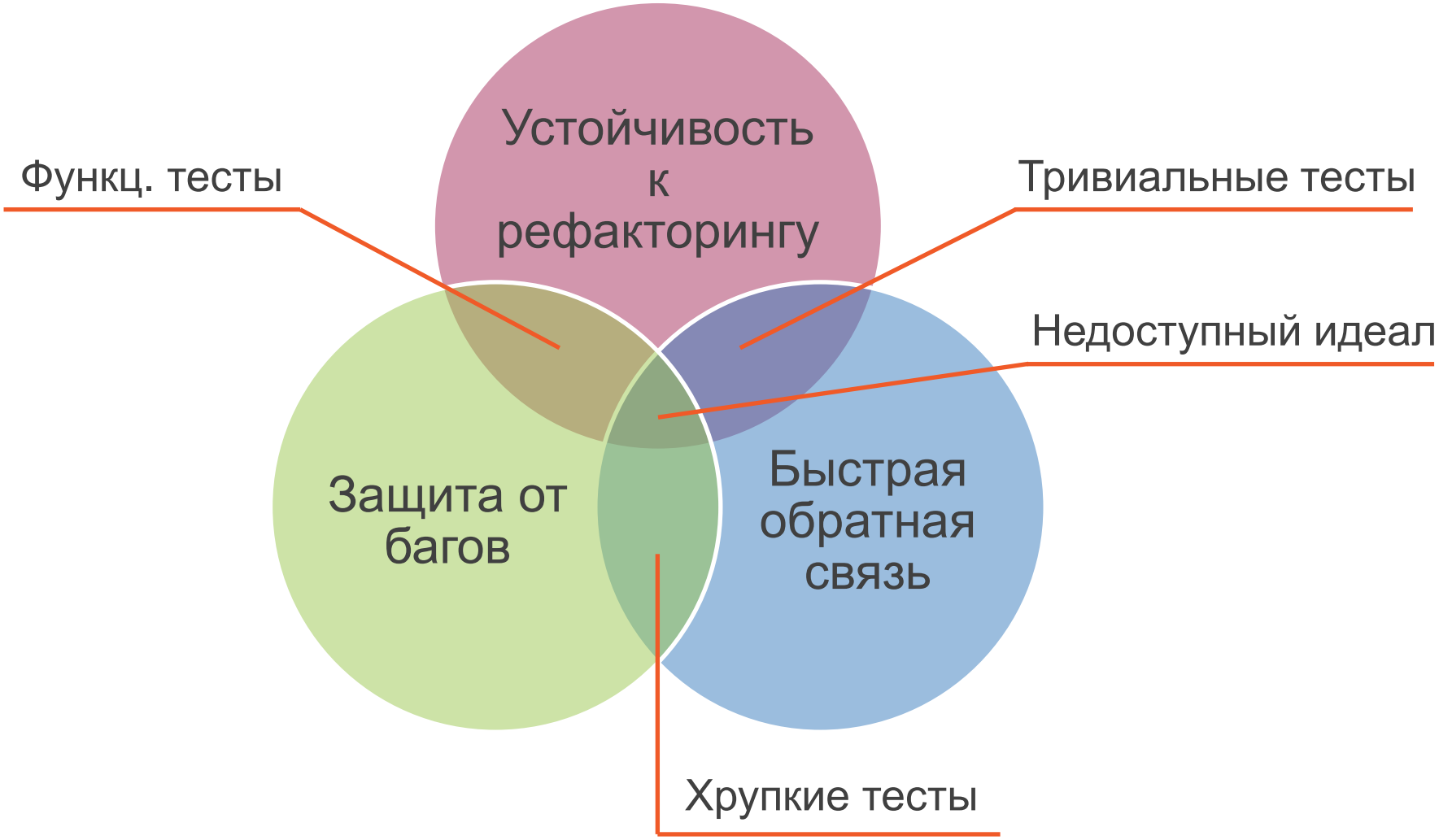
Плохая устойчивость к рефакторингу

Хорошая защита от багов

# Хрупкие тесты

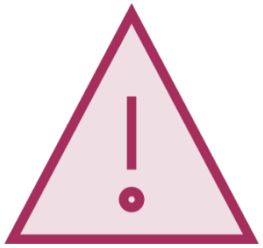


# Идеальный тест



# План доклада

# Написание эффективных тестов



Эффективное юнит тестирование
требует рефакторинга кода

# Написание эффективных тестов

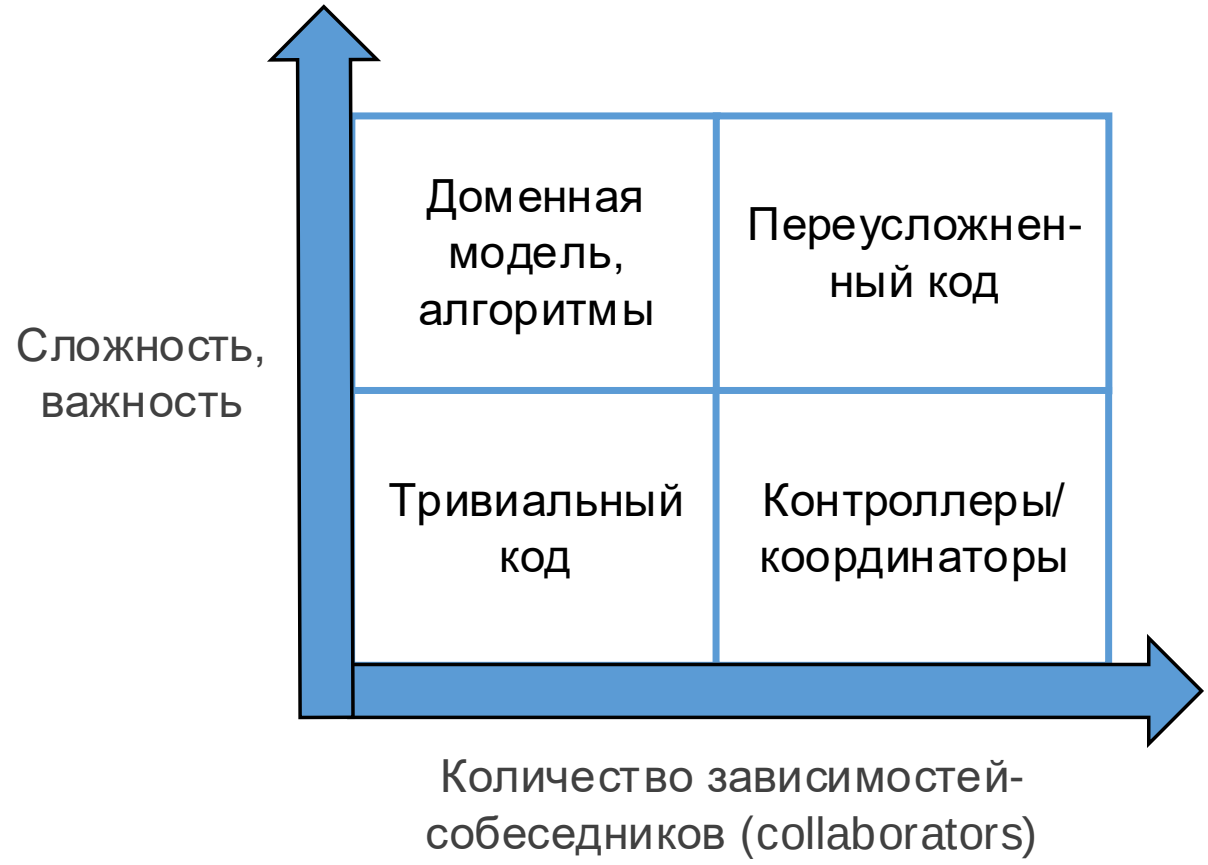
Сложность, важность

Количество зависимостей-собеседников (collaborators)

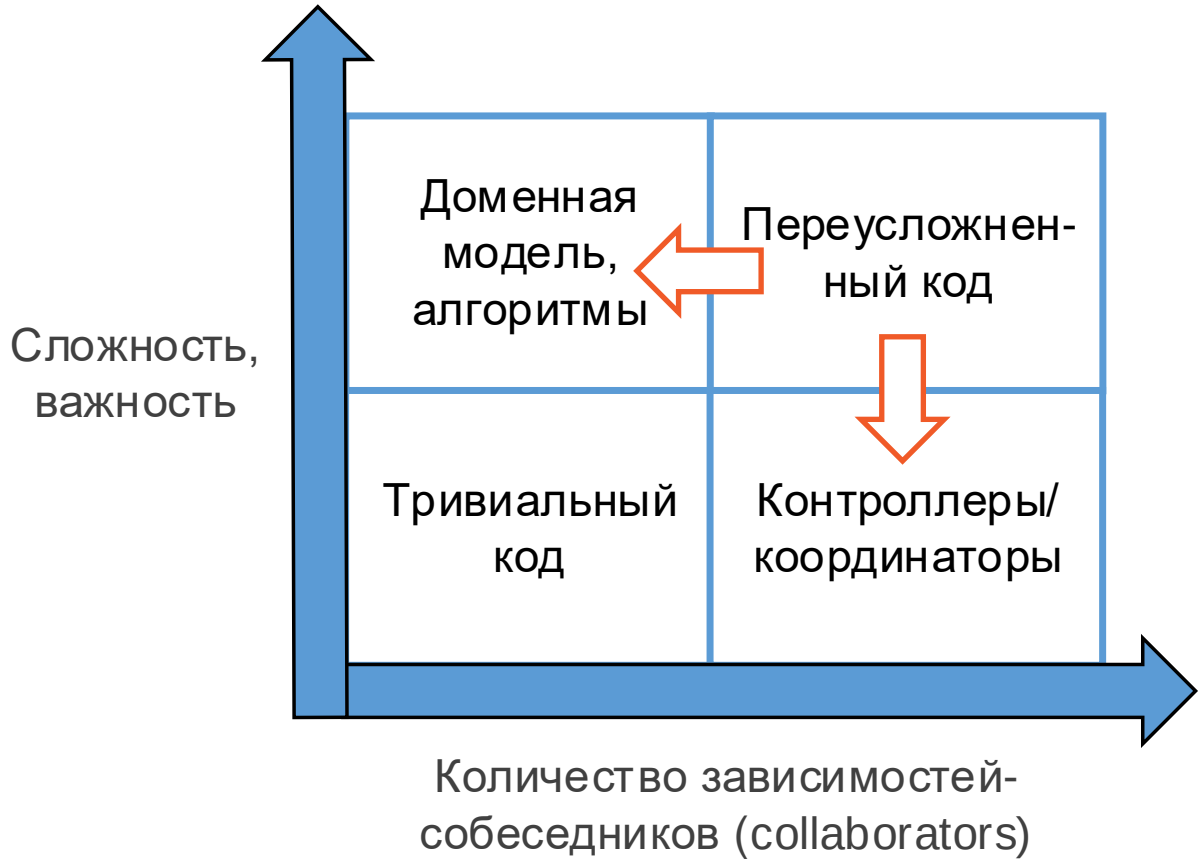
Хорошая защита от багов

Большая стоимость поддержки

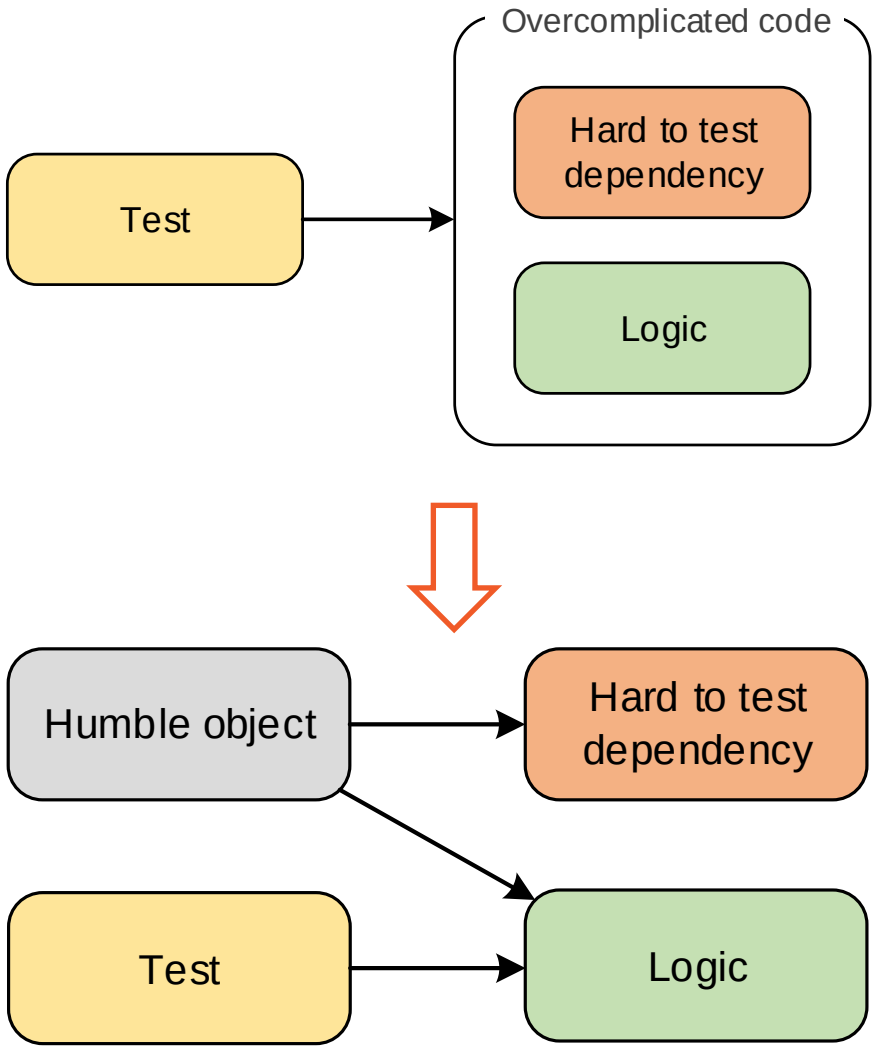
# Написание эффективных тестов



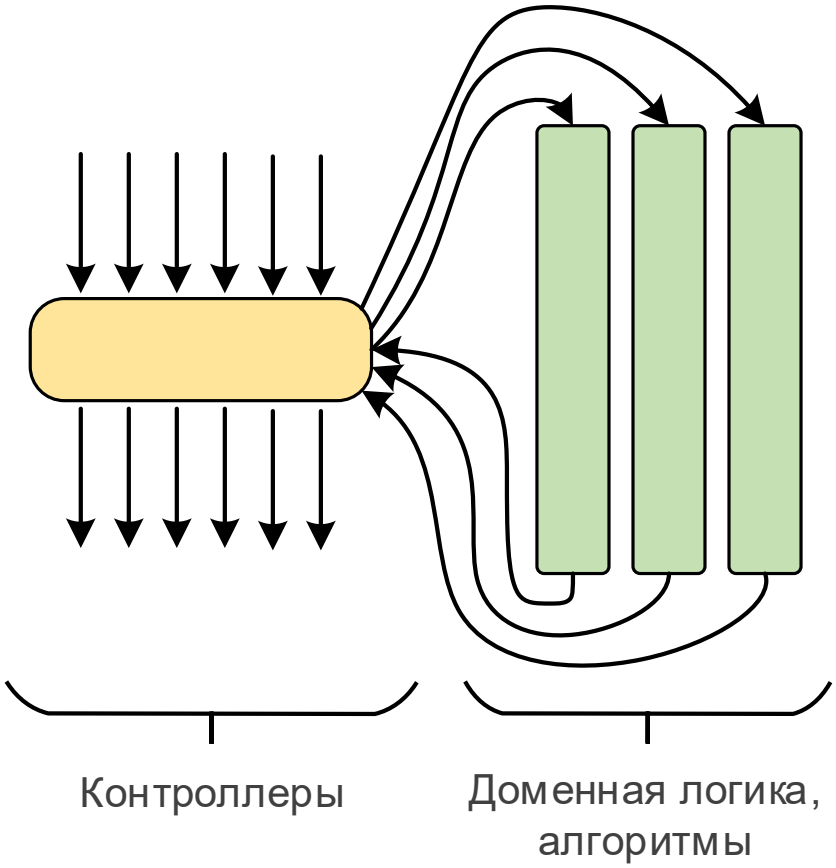
# Написание эффективных тестов



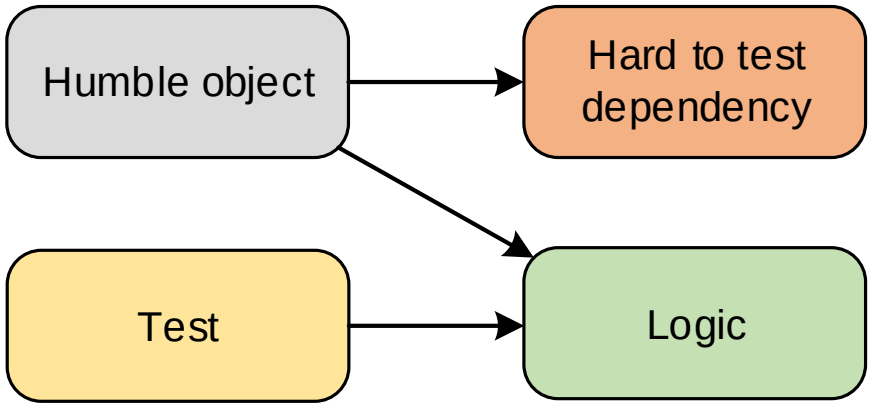
# Применение Humble Object паттерна



# Применение Humble Object паттерна



# Применение Humble Object паттерна



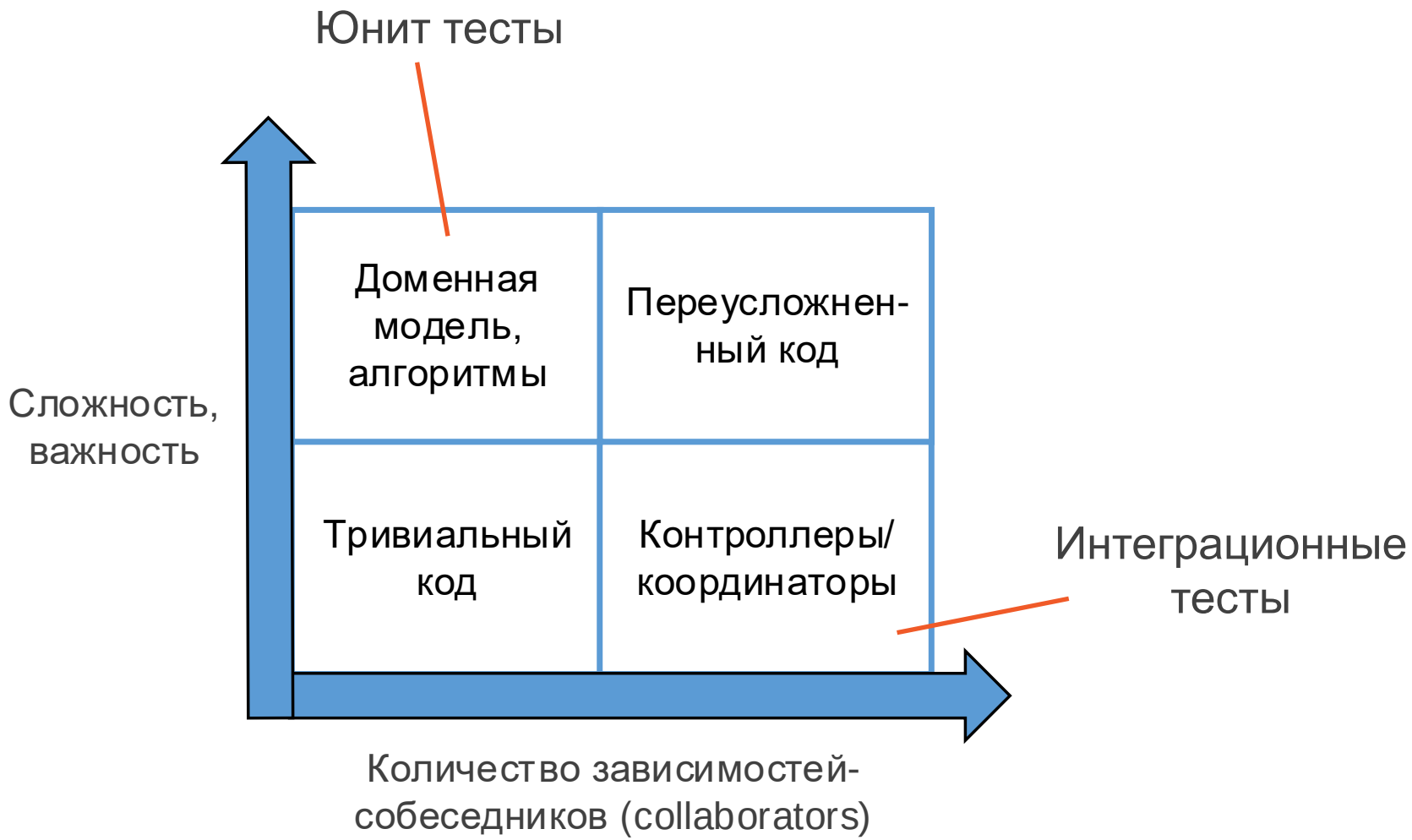

MVC (Model-View-Controller)

Model = Logic

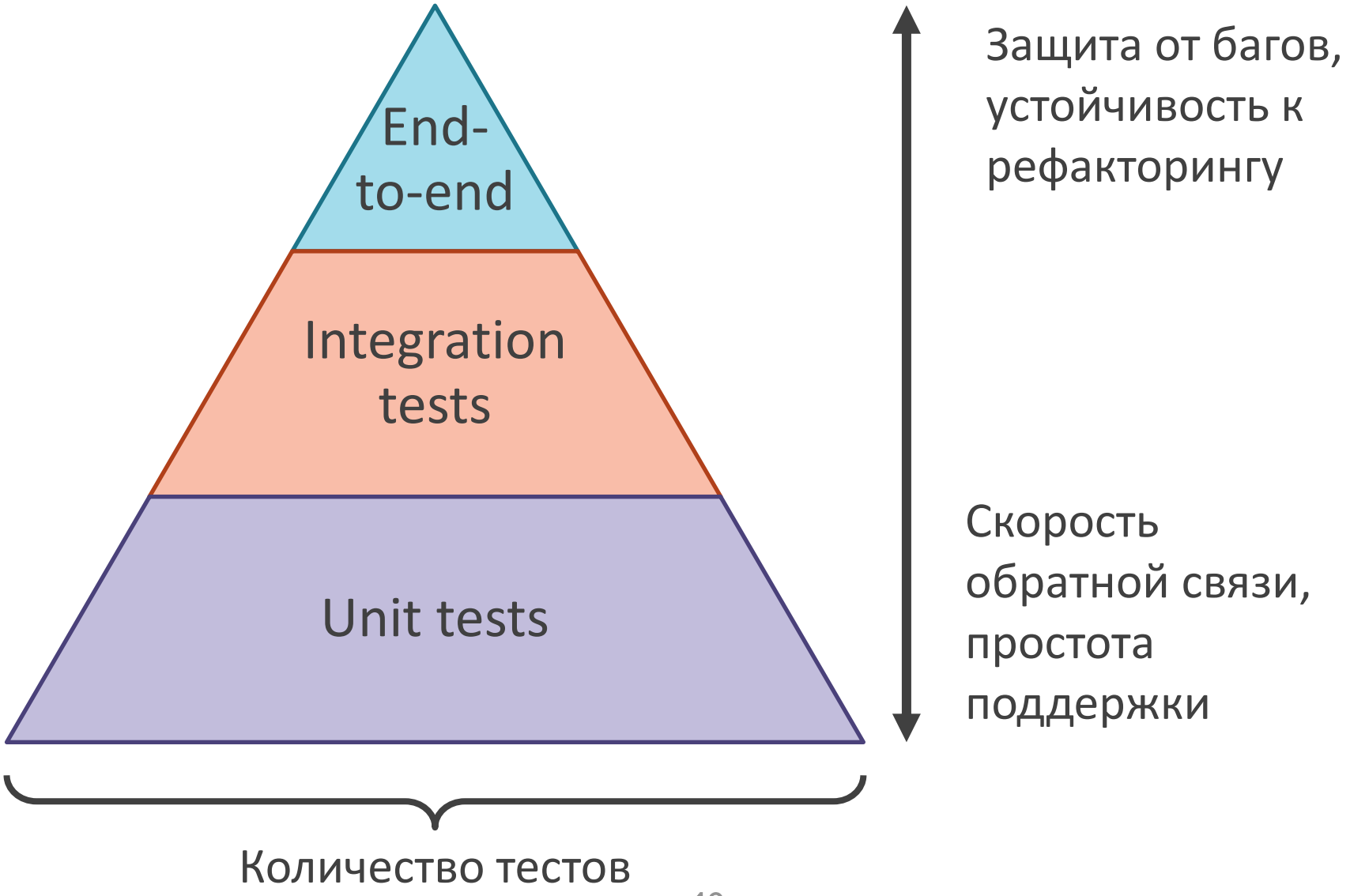
View = Hard to test dependency

Controller = Humble object

# Применение Humble Object паттерна

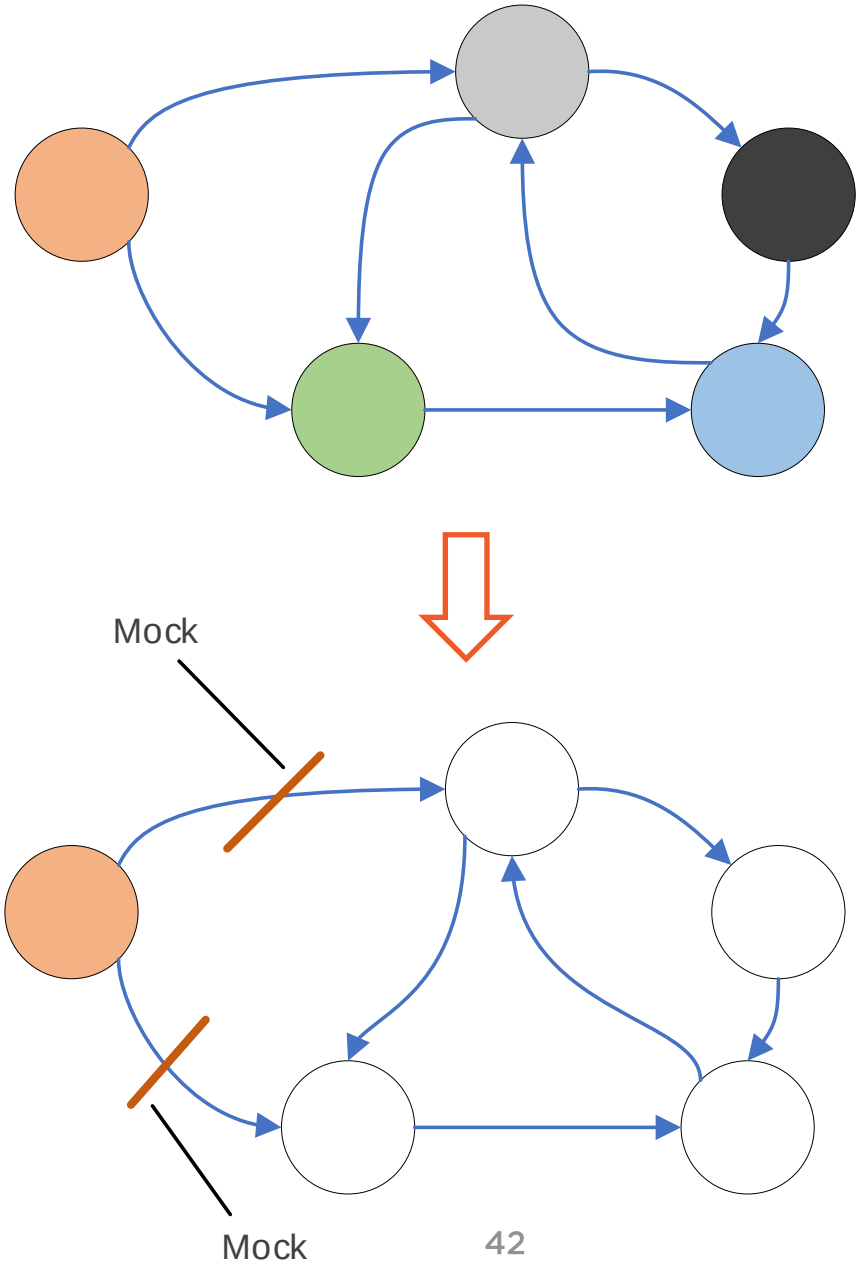


# Применение Humble Object паттерна

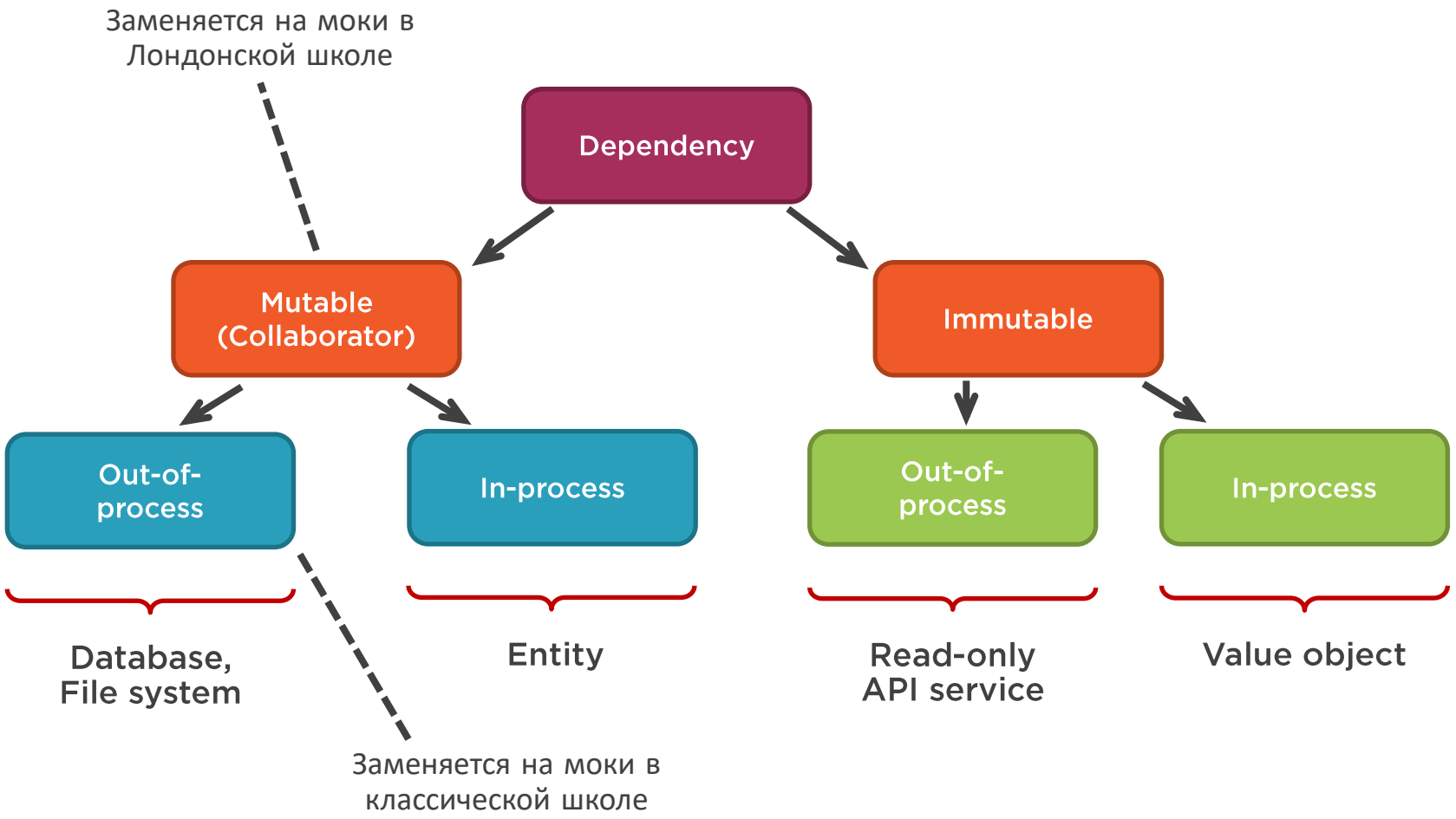


# План доклада

# Моки

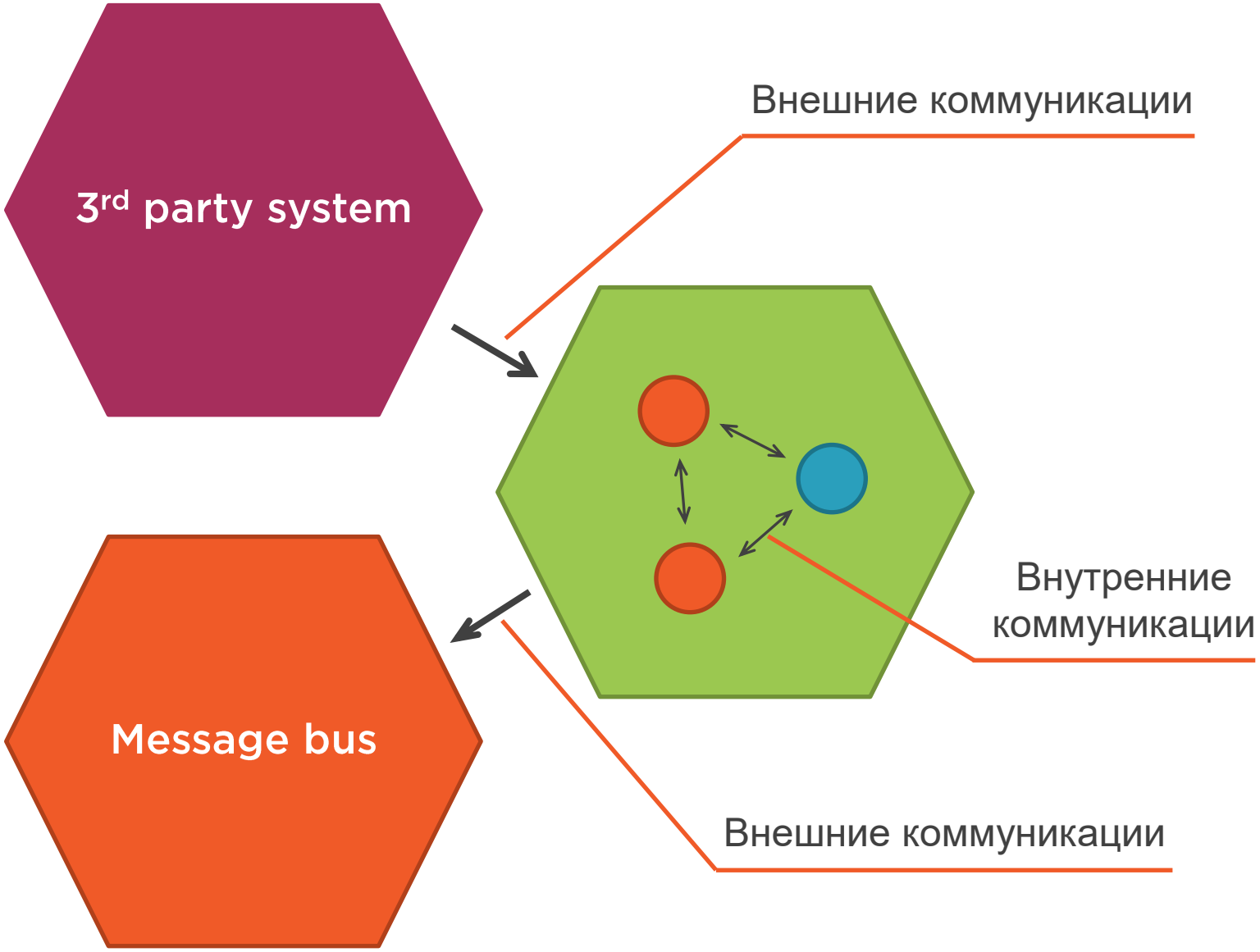


# Моки

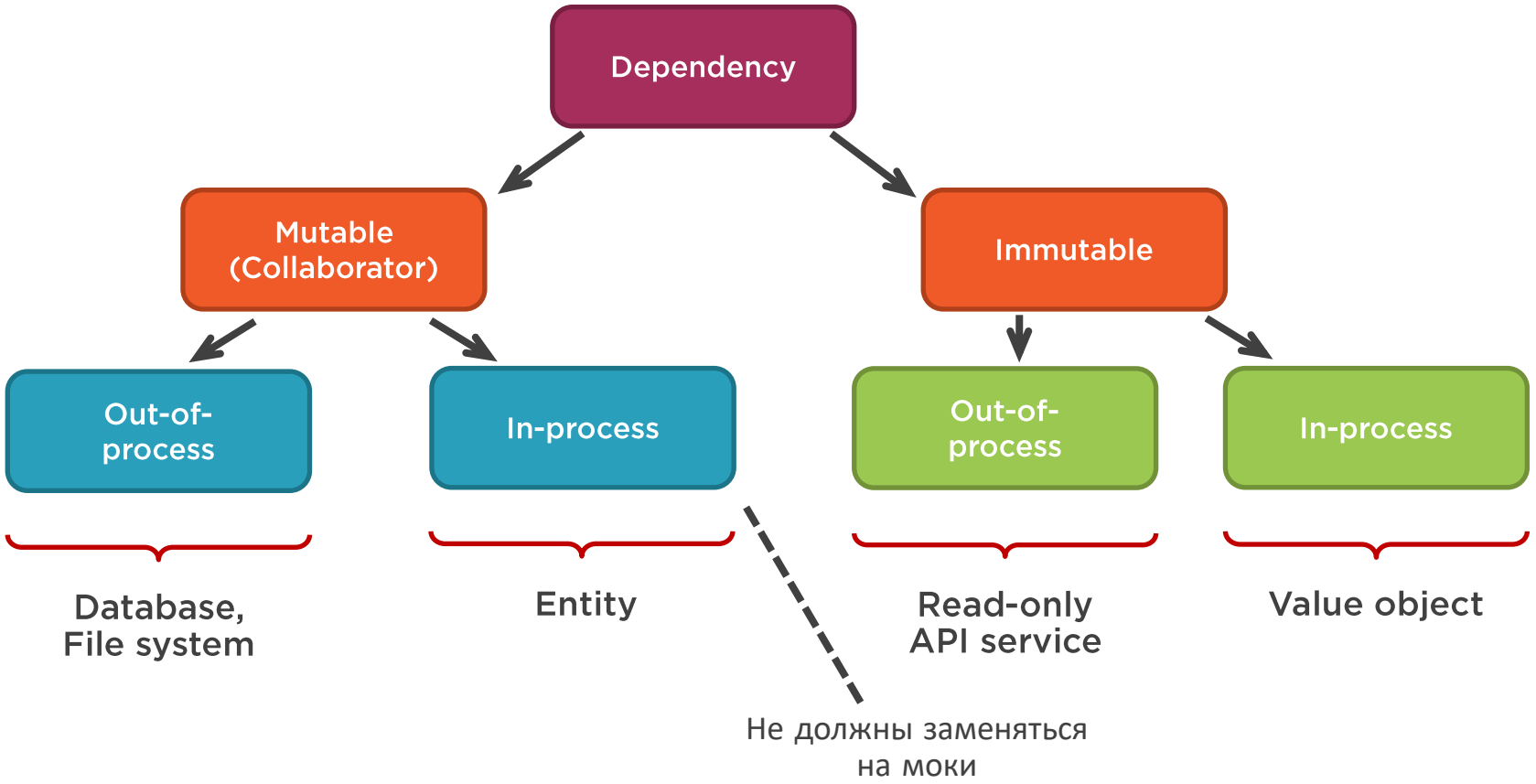



Обе школы неверны

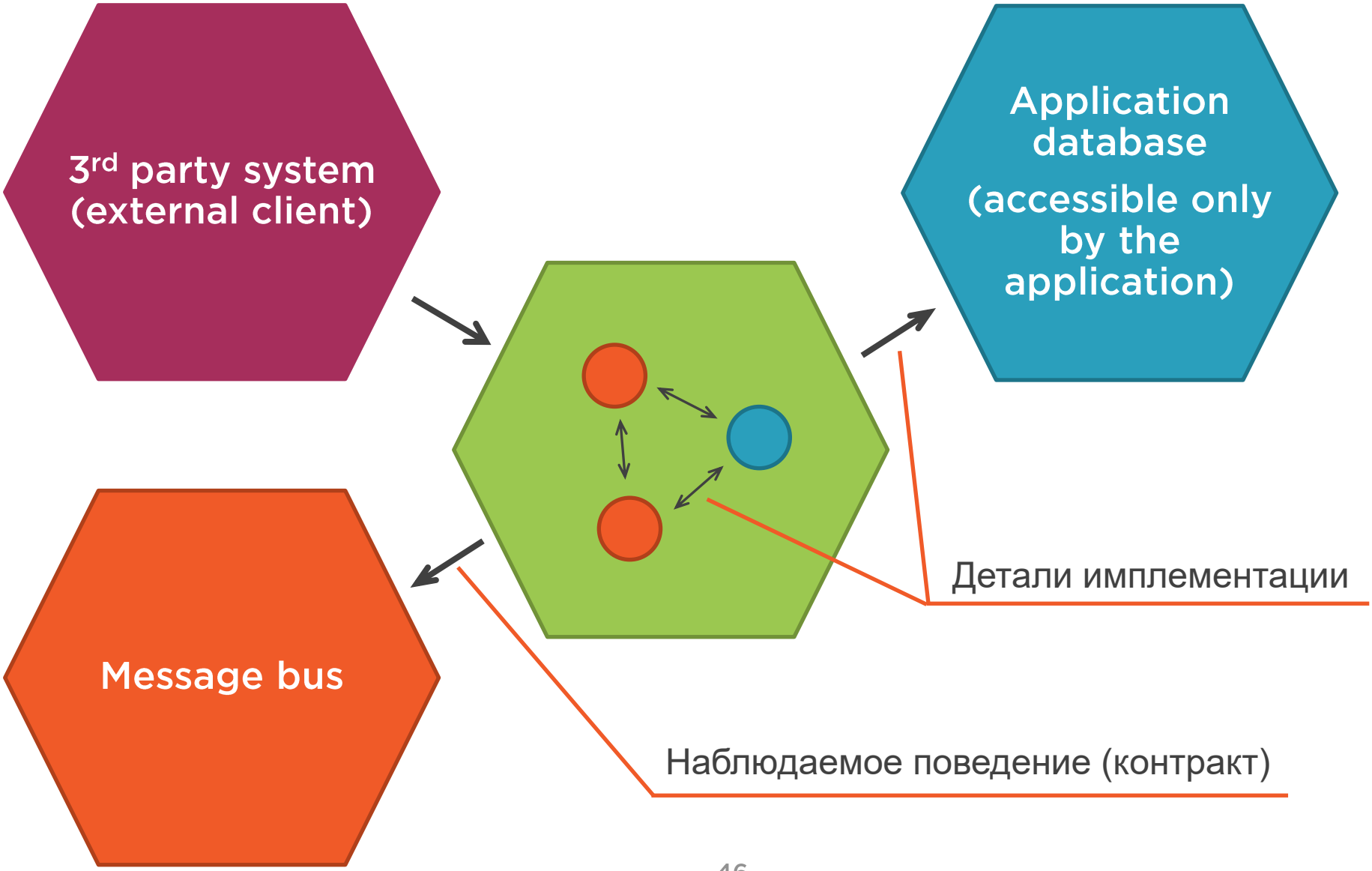
# Моки


3rd party system
Внешние коммуникации
Внутренние коммуникации
Message bus
Внешние коммуникации

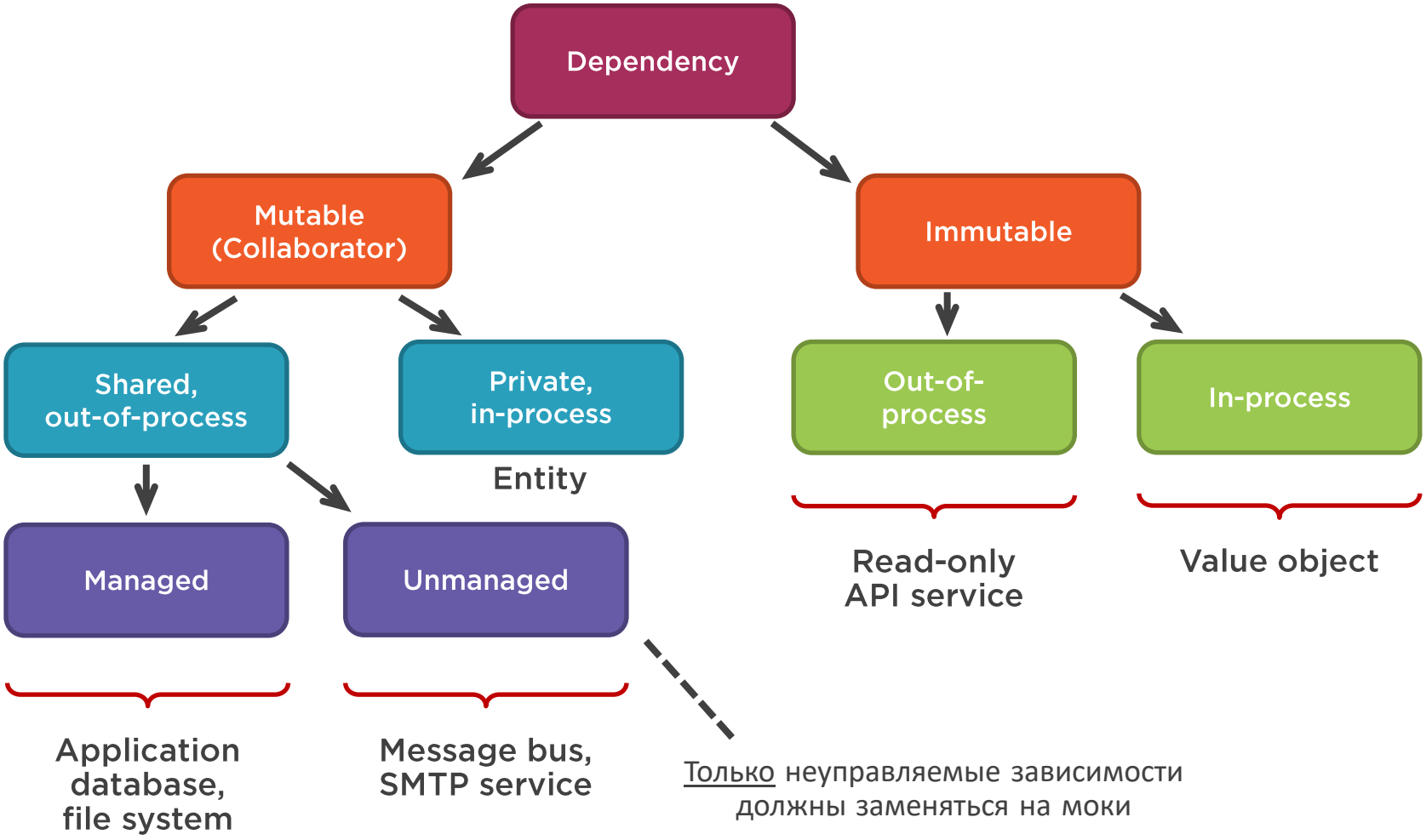
# Моки

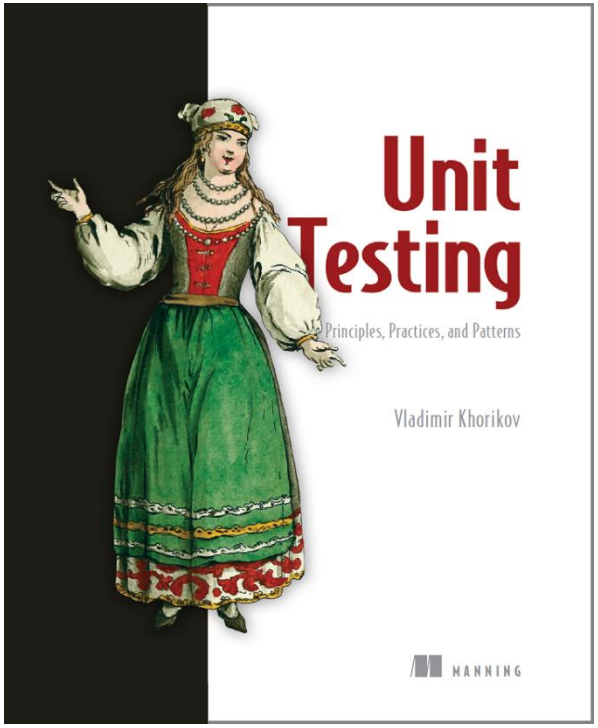

Dependency
Mutable (Collaborator)
Immutable
Out-of-process
In-process
Out-of-process
In-process
Database, File system
Entity
Read-only API service
Value object
Не должны заменяться на моки

# Моки


3rd party system (external client)
Application database (accessible only by the application)
Message bus
Детали имплементации
Наблюдаемое поведение (контракт)

# Моки


Dependency
Mutable (Collaborator)
Immutable
Shared, out-of-process
Private, in-process
Out-of-process
In-process
Entity
Managed
Unmanaged
Read-only API service
Value object
Application database, file system
Message bus, SMTP service
Только неуправляемые зависимости должны заменяться на моки




**Unit Testing Principles, Practices, and Patterns**

http://bit.ly/testing-book

**Contacts:**

https://enterprisecraftsmanship.com/

# Вопросы?