МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Evgeniy Shvetsov

2021

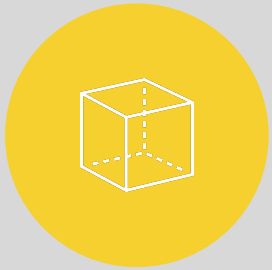
Модульное тестирование

МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

ТИП ТЕСТИРОВАНИЯ, ПРИ КОТОРОМ ТЕСТИРУЮТСЯ ОТДЕЛЬНЫЕ МОДУЛИ ИЛИ КОМПОНЕНТЫ СИСТЕМЫ.

ЦЕЛЬ

ПРОВЕРИТЬ ПРАВИЛЬНОСТЬ РАБОТЫ КАЖДОЙ ЕДИНИЦЫ ПРОГРАММНОГО КОДА



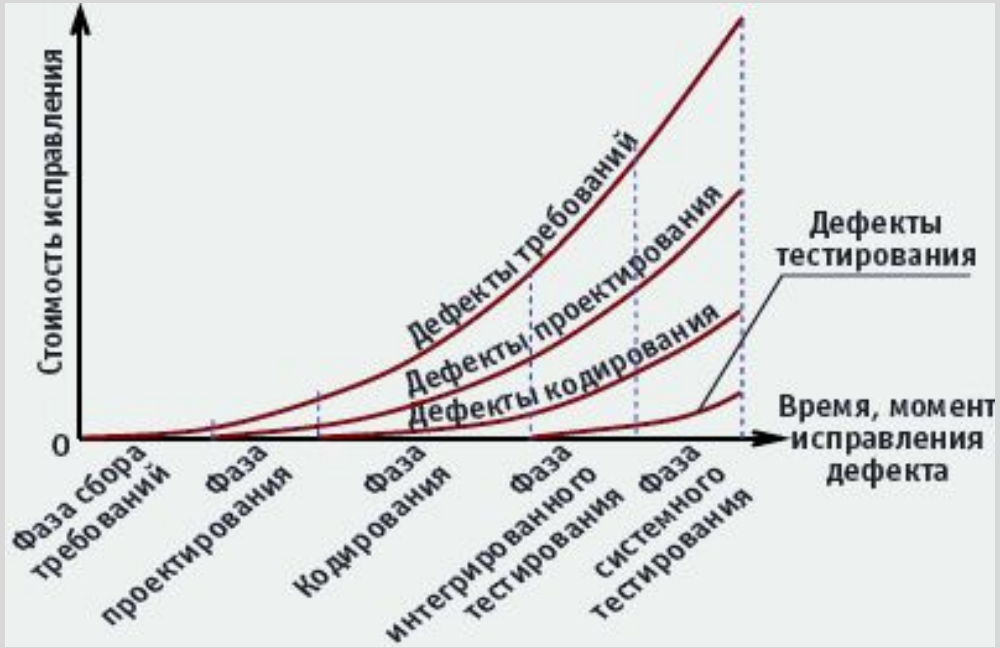
ЕДИНИЦА ТЕСТИРОВАНИЯ

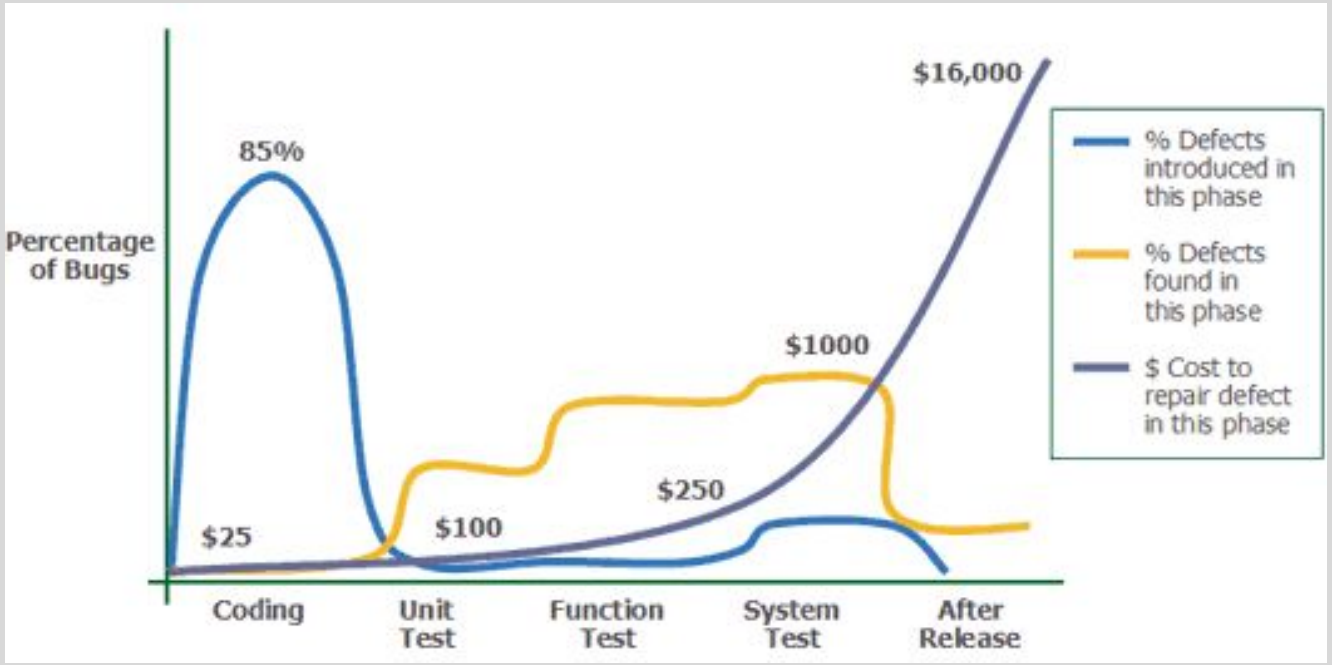
ФУНКЦИЯ, МЕТОД, ОБЪЕКТ ИЛИ МОДУЛЬ

ИСПОЛНИТЕЛЬ

РАЗРАБОТЧИК МОДУЛЯ

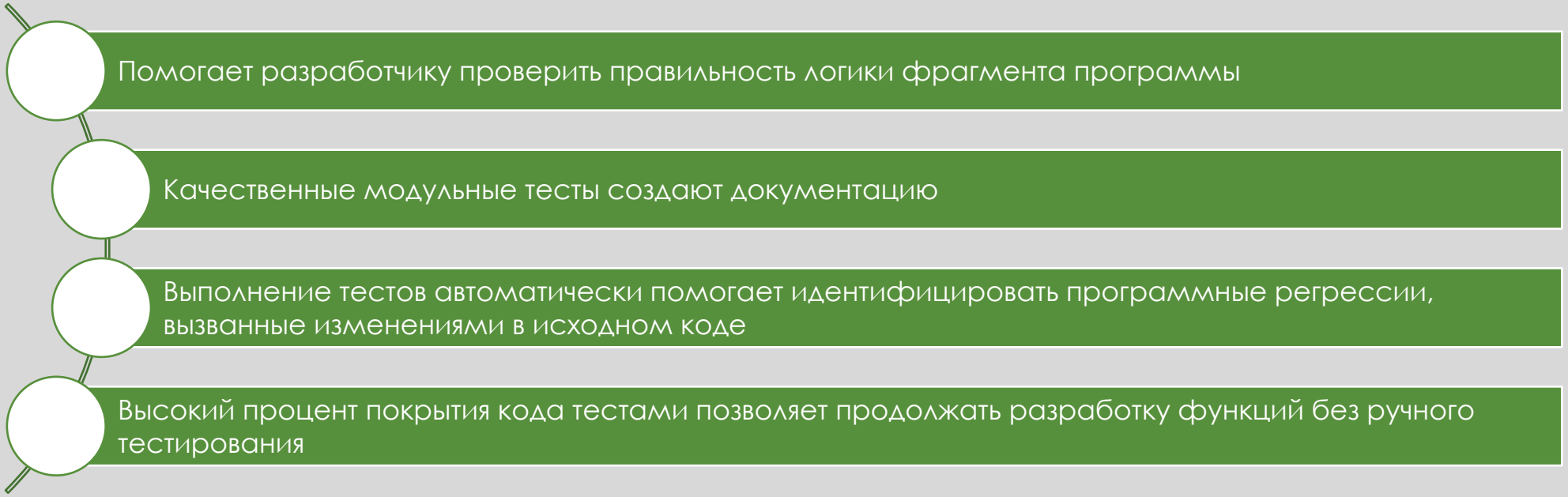
Важность модульного тестирования





Принципы модульного тестирования

- Тест – это часть кода, которая выполняет другой код и проверяет, приводит ли этот код к ожидаемому состоянию (тестирование состояния) или выполняет ожидаемую последовательность событий (тестирование поведения).



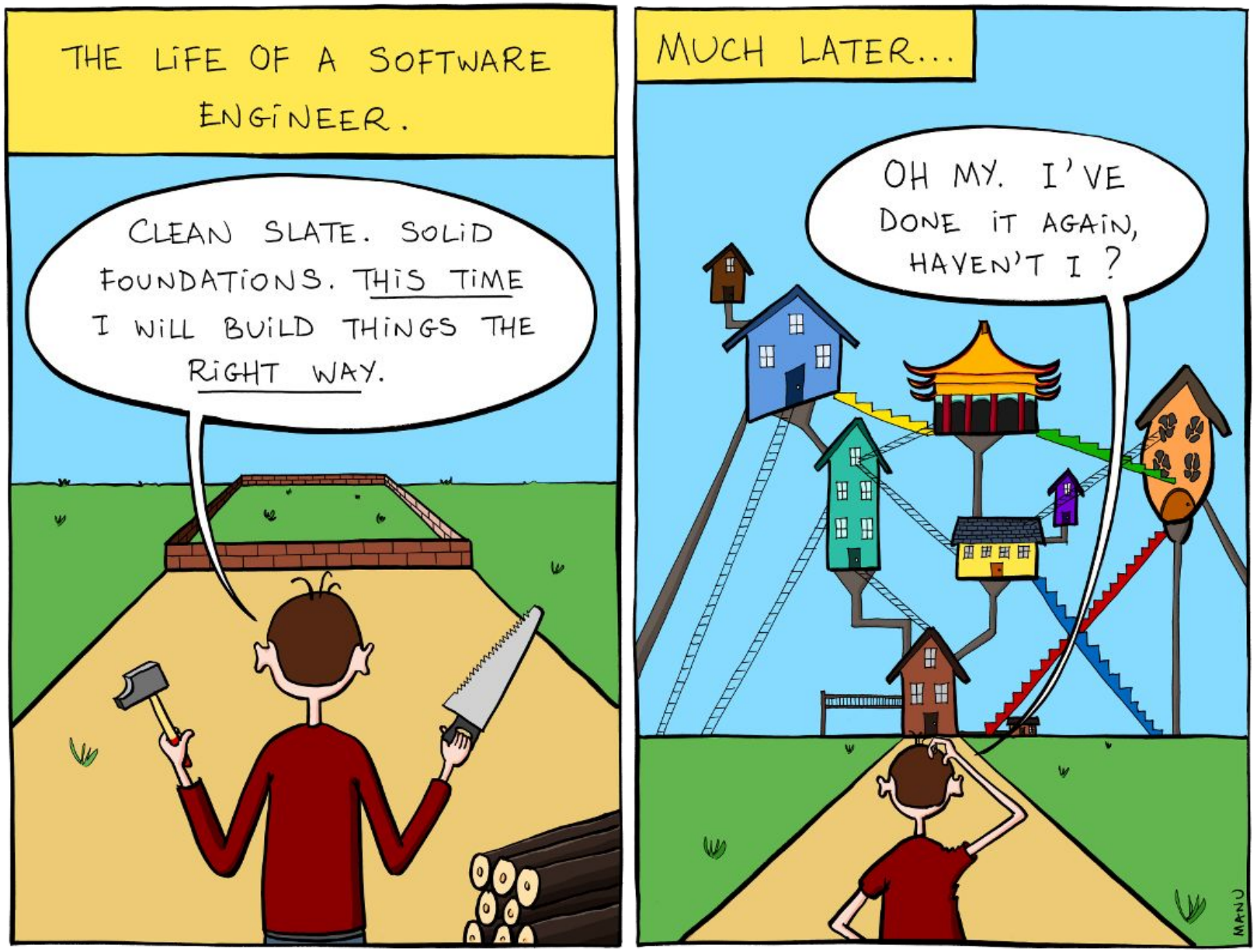
Преимущества и недостатки модульного тестирования

Преимущества

- Устранение ошибок на самых ранних этапах
- Атомарность разработки модулей
- Рефакторинг кода
- Тест есть документация

Недостатки

- Невозможность поиска всех ошибок модуля
- Отлавливает только низкоуровневые ошибки



Рефакторинг кода

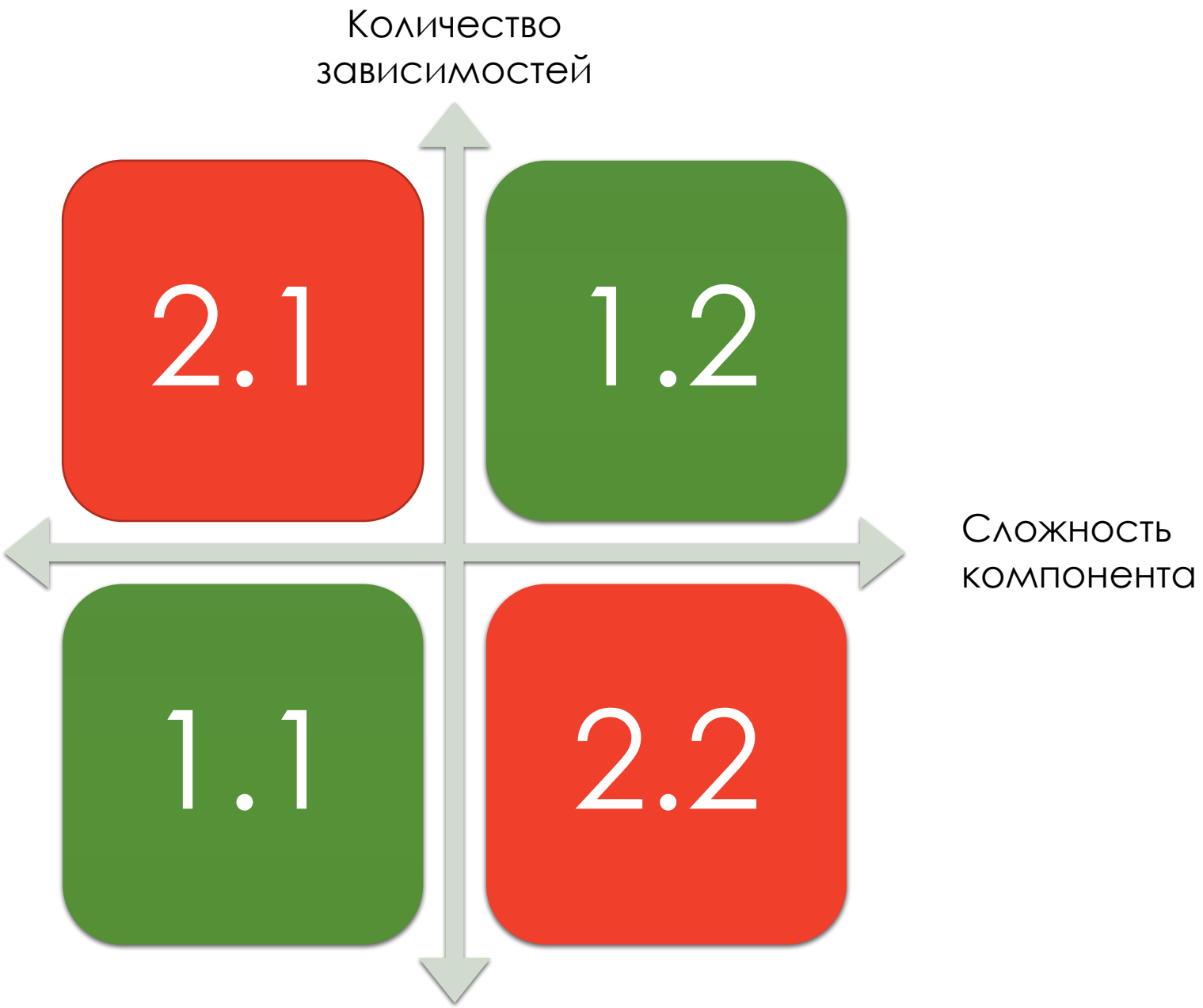
Рефакторинг кода – процесс улучшения кода без внесения изменений функциональности.

Грязный код:

- появляется в процессе частых изменений;
- низкая квалификация, лень;
- источник ошибок и непредсказуемого поведения

Чистый код:

- легкость чтения, понимания и поддержки;
- снижение вероятности появления ошибок;
- упрощение отлова и исправления ошибок;



Области тестирования

Тестирование нецелесообразно:

1.1 – простой код без зависимостей – код тривиальный и не предполагает ошибок

1.2 – сложный код с большим количеством зависимостей – код плохо скомплектован и требует пересмотра

Тестирование целесообразно:

2.1 – сложный код без зависимостей – обычно правильно описанная бизнес-логика

2.2 – простой код с зависимостями – обычно код, связывающий различные компоненты, соприкасается с интреграционном тестированием

ПРИНЦИПЫ F.I.R.S.T.

CLEAN CODE
BY ROBERT MARTIN

FAST

Быстрота = скорость выполнения + легкость понимания

Пути повышения быстроты тестов:

1. Один assert за тест;
2. Использование Dependency Injection;
3. Интуитивно-понятное именование;
4. Применение DSL (domain-specific language);

Проблема медленных тестов:

Запускаются редко

->

Повышается вероятность пропуска ошибки

->

«Дырявые» тесты требуют постоянной переработки

->

Поддержка тестов откладывается

->

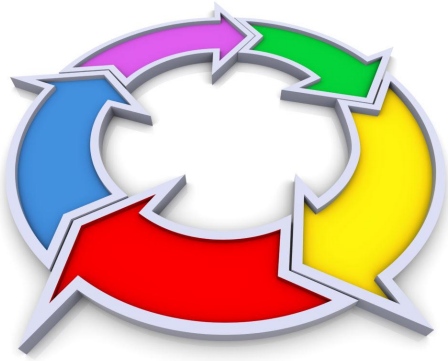
Повышается вероятность выхода ошибок в production.

INDEPENDENT ISOLATED

Независимость тестов достигается путем применения паттерна **BUILD – OPERATE – CHECK**.

BUILD – настройка теста
OPERATE – запуск тестируемой функциональности
CHECK – сравнение полученного результата с ожидаемым

REPEATABLE



Повторяемость достигается с помощью применения тестовых двойников (test doubles):

- **Dummy objects** – передаваемые, но неиспользуемые объекты;
- **Fake objects** – заведомо подмененные системы с аналогичной функциональностью, но упрощенные для использования в тестовом случае (InMemoryDatabases, HashTables, etc.);
- **Mocks** – заглушки, предназначенные для регистрации вызовов;
- **Stubs** – заглушки, отвечающие на вызовы согласно настроенному сценарию ответов;
- **Spies** – заглушки (аналогично Stubs), но позволяющие регистрировать действия;

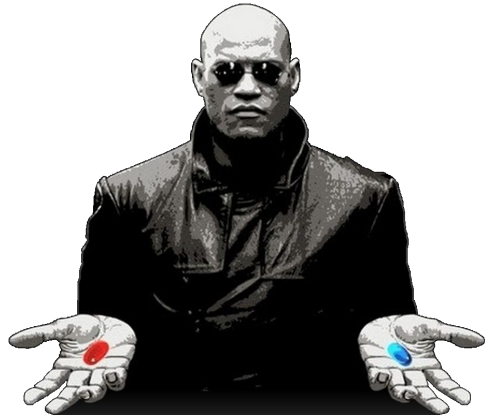
* by Martin Fowler

SELF-VALIDATING

Результат теста должен представлять собой булево значение и не требовать дополнительной интерпретации.

<u>Принцип достигается, используя:</u>

1. Булев результат;
2. Один assert на тест;
3. Build – Operate – Check паттерн;



True **False**

TIMELY

Значение новых модульных тестов падает пропорционально объему написанного кода.

Высшее выражение принципа – TDD подход.

Эволюция модульного тестирования

1st Generation

boolean assert(boolean condition);

- JUnit before v.3

2nd Generation

boolean assertEquals(…);
boolean assertTrue(…);
boolean assertFalse(…);
etc;

- JUnit v.3+

3rd Generation

void assertThat(…);

- Hamcrest
- AssertJ

Фреймворк JUnit

JUnit – фреймворк автоматизированного модульного тестирования для Java. Также портирован на многие языки программирования.

Разработчики: Кент Бек, Эрих Гамма

Платформа: Java

Основные особенности JUnit-тестов:

- регрессионность;
- повторяемость

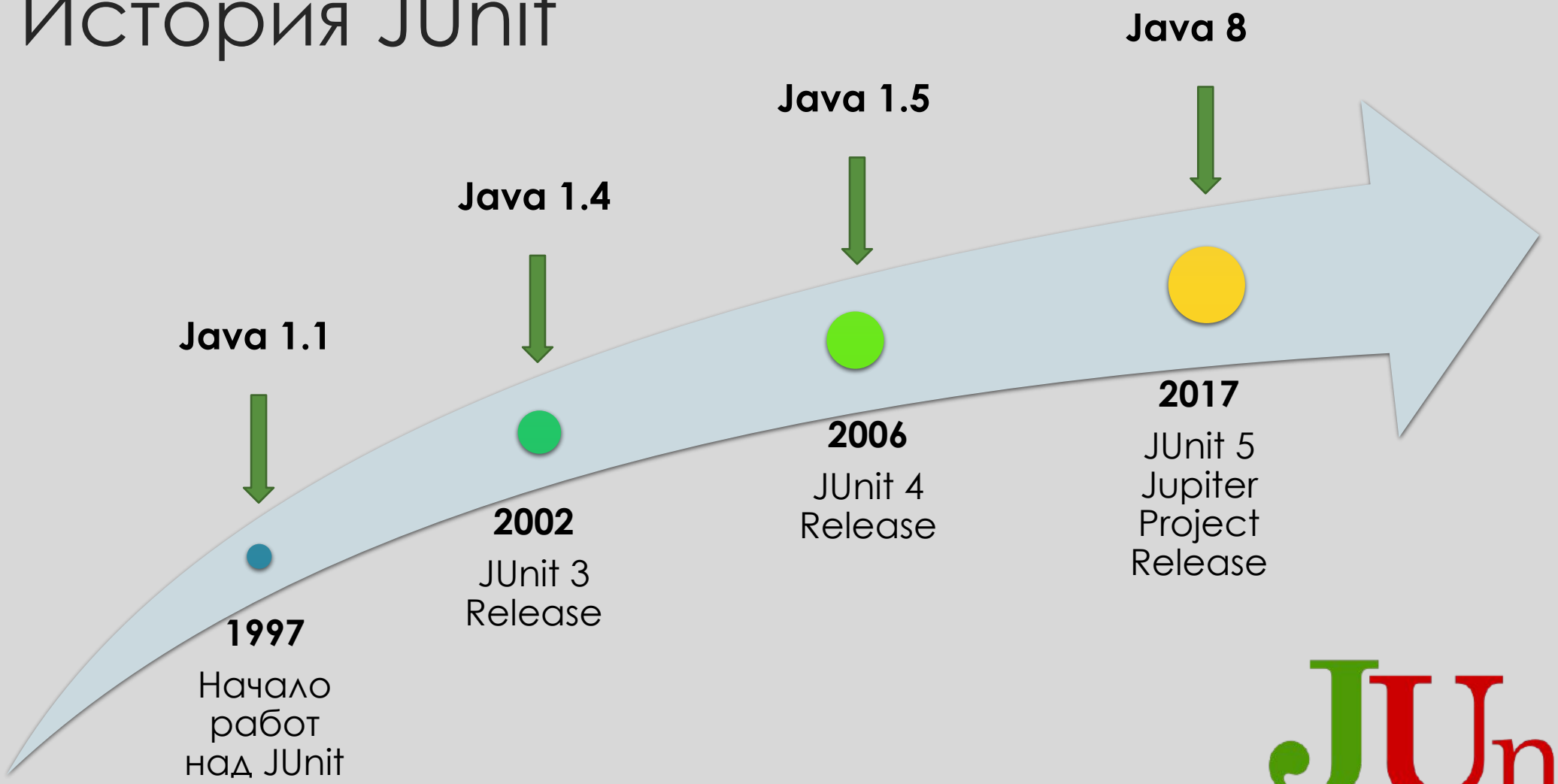
Преимущества:

- доступен для написания модульных и интеграционных тестов;
- является хорошим средством документирования кода;
- совместимость с популярными IDE;
- использует синтаксис Java;

Недостатки:

- Не осуществляет тестирование зависимостей;

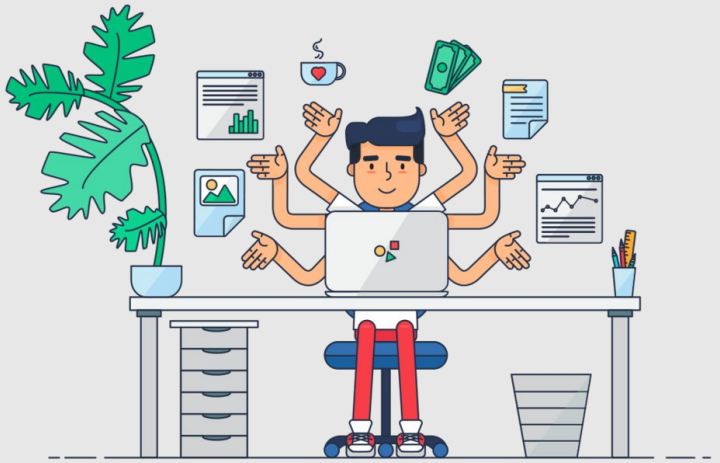
История JUnit



JUnit

Аннотация	Описание
@Test	Определяет метод как метод тестирования
@Test(expected = Exception.class)	Сбой, если метод не генерирует исключение
@Test(timeout = 100)	Сбой, если метод занимает более 100 миллисекунд
@BeforeClass *static*	Метод выполняется до начала выполнения всех тестов. Выполнение операций инициализации.
@AfterClass *static*	Метод выполняется после выполнения всех тестов. Выполнение операций по очистке ресурсов.
@Before	Метод выполняется перед каждым тестом. Применяется для подготовки тестовой среды.
@After	Метод выполняется после каждого теста. Применяется для очистки тестовой среды.
@Ignore("Some message")	Позволяет игнорировать выполнение теста.
@Rule	Создание правила – трансциндентного объекта.

JUnit 4
Annotations



Assertion	Описание
assertTrue(message *, condition)	Проверяет, является ли condition true.
assertFalse(message *, condition)	Проверяет, является ли condition false.
assertEquals(message *, expected, actual, tolerance *)	Проверяет, эквивалентны ли значения expected и actual, tolerance применяется для значений с плавающей точкой.
assertNull(message *, object)	Проверяет, является ли объект null
assertNotNull(message *, object)	Проверяет, является ли объект not null.
assertSame(message *, expected, actual)	Проверяет объекты на эквивалентность ссылок.
assertNotSame(message *, expected, actual)	Проверяет объекты на неэквивалентность ссылок.
assertThrows(message *, Exception.class, runnable)	Проверяет на выбрасывание исключения в runnable.

JUnit 4
Assertions

ЗАМЕТКИ НА ПОЛЯХ

JUnit 4

```
Assert.assertEquals(null, object);
Assert.assertEquals(true, object.isTrue());
Assert.assertEquals(false, object.isFalse());
```

```
Assert.assertNull(object);
Assert.assertTrue(object.isTrue());
Assert.assertFalse(object.isFalse());
```

```
Assert.assertTrue(object == null);
Assert.assertTrue(object.isTrue());
Assert.assertTrue(object.isFalse());
```

```
Assert.assertNull(object);
Assert.assertTrue(object.siTrue());
Assert.assertFalse(object.isFalse());
```

Фреймворк Junit 5

JUnit 5

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

JUnit Platform – платформа для запуска TestEngine API.

JUnit Jupiter – проект для написания и запуска тестов, написания собственных расширений.

JUnit Vintage – проект для обеспечения совместимости с предыдущими версиями JUnit.

Основные нововведения JUnit 5:

- Все тестовые методы публичные, модификатор доступа public опускается;
- Аргумент message в выражениях assert поставлен последним;
- Группировка тестов с помощью assertAll();
- Новый assert для работы с Iterable;
- Новый assert для работы со строками, поддерживающий regexp'ы;
- Работа с исключениями через assertThrows();
- Создание вложенных тестовых классов через аннотацию @Nested;
- Параметризация тестов;
- Механизм расширений, заменивший правила;
- Обновление базовых аннотаций;

JUnit 5
Annotations

Аннотация	Описание
@Test	Определяет метод как метод тестирования
@DisplayName	Переопределяет отображаемое имя тестового метода
@BeforeAll *static*	Метод выполняется до начала выполнения всех тестов. Выполнение операций инициализации.
@AfterAll *static*	Метод выполняется после выполнения всех тестов. Выполнение операций по очистке ресурсов.
@BeforeEach	Метод выполняется перед каждым тестом. Применяется для подготовки тестовой среды.
@AfterEach	Метод выполняется после каждого теста. Применяется для очистки тестовой среды.
@Disabled("Some message")	Позволяет игнорировать выполнение теста.
@RepeatedTest(number)	Осуществляется number повторений теста.

Одно из нововведений JUnit 5 – расширения (extensions), заменяющие правила (rules) JUnit 4.

JUnit 5 определяет 5 типов расширений:

- **test instance post-processing** – расширение применяется после создания сущности теста;
- **conditional test execution** – определяет, будет ли запускаться тест
- **life-cycle callbacks** – связан с вызовом методов обратного вызова;
- **parameter resolution** – позволяет работать с параметризованными методами;
- **exception handling** – позволяет определять логику в случае выбрасывания исключения.

JUnit 5
Extensions

Библиотеки модульного тестирования

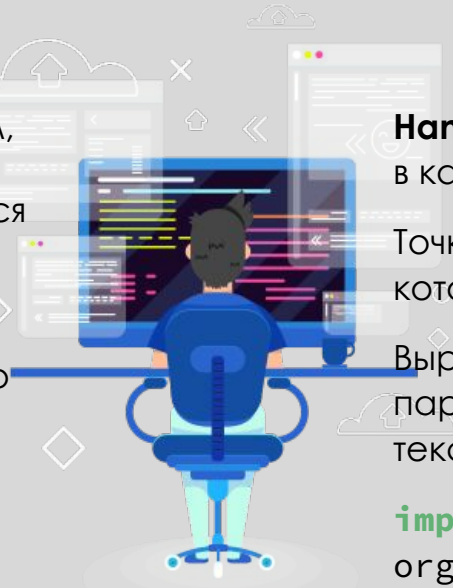
Библиотеки

AssertJ

AssertJ – библиотека с открытым исходным кодом, предназначенная для написания наглядных и интуитивно понятных assert'ов. Активно применяется для тестирования совместно с JUnit.

Точка входа в выражения AssertJ – assertThat(), в которым передаются тестируемые данные. Само выражение не выполняет проверку, а позволяет строить проверочные цепочки.

```
import static
org.assertj.core.api.Assertions.*;
```



Hamcrest

Hamcrest – библиотека, работающая совместно с JUnit в качестве имплементации механизма проверок.

Точка входа в выражения Hamcrest – assertThat(), в которую передаются тестируемые данные.

Выражение в общем случае получает в виде параметров исследуемый объект, условия проверки и текст сообщения.

```
import static
org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;
```

СПАСИБО ЗА ВНИМАНИЕ!

МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Evgeniy Shvetsov

2021