UNIT TESTING

06.11.2018

ПРЕДНАТА ЛЕКЦИЯ ГОВОРИХМЕ ЗА:

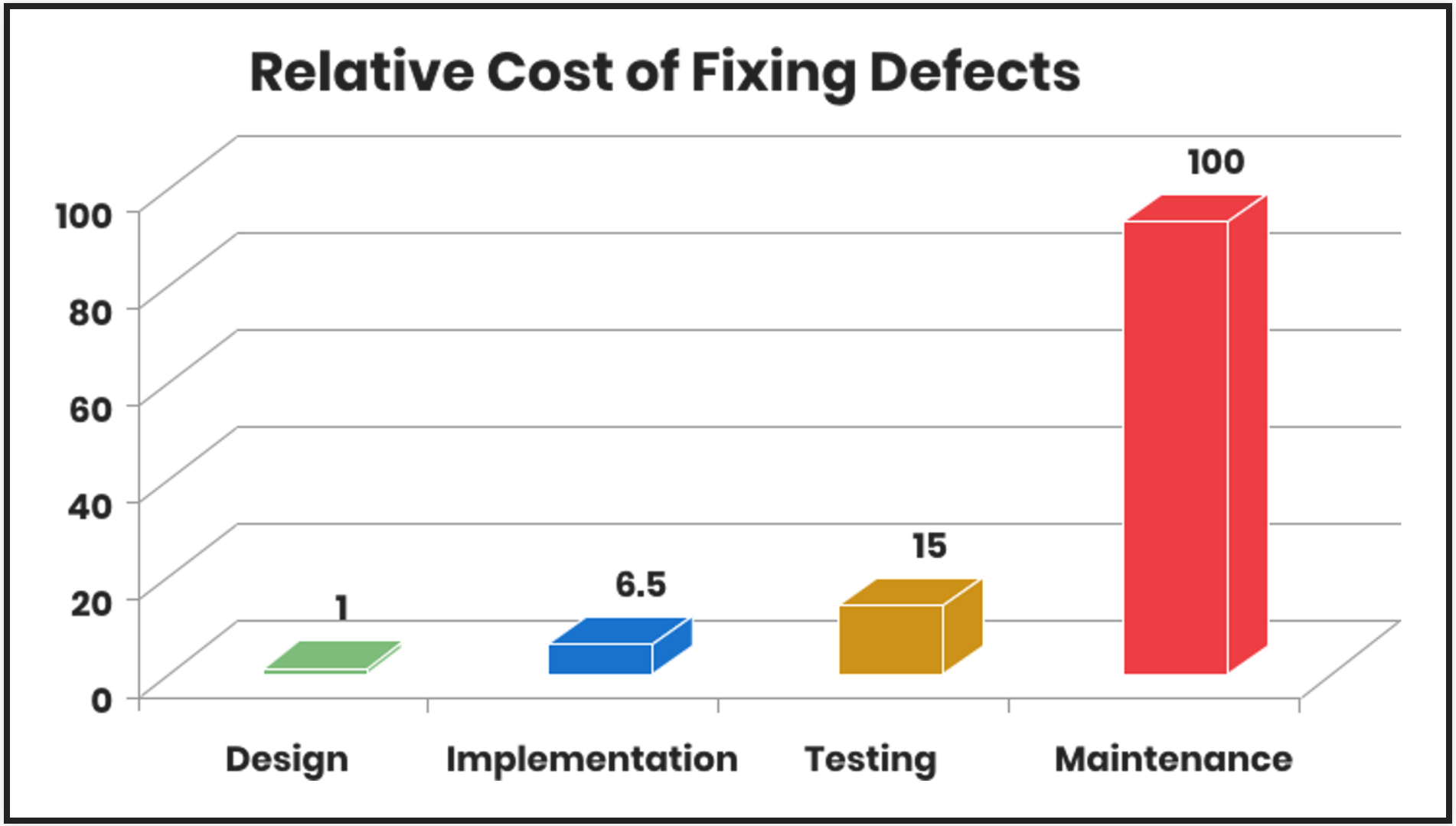
- Колекции (collections)
- Шаблонни типове (generics)

ДНЕС ЩЕ РАЗГЛЕДАМЕ:

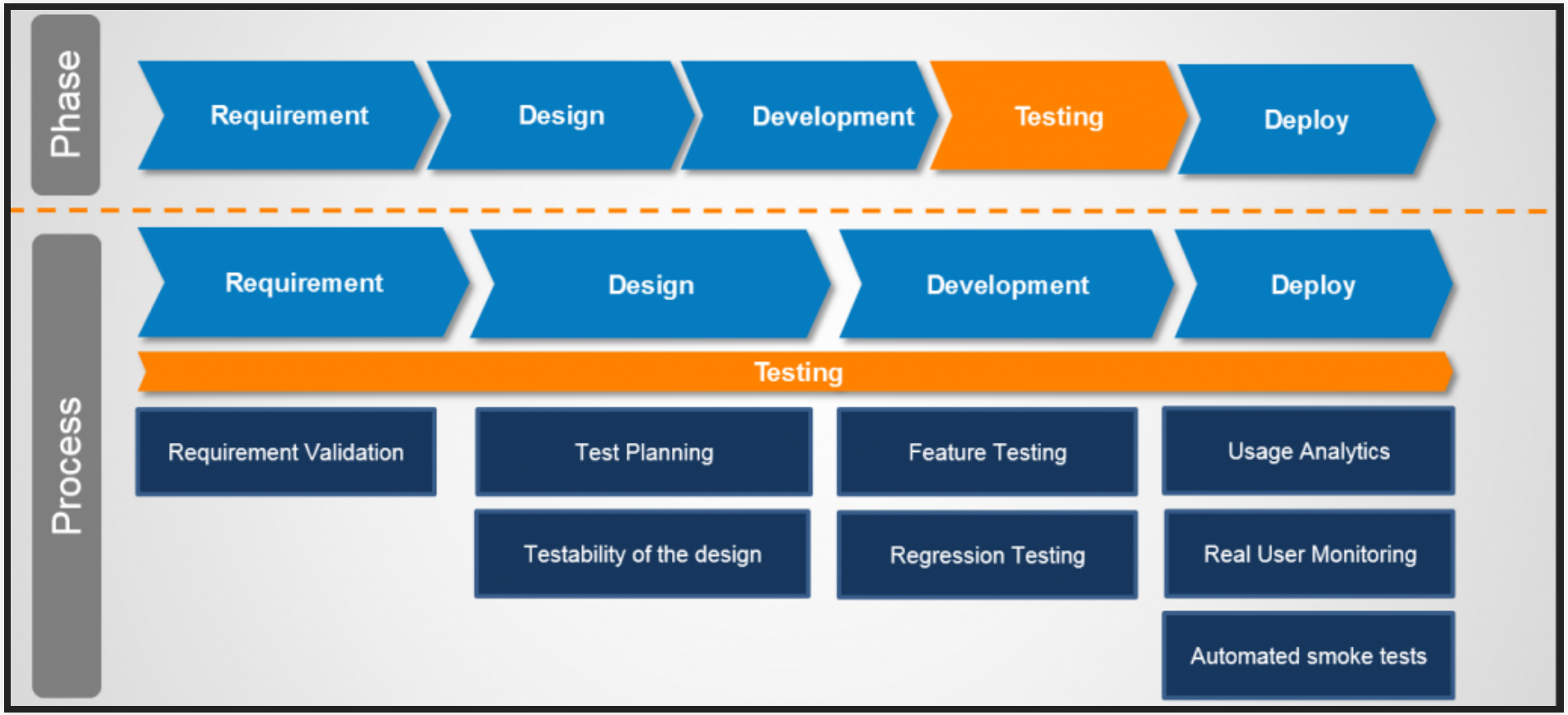
- Как се тества софтуер, и кому е нужно
- Какво е unit testing
- JUnit
- Test-Driven Development (TDD)
- Stubbing и mocking

ЗАЩО Е НУЖНО ДА СИ ТЕСТВАМЕ КОДА?

КОГА ДА ТЕСТВАМЕ?



ТЕСТВАНЕТО НЕ Е ФАЗА, А Е ПРОЦЕС



ОСНОВНИ ВИДОВЕ ТЕСТОВЕ

- Ръчни:
  - В професионалната софтуерна разработка, липса на автоматични тестове == липса на тестове въобще
- Автоматични:
  - функционални
  - нефункционални

- Функционални тестове
  - unit тестове
  - integration тестове
- Нефункционални тестове
  - performance тестове
  - stress тестове
  - crash тестове
  - security тестове
  - usability тестове

ОЩЕ ВИДОВЕ ТЕСТОВЕ

- Alpha
- Acceptance
- Ad-hoc
- Accessibility
- Beta
- Back-end
- Browser compatibility
- Backward compatibility
- Black box
- Boundary value
- Branch
- Comparison
- Compatibility
- Component
- End-to-end
- Equivalence Partitioning
- Example
- Exploratory
- Functional
- GUI
- Gorilla
- Happy path
- Incremental integration
- Install/uninstall
- Integration
- Load
- Monkey
- Mutation
- Negative
- Non-functional
- Performance
- Recovery
- Regression
- Risk-based
- Sanity
- Security
- Smoke
- Static
- Stress
- System
- Unit
- Usability
- Vulnerability
- Volume
- White box

UNIT TESTING

- Unit test е код, който изпълнява специфична, „атомарна“ (т.е. която не може да се разбие по смислен начин на по-малки) функционалност на кода, която да бъде тествана
- Един unit test цели да тества малък фрагмент код - обикновено един метод или най-много един клас

- Unit тестовете гарантират, че кодът работи както очакваме.
- Подсигуряват, че кодът ще продължи да работи както се очаква, в случай че го модифицираме, за да оправим бъг, рефакторираме или разширяваме функционалността

МАЛКО ДЕФИНИЦИИ

- *Продуктивен код* (a.k.a. *code under test*) – това е кодът, който реализира потребителските изисквания и удовлетворява сценариите на клиентите
- Процентът на продуктивния код, който се тества от автоматични тестове се нарича *test coverage* или *code coverage*. Високият test coverage на кода ни дава увереност да разработваме функционалности без да се налага да правим много ръчни тестове

- *Test Driven Development* (TDD) е методология, при която кодът на тестовете се пише преди продуктивния код, така че щом даден тест бъде удовлетворен (т.е. минава успешно, стане „зелен“), съответният use-case е реализиран („done“)

ОЩЕ ДЕФИНИЦИИ

- *Test fixture* е фиксирано състояние на софтуера, който тестваме, което е началното условие (предусловието) за изпълняване на тестовете
- *Integration test* тества поведението на компонент или интеграцията между множество компоненти

- Терминът *функционален тест* понякога се ползва като синоним на integration test
- *Performance test* измерва бързодействието (ефективността) на даден софтуер по repeatable начин

JUNIT

JUNIT FRAMEWORK

- JUnit е най-популярният и *de facto* стандартният testing framework в Java
- Актуалната версия е JUnit 5
- Засега в курса ще ползваме JUnit 4 (защо?)

ЗАЩО НИ ТРЯБВАТ TESTING FRAMEWORKS?

- Улесняват ни да пишем и изпълняваме тестове
- Стандартизират разработката и поддръжката на тестове

JUNIT

- JUnit се базира на анотации
- Всеки JUnit тест е метод, анотиран с `@Test`, съдържащ се в клас, който се използва само за тестване
- Такъв клас се нарича Test Case

ПРИМЕР ЗА JUNIT ТЕСТ

```
import static org.junit.Assert.*;

@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {
    // MyClass is tested
    MyClass tester = new MyClass();

    // Tests
    assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0)
    assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10)
    assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
}
```

КОНВЕНЦИИ ЗА ИМЕНУВАНЕ

- Прието е да се добавя `Test` към името на класа, който се тества
- Една конвенция за имената на тестовите методи е да се използва думата „should“, например `ordersShouldBeCreated()` или `menuShouldGetActive()`
- Името трябва да описва какво проверява теста

- Друга популярна конвенция е методите да се кръщават

`given[Input]When[Condition]Then[Expected]`

```
givenHomePageWhenUserClicksLoginButtonThenLoginPageShouldBeRen
givenNegativeNumbersWhenUserAppliesMultiplicationThenResultSho
givenVirtualMachineWhenUserDeletesItThenVolumeShouldBeDeletedA
```

СТАТИЧНИ МЕТОДИ НА ASSERT КЛАСА

JUnit предоставя статични методи в класа `org.junit.Assert` за тестване на определени условия

fail()

- `fail(String message)` – фейлва теста
- Може да се използва за проверка, че определена част от кода не се достига или като временна dummy имплементация, която да се замести от реален тест

- `assertTrue(String message, boolean condition)` – проверява, че булевото условие е истина
- `assertFalse(String message, boolean condition)` – проверява, че булевото условие е лъжа

- `assertNull(String message, Object o)` – проверява, че обектът е null
- `assertNotNull(String message, Object o)` – проверява, че обектът не е null

- `assertEquals(String message, expected, actual)` – проверява за равенство на два обекта
- масивите се сравняват по референции, не по съдържание
- assertArrayEquals(String message, expected, actual) - проверява за равенство на два масива по дължина и съдържание

- `assertEquals(String message, expected, actual, delta)` – проверява за равенство на числа с плаваща точка
- делтата (delta) определя точността на сравнението

- `assertSame(String message, expected, actual)` – проверява, че двете референции съвпадат
- `assertNotSame(String message, expected, actual)` – проверява, че двете референции са различни

- Във всички assert методи, String параметърът (message) е опционален
- Добра практика да го подавате винаги и съобщението да е конкретно, подробно и смислено
- Важно е в случаите, в които различни програмисти пишат продуктивния код и тестовете

РЕД НА ИЗПЪЛНЕНИЕ

- JUnit предполага, че всички тестови методи могат да се изпълняват в произволен ред
- Добре написаните тестове не трябва да разчитат на конкретен ред на изпълнение
- Т.е. тестовете не трябва да зависят от други тестове

РЕД НА ИЗПЪЛНЕНИЕ

От JUnit 4.11 натам, може явно с анотация на класа да определите реда на изпълнение да е лексикографския ред на имената на тестовите методи.

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
```

Ако имате няколко тестови класа, може да ги обедините в test suite. Изпълнението на test suite ще изпълни всички тестови класове в него в указания ред.

ПРИМЕР ЗА JUNIT TEST SUITE

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({
    MyClassTest.class,
    MySecondClassTest.class
})
public class AllTests {

}
```

JUNIT АНОТАЦИИ

- `@Test`
  - обозначава метод като тестов метод
- `@Test(expected = Exception.class)`
  - фейлва, ако методът не хвърли указаното изключение
- `@Test(timeout = 100)`
  - фейлва, ако изпълнението на метода продължи повече от 100 милисекунди

JUNIT **АНОТАЦИИ** (2)

- `@Before public void method()`
  - този метод се изпълнява преди всеки тест
  - използва се за подготовка на тестовата среда
- `@After public void method()`
  - този метод се изпълнява след всеки тест
  - използва се да зачисти тестовата среда

КЪДЕ "ЖИВЕЯТ" UNIT ТЕСТОВЕТЕ?

- Обикновено unit тестовете се разполагат в отделен проект или в отделна source директория, за да са отделени от продуктивния код
- Няма единен стандарт - зависи с какви други tools (например за build) искате интеграция

- Един вариант (maven)

```
fancy-project
└─ src/main/java
   └─ (...)
└─ src/test/java
   └─ (...)
```

- Друг вариант

```
fancy-project
└─ src
   └─ (...)
└─ test
   └─ (...)
```

КАК СЕ ИЗПЪЛНЯВАТ?

- През IDE
- През конзола
- През build системи (maven, gradle)
- През Continuous Integration (CI) системи (Jenkins, Travis)
- Засега ще се ограничим да ги изпълняваме през IDE-то

CODE COVERAGE PLUG-INS

- EclEmma for Eclipse
- Code coverage runner for IntelliJ

BEST PRACTICES

- Не тествайте тривиален код като getters/setters
- Тествайте private методи само косвено
- Стремете се към 70-80% code coverage
- Пишете кратки, ясни и бързи unit тестове
- Избягвайте `try-catch`
- Не ползвайте `Thead.sleep`

JUNIT 4 VS. JUNIT 5

	JUnit 4	JUnit 5
Пакет	`org.junit`	`org.junit.jupiter.api`
Анотиране на тест	`@Test`	`@Test`

ИНИЦИАЛИЗАЦИЯ

	JUnit 4	JUnit 5
Обща	`@BeforeClass`	`@BeforeAll`
Преди всеки тест	`@Before`	`@BeforeEach`

ЗАКЛЮЧИТЕЛНИ ОПЕРАЦИИ

	JUnit 4	JUnit 5
След всеки тест	`@After`	`@AfterEach`
Общи	`@AfterClass`	`@AfterAll`

ВРЕМЕННО ИЗКЛЮЧВАНЕ НА ТЕСТОВ МЕТОД ИЛИ КЛАС

JUnit 4	JUnit 5
`@Ignore`	`@Disabled`

STUBBING AND MOCKING

UNIT ТЕСТВАНЕ НА КЛАСОВЕ СЪС ЗАВИСИМОСТИ

- Често нашите класове, за да свършат своята работа, използват други класове
- Познато ви е като композиция
- Това е съвсем очаквано и нормално :)
- Как unit тестваме такива класове?
- Да разгледаме един такъв клас

```
public class UserService {

    private UserRepository repository;
    private MailService mailService;

    public User register(String email, String password) {
        if (repository.exists(email)) {
            throw new UserAlreadyExistsException();
        }

        User user = new User(email, password);
        repository.save(user);
        mailService.sendWelcomeMail(email);
        return user;
    }
```

- Нека UserRepository е интерфейс, чиято задача е да съхранява User-и (in-memory, file system, database, etc.)
- Нека MailService също е интерфейс, чиято задача е да праща мейли

TEST CASES FOR REGISTER()

- Методът register() има 2 exit point-а - следователно имаме 2 сценария за покриване
- [TC1] register() хвърля подходящо изключение, когато мейл адресът вече съществува в хранилището
- [TC2] register() запазва подходящия user в хранилището и изпраща welcome мейл, когато мейл адресът не съществува в хранилището
- Как unit тестваме UserService класа?

- При unit тестване се интересуваме от функционалната коректност само на класа, който се тества
- Трябват ни инструменти, чрез които да "изолираме" композираните класове
- Композираните класове могат да бъдат трудни за инстанциране
- Например UserRepository изисква connectivity към база от данни

- Доброто unit тестване се базира на изолация
- Изолацията се постига чрез stub/mock обекти

STUBBING

- Stub - клас, който отговаря на дадени извиквания на методи с предварително зададени отговори
- В unit тестването ни служат за справяне с проблема с композираните класове

STUBBING

- Обикновено имплементират по минимален начин даден интерфейс и се подават на класа, който се тества
- Извън unit тестването могат да бъдат използвани и като заместител на код, който още не е разработен

PositiveUserRepositoryStubImpl

```
public class PositiveUserRepositoryStubImpl
                implements UserRepository {

    @Override
    public boolean exists(String email) {
        return true;
    }

    @Override
    public void save(User user) {
        // Do nothing
    }
}
```

InMemoryUserRepositoryStubImpl

```
public class InMemoryUserRepositoryStubImpl
                implements UserRepository {

    private Map<String, User> users = new HashMap<>();

    @Override
    public boolean exists(String email) {
        return users.containsKey(email);
    }

    @Override
    public void save(User user) {
        users.put(user.getEmail(), user);
    }
}
```

THE STUB WAY

```
@Test(expected = UserAlreadyExistsException.class)
public void testRegisterThrowsAppropriateException() {
    UserService service =
        new UserService(new PositiveUserRepositoryStubImpl());

    service.register("test@test.com", "weak");
}
```

TEST LIFECYCLE WITH STUB

1. Setup data - подготвяме обекта, който ще се тества, както и stub събратята му
2. Exercise - извикваме метода
3. Verify state - използваме assertions, за да проверим състоянието на обекта
4. Teardown - освобождаваме използваните ресурси

ХАРАКТЕРИСТИКИ НА STUB-ОВЕТЕ

- Могат да съдържат логика, която не е тривиална (напр. InMemoryUserRepositoryStubImpl) (+)
- Броят на stub-овете расте експоненциално (-)
- Не може да проверим дали даден метод на stub-а е извикан определен брой пъти (-)

MOCKING

- Mock - конфигуриран обект с предварително зададени отговори на дадени извиквания на методи
- Динамични wrapper-и за композираните класове
- По подобие на stub-овете ни служат за справяне с проблема с композираните класове

THE MOCK WAY

```
@Test(expected = UserAlreadyExistsException.class)
public void testRegisterThrowsAppropriateException() {
    UserRepository mock = mock(UserRepository.class);
    when(mock.exists("test@test.com")).thenReturn(true);

    UserService service = new UserService(mock);
    service.register("test@test.com", "weak");
}
```

TEST LIFECYCLE WITH MOCK

1. Setup data - подготвяме обекта, който ще се тества, както и mock събратята му
2. Setup expectations - задаваме желаните отговори
3. Exercise - извикваме метода
4. Verify expectations - уверяваме се, че правилният метод на mock-а се е извикал
5. Verify state - използваме assertions, за да проверим състоянието на обекта
6. Teardown - освобождаваме използваните ресурси

TO MOCK OR NOT TO MOCK?

```
public class Cinema {
    private Map<String, Projection> projections;

    public Cinema(Map<String, Projection> projections) {
        this.projections = projections;
    }

    public boolean buyTicket(String projection, int amount) {
        if (!projections.containsKey(projection)) {
            return false;
        }
        // [...]
        return true;
    }
}
```

DO NOT MOCK

```
@Test
public void testBuyTicket() {
    Map<String, Projection> projections =
            Map.of("foo", new Projection("foo"));
    Cinema cinema = new Cinema(projections);

    boolean actual = cinema.buyTicket("bar", 3);
    assertFalse(actual);
}
```

TO MOCK OR NOT TO MOCK?

```
public class Cinema {
    private ProjectionService service;

    public Cinema(ProjectionService service) {
        this.service = service;
    }

    public boolean buyTicket(String projection, int amount) {
        if (!service.contains(projection)) {
            return false;
        }
        // [...]
        return true;
    }
}
```

MOCK

```
@Test
public void testBuyTicket() {
    ProjectionService mock = mock(ProjectionService.class);
    when(mock.contains("bar")).thenReturn(false);

    Cinema cinema = new Cinema(mock);

    boolean actual = cinema.buyTicket("bar", 3);
    assertFalse(actual);
}
```

ИЗПОЛЗВАЙТЕ MOCK-ВЕ, КОГАТО:

- Композираният клас се обръща към външен ресурс (REST API, database, file system, etc.)
- Логиката в композирания клас не е тривиална
- Не може да настроите test environment-а по тривиален начин

НЕ ИЗПОЛЗВАЙТЕ MOCK-ВЕ, КОГАТО:

- Композираният клас представлява value object, който може да подадете отвън
- Може тривиално да настроите test environment-а

MOCKING LIBRARIES

- Mockito
- EasyMock
- PowerMock*
- * Putting it in the hands of junior developers may cause more harm than good.

MOCKITO

- Ще разглеждаме mockito (2.23.0) като mocking library
- Възниква като разширение на функционалността на EasyMock
- Една от 10-те най-популярни Java библиотеки като цяло
- Open-source

SETUP

- Mockito е външна библиотека
- Може да я изтеглите от тук
- Изтеглете mockito-core jar-а и 3-те му compile dependency-та
- Ако ползвате IDE, добавете въпросните jar-ки в class path-а на проека ви
- Алтернативно, ако сте запознати с maven/gradle, ползвайте тях :)

SETUP (2)

```
fancy-project
  └─ src/
    └─ (...)
  └─ test/
    └─ (...)
  └─ lib/
    └─ byte-buddy-1.9.0.jar
    └─ byte-buddy-agent-1.9.0.jar
    └─ mockito-core-2.23.0.jar
    └─ objenesis-2.6.jar
```

mock() and verify()

```
import static org.mockito.Mockito.*;

List mockedList = mock(List.class);

mockedList.add("one");
mockedList.clear();
mockedList.get(0);

verify(mockedList).add("one");
verify(mockedList, atLeastOnce()).clear();
verify(mockedList, never()).add("two");
```

when()

```
LinkedList mockedList = mock(LinkedList.class);

when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new RuntimeException());

mockedList.get(0);
mockedList.get(1);
```

Argument matchers

```
when(mockedList.get(anyInt())).thenReturn("element");

mockedList.get(999);
```

@Mock annotation

```
@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {
    @Mock
    private UserRepository repositoryMock;

    @Test(expected = UserAlreadyExistsException.class)
    public void testRegisterThrowsAppropriateException() {
        when(repositoryMock.exists("test@test.com"))
                .thenReturn(true);

        UserService service =
                new UserService(repositoryMock);

        service.register("test@test.com", "weak");
    }
```

MOCKITO LIMITATIONS

- Не може да mock-ва static методи (static is evil)
- Не може да mock-ва конструктори
- Не може да mock-ва final класове и методи
- Не може да mock-ва методите equals() и hashCode()

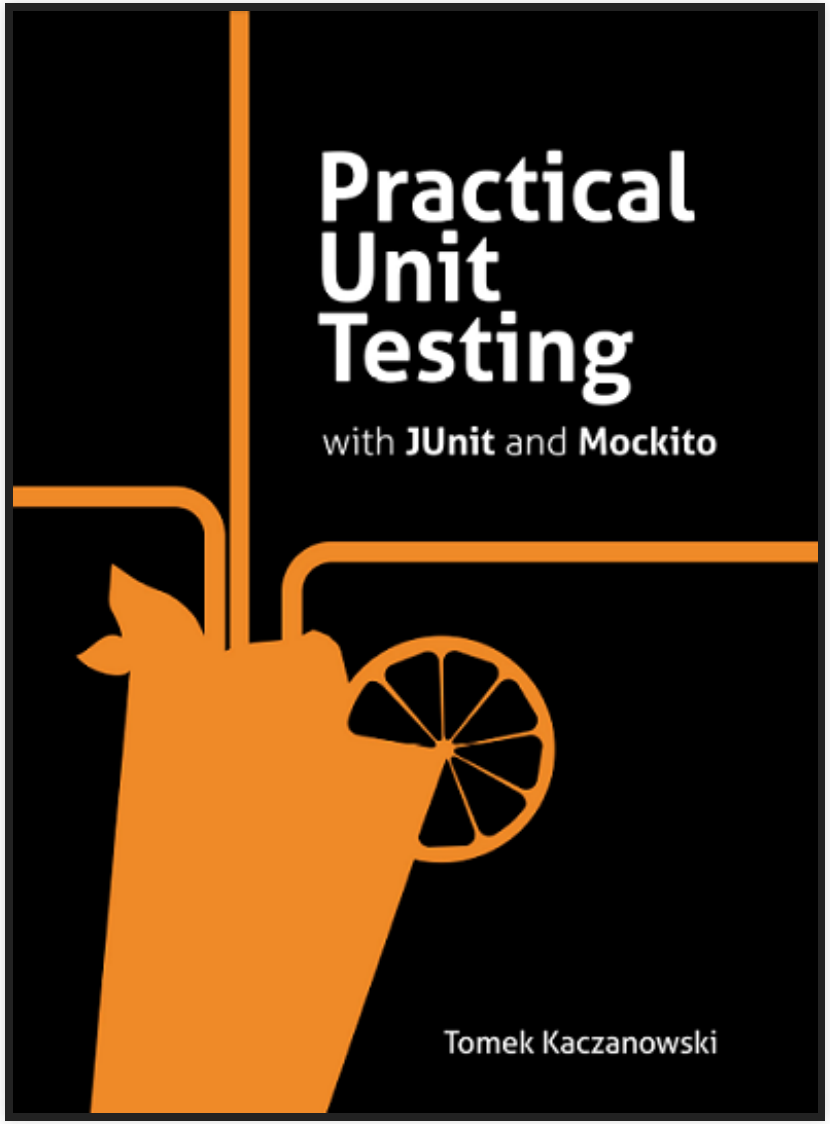
BEST PRACTICES

- Не правете йерархии от тестови класове
- Не mock-вайте типове, които не притежавате
- Mock-вайте само толкова колкото ви трябва за конкретния тест
- Не mock-вайте value обекти
- Keep it short and simple (KISS)
- Redesign when you cannot test it

ПОЛЕЗНИ ЧЕТИВА

- JUnit javadoc
- Unit Testing with JUnit
- Mocks aren't stubs by Martin Fowler
- Writing good tests by Mockito team

ПОЛЕЗНИ ЧЕТИВА



ВЪПРОСИ