Test Driven Development (TDD)

Unit Testing with JUnit 4, JUnit 5 and TestNG

Unit Testing with JUnit 4 and JUnit 5

The solution to the problem of costly tests, however, is not to stop testing but instead to get better at it. Getting good value from tests requires clarity of intention and knowing what, when, and how to test.

— Sandi Metz, Practical Object Oriented Design in Ruby, page 192

Источник : http://c2.com/cgi/wiki?CodeUnitTestFirst

Съдържание - I

1. Software testing
2. Types of testing
3. Levels of testing
4. Specific goals during testing
5. Test Driven Development (TDD)
6. Agile Testing - TDD
7. JUnit 4
8. Basic annotations in JUnit 4
9. Validity conditions (Assertions)
10. Example test class and test suite
11. Parametric tests with JUnit 4

Съдържание - II

12. JUnit 5
13. Basic annotations in JUnit 5
14. Validity conditions (Assertions)
15. Example test class and test suite
16. Parametric tests with JUnit 5

Writing High Quality Code

According to Sandi Metz these three skills required to build high quality, maintainable code:

- Understanding Object-Oriented Design
- Refactoring
- Writing high-value, efficient tests

- Test is executable documentation for the code.

Источник : http://c2.com/cgi/wiki?CodeUnitTestFirst

Тестване на софтуера

Софтуерното тестване е процес на изследване на софтуера, с цел получаване на информация за качеството на продукта или услугата, която се изпитва. Софтуерното тестване може да осигури обективен, независим поглед, който да даде възможност на клиента да разбере рисковете при реализацията на софтуера. Техниките за тестване включват (но не са ограничени до) изпълнение на програмата с намерение да се открият софтуерни бъгове (грешки или други дефекти). Процесът на софтуерно тестване е неразделна част от софтуерното инженерство и осигуряване на качеството на софтуера. [Wikipedia]

Източник : Wikipedia - http://en.wikipedia.org/wiki/Software_testing

Видове тестване

- Static & dynamic testing
- White-Box testing – тества вътрешната структура и работа на софтуера (API testing, Code coverage, Fault injection – Stress testing, Mutation testing)
- Black-box testing – тества функционалността без да се интересува от вътрешната реализация

- Grey-box testing – използва познания за структурите от данни и алгоритмите при разработката на тестове
- Visual testing – записват се всички действия на тестващия с цел лесно да се възпроизведе проблема
- Functional testing - validates the software system against the functional requirements/specifications. The purpose of Functional tests is to test each function of the software application, by providing appropriate input, verifying the output against the Functional requirements.

Нива на тестване

- Unit testing – компонентно тестване, при което се тества функционалността на специфична секция от кода (обикновено метод – като минимум конструкторите)
- Integration testing – проверява дали интерфейсите между компонентите са реализирани според спецификацията им
- System testing – тества се напълно интегрираната система за да се определи дали реализацията съответства на изискванията
- Acceptance testing – тестване на системата от крайните й потребители

Източник : Wikipedia - http://en.wikipedia.org/wiki/Software_testing

Специфични цели при тестване

- Installation testing
- Compatibility testing
- Smoke and sanity testing
- Regression testing
- Acceptance testing
- Alpha testing
- Beta testing
- Functional vs non-functional testing
- Destructive testing
- Software performance testing
- Usability testing
- Accessibility
- Security testing
- Internationalization and localization
- Development testing

Източник : Wikipedia - http://en.wikipedia.org/wiki/Software_testing

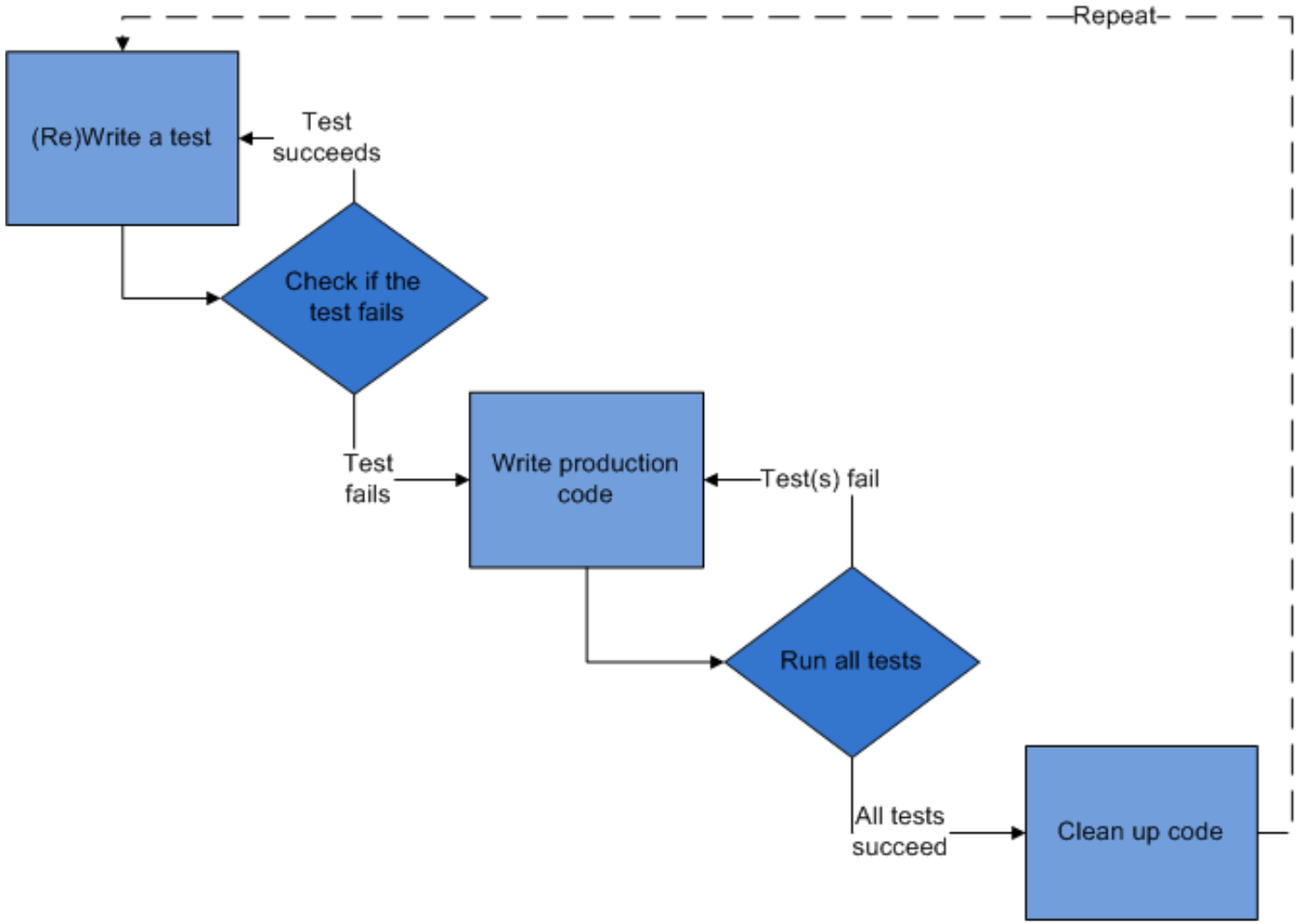
Test Driven Development (TDD) с JUnit 4

- Test-Driven Development (TDD) е техника, при която разработката на софтуер се насочва чрез писане на тестове.
- Първоначално е развита от Kent Beck (в края на 90-те).
- Основната идея е да се повтарят последователно следните пет стъпки:
  1. Пишем автоматичен тест (Unit test) за следващата **малка** част нова функционалност като си представяме че кодът вече съществува;
  2. Пишем празни методи (Stubs), така че кодът да се компилира;
  3. Пускаме теста – той трябва да пропадне, иначе тестът не е добър;
  4. Пишем **минималното** количество функционален код така, че тестът да успее – ако тестът не минава успешно, значи кодът не е добър;
  5. Променяме (Refactor) както стария, така и новия код, за да го структурираме по-добре.

Test Driven Development (TDD)

- **Test-Driven Development (TDD)** is a technique allowing to guide the software development by writing tests.
- Initially developed by Kent Beck (in the end 90s).
- The main idea is to repeat the following 5 steps:
  1. Write automatic test (Unit test) for the next **small** new functionality, imagining that the code implementing it already exists;
  2. Write empty methods (Stubs), to make the code compile;
  3. Run the test – it **should fail**, otherwise the test is not good;
  4. Write **minimal** functional code, so that the **test will pass** – it the test does not pass, then the code is not good;
  5. Change (Refactor) the old as well the new code in order to structure it better.

Sequential Stages in TDD



Източник: Wikipedia (http://en.wikipedia.org/wiki/File:Test-driven_development.PNG), Автор: Excirial, Лиценз: Creative Commons Attribution, ShareAlike 3.0 License / GNU Free Documentation License, Version 1.2

Agile Testing - TDD

A good way to develop new functionality is the following:

1. Consider what you have tot do.
2. Write a UnitTest for the desired functionality, and the least possible code increment to implement it.
3. Run the UnitTest. If the test passes you are ready; go to step 1 or if you have completed everything go home ☺
4. Solve the current problem: may be you have not written correctly the new method. May be the method is not working as expected. Do the necessary corrections. Go to step 3.

Източник : http://c2.com/cgi/wiki?CodeUnitTestFirst

Should We Test Private Methods?

- I short: you shouldn't.
- Instead you can test indirectly their effects on the public methods calling them.
- Unit tests are clients for the object under test, which is not different from other clients of the object. Unit test is you first client in TDD.
- If it is hard to test the object via its public interface, it will be hard to use in the production code.
- It is a code smell and a good example how testing guides the good object-oriented design.

Източник : http://c2.com/cgi/wiki?CodeUnitTestFirst

JUnit 4

- **UnitTest** – tests the specific class and in order to access all its methods is in the same package (if we want to access the private class members we can use рефлексия or we can implement the test as internal class, but most of the time it is better to refactor your code as explained in previous slides). In JUnit 3 it was necessary to extend the class junit.framework.TestCase and every method was named testXXX. In JUnit 4 all these requirements are removed and instead each test method should be annotated with **@Test**.
- **Test Suite** – allows to group test with similar purpose in a suite:

```
@RunWith(Suite.class)

@SuiteClasses({ PriceComparatorTest.class, TransactionTest.class })

public class AllTests { }
```

JUnit 4

- **UnitTest** – tests the specific class and in order to access all its methods is in the same package (if we want to access the private class members we can use рефлексия or we can implement the test as internal class, but most of the time it is better to refactor your code as explained in previous slides). In JUnit 3 it was necessary to extend the class junit.framework.TestCase and every method was named testXXX. In JUnit 4 all these requirements are removed and instead each test method should be annotated with **@Test**.
- **Test Suite** – allows to group test with similar purpose in a suite:

```
@RunWith(Suite.class)

@SuiteClasses({ PriceComparatorTest.class, TransactionTest.class })

public class AllTests { }
```

Основни анотации в JUnit 4

- **@Test** – identifies a test method in JUnit 4
- **@Test (expected = ExtendingException.class)**
- **@Test (timeout=200)** – maximal allowed time in milliseconds
- **@Before** – executed before each test
- **@After** – executed sfter each test
- **@BeforeClass** – executed once before all tests – the method should be static
- **@AfterClass** – executed once after finishing all tests – the method should be static
- **@Ignore** – ignores the test method
- **@RunWith** – specifies the Runner class which executes tests
- **@SuiteClasses** – annotates a test suit class grouping tests with similar purpose, used with **@RunWith(Suite.class)**

Test Class Pattern for JUnit 4

```
import org.junit.Ignore;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

/** Tests for {@link Foo}. */
@RunWith(JUnit4.class)
public class FooTest {
    @Test
    public void thisAlwaysPasses() {}

    @Test
    @Ignore
    public void thisIsIgnored() {}
}
```

Источник : https://github.com/junit-team/junit/wiki/Getting-started

JUnit 4 Unit Test Example (1)

```
public class GcdTest {
   @Test
   public void testIsPositiveInteger() {
      String[] testData = {"346", "-23", "29a34", "17.5"};
      boolean[] resultData = {true, false, false, false };
      for(int i = 0; i < testData.length; i++){
         assertEquals(resultData[i],
            Gcd.isPositiveInteger(testData[i]));
      }
   } ...
```

JUnit 4 Unit Test Example (2)

```
@Test
public void testGreatestCommonDenominator() {
 int [][] testData= {{48, 72, 24}, {17, 351, 1},
     {81, 63, 9}};
 for(int[] data: testData){
   int result = Gcd.greatestCommonDenominator(data[0],
      data[1]);
   assertEquals(data[2], result);
   result = Gcd.greatestCommonDenominator(data[1],data[0]);
   assertEquals(data[2], result);
 }
}
```

Validity constraints (Assertions)

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

assertArrayEquals("values not same", expected, actual)

assertFalse("failure - should be false", expected)

assertTrue("failure - should be true", expected)

assertNotNull("should not be null", myObject)

assertNotSame("should not be same Object", myObject, other)

assertNull("should be null", null)

assertSame("should be same", aNumber, aNumber)

assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))))  ...
```

JUnit 4 Test Lifecycle – Order of Execution

1. @BeforeClass setupClass()
2. @Before setup()
3. @Test test1()
4. @After cleanup()
5. @Before setup()
6. @Test test2()
7. @After cleanup()
8. @AfterClass cleanupClass()

Example - JUnit 4 (1)

```
public class TransactionTest {

private static InputStream in;

@BeforeClass

public static void setUpBeforeClass() throws Exception {

  String data="Goole Inc.\nJohn Smith\nGOGL\n42.78\n120\n";

  in = new ByteArrayInputStream(data.getBytes()));

}

@AfterClass

public static void tearDownAfterClass() throws Exception {

  in.close();

}
```

Example - JUnit 4 (2)

```
@Test
public void testTransactionFullConstructor() {
  Transaction t = new Transaction("Google Inc.", "John Smith", "GOGL", 27, 19.439834456455544);
  assertNotNull(t);
  assertTrue("Transaction ID not correct", t.getId() > 0);
  assertTrue("'timestamp' not correct",
     t.getTimestamp().getTime() <= new Date().getTime());
  assertEquals("Google Inc.", t.getSeller());
  assertEquals("John Smith", t.getBuyer());
  assertEquals("GOGL", t.getSymbol());
  assertEquals(27, t.getQuantity());
  assertEquals(19.4398, t.getPrice(),1E-4);
}
```

Example - JUnit 4 (3)

```
  @Test
  public void testInput() {
    Transaction t = new Transaction();
    assertNotNull(t);
    t.input(in);
    assertTrue("Transaction ID not correct", t.getId()>0);
    assertEquals("Google Inc.", t.getSeller());
    assertEquals("John Smith", t.getBuyer());
    assertEquals("GOGL", t.getSymbol());
    assertEquals(120, t.getQuantity());
    assertEquals(42.78, t.getPrice(),1E-4);
  }
  ...
}
```

Test Suite Example – JUnit 4

```
import org.junit.runner.RunWith;

import org.junit.runners.Suite;

import org.junit.runners.Suite.SuiteClasses;



@RunWith(Suite.class)

@SuiteClasses({ PriceComparatorTest.class,
  TransactionTest.class })

public class AllTests {

}
```

Parameterized Tests with JUnit 4 (1)

```
@RunWith(value = Parameterized.class)
public class GcdTestWithParameters {
    private int numberA;
    private int numberB;
    private int expected;
    //pass parameters using test constructor
    public GcdTestWithParameters(
                    int numberA, int numberB, int expected) {
        this.numberA = numberA;
        this.numberB = numberB;
        this.expected = expected;
    }
```

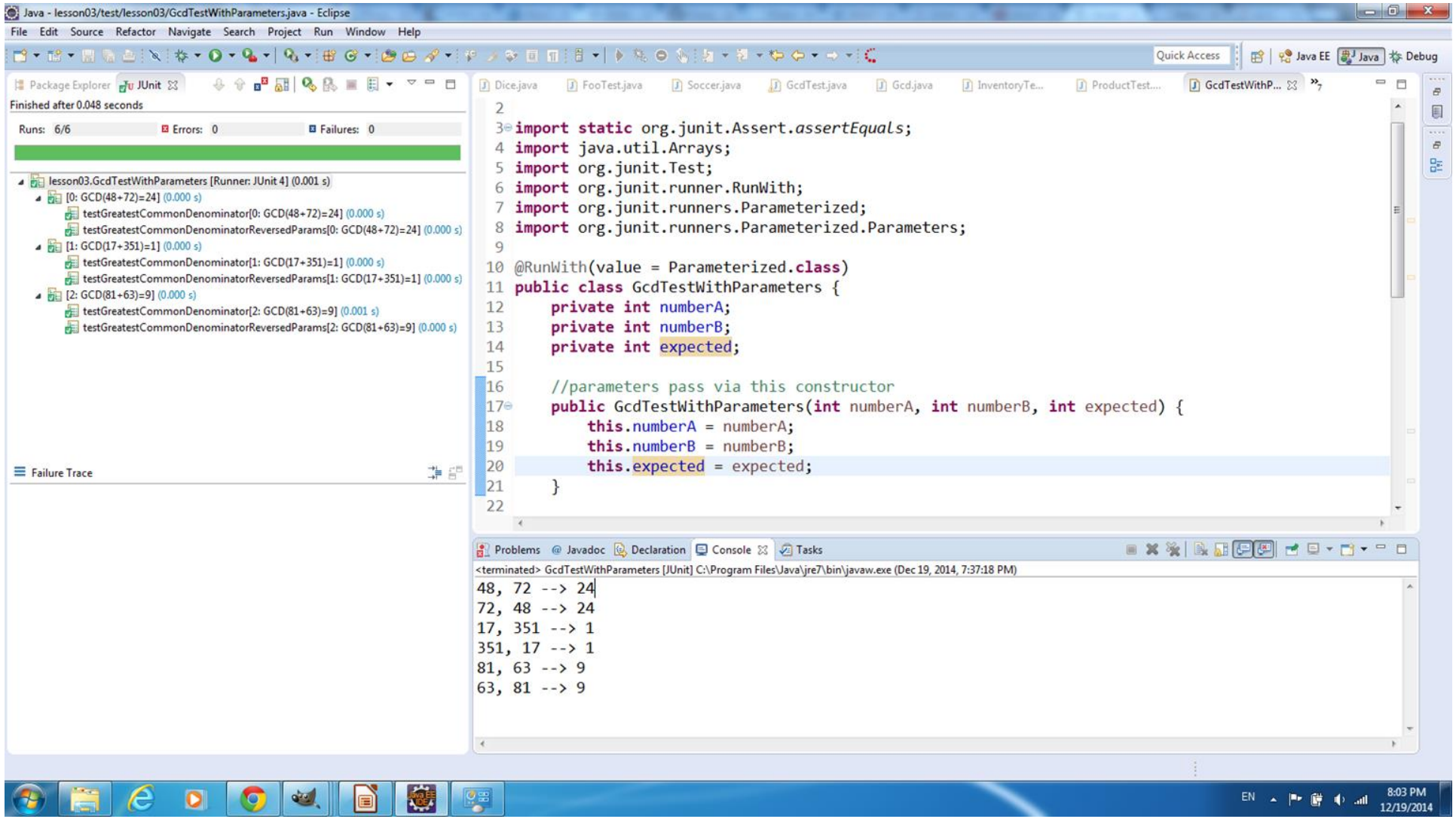
Parameterized Tests with JUnit 4 (2)

```
//Declared test parameters
@Parameters(name = "{index}: GCD({0}+{1})={2}")
public static Iterable<Object[]> data1() {
    return Arrays.asList(new Object[][] {
        {48, 72, 24},
        {17, 351, 1},
        {81, 63, 9}
    });
}
```

Parameterized Tests with JUnit 4 (3)

```
    @Test
    public void testGreatestCommonDenominator() {
        int result = Gcd.greatestCommonDenominator(numberA, numberB);
        assertEquals(expected, result);
    }
    @Test
    public void testGreatestCommonDenominatorReversedParams() {
        int result = Gcd.greatestCommonDenominator(numberB, numberA);
        assertEquals(expected, result);
    }
}
```

Parameterized Tests with JUnit 4 (4)



Core Principles Guiding Evolution of JUnit

- **Prefer extension points over features** - it's better to enable new functionality by creating or augmenting an extension point rather than adding the functionality as a core feature
- **Complementarily, an extension point should be good at one thing** - let an extension point be good at what it's good at, and don't be afraid to introduce new extension points to handle weak points in existing ones.
- **It should be hard to write tests that behave differently based on how they are run**
- **Tests should be easy to understand** - It should be possible to understand how JUnit will treat a class based on reading the test class (and base class) and looking at the annotations.
- **Minimize dependencies (especially third-party)** – example Hamcrest assertThat

JUnit 5

JUnit 5 = *JUnit Platform + JUnit Jupiter + JUnit Vintage*

- **JUnit Platform** - IDEs, build tools or plugins need to extend platform APIs to launch JUnit tests,. Includes TestEngine API for developing testing frameworks that run on the platform. Provides a Console Launcher to launch the platform from the command line and build plugins for Gradle and Maven.
- **JUnit Jupiter** - new programming and extension models for writing tests. It has all new JUnit annotations and TestEngine implementation to run tests written with these annotations.
- **JUnit Vintage** - supports running JUnit 3 and JUnit 4 written tests on the JUnit 5 platform for backward compatibility.

JUnit 5 New Annotations – II

@Test - method is a test method. Unlike JUnit 4's @Test annotation, this annotation does not declare any attributes, since test extensions in JUnit Jupiter operate based on their own dedicated annotations. Such methods are inherited unless they are overridden.

@DisplayName – defines custom display name for a test class or method

@Nested – denotes that the annotated class is a nested, non-static test class. @BeforeAll and @AfterAll methods cannot be used directly in a @Nested test class unless the "per-class" test instance lifecycle is used.

@Tag – declares tags for filtering tests

@Disable – it is used to disable a test class or method (previously @Ignore)

JUnit 5 New Annotations – I

@ExtendWith – it is used to register custom extensions declaratively

@BeforeEach – denotes that the annotated method will be executed before each test method (previously @Before)

@AfterEach – denotes that the annotated method will be executed after each test method (previously @After)

@BeforeAll – denotes that the annotated method will be executed before all test methods in the current class (previously @BeforeClass)

@AfterAll – denotes that the annotated method will be executed after all test methods in the current class (previously @AfterClass)

JUnit 5 New Annotations – III

@ParameterizedTest - denotes that a method is a parameterized test

@RepeatedTest - method is a test template for a repeated test

@TestFactory – denotes a method that is a test factory for dynamic tests

@TestTemplate - denotes that a method is a template for test cases designed to be invoked multiple times depending on the number of invocation contexts returned by the registered providers

@TestMethodOrder - used to configure the test method execution order for the annotated test class; similar to JUnit 4's @FixMethodOrder

@TestInstance - used to configure the test instance lifecycle for the annotated test class.

JUnit 5 New Annotations – IV

@DisplayNameGeneration - declares a custom display name generator for the test class.

@Timeout - used to fail a test, test factory, test template, or lifecycle method if its execution exceeds a given duration

@RegisterExtension - used to register extensions programmatically via fields

@TempDir - used to supply a temporary directory via field injection or parameter injection in a lifecycle method or test method; located in the org.junit.jupiter.api.io package

JUnit 5 Maven Dependencies

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.15.0</version>
    <scope>test</scope>
</dependency>
```

JUnit 5 Lifecycle Methods - I

```
@BeforeAll
static void setup() {
    log.info("@BeforeAll - executes once before all test methods in this class");
}

@AfterAll
static void cleanup() {
    log.info("@AfterAll - executes once before all test methods in this class");
}
```

JUnit 5 Lifecycle Methods - II

```
@BeforeEach
void init() {
    log.info("@BeforeEach - executes before each test method in this class");
    repo = new ProductRepositoryMemoryImpl(new LongKeyGenerator());
    SAMPLE_PRODUCTS.forEach(p -> {
        try {
            repo.create(p);
        } catch (EntityAlreadyExistsException e) {
            e.printStackTrace();
        }
    });
}

@AfterEach
void tearDown() {
    log.info("@AfterEach - executes before each test method in this class");
}
```

JUnit 5 Test Methods - I

```
@Test
void findById() throws EntityAlreadyExistsException {
    assertEquals(repo.create(NEW_PRODUCT).getCode(), "CB001");
}
```

```
@Test
@Disabled("Not implemented yet")
void create() {
}
```

JUnit 5 Test Methods – Using AssertJ Soft Assertions

```
@Test
@DisplayName("Find all products")
void findAll() {
    List<Product> result = repo.findAll();
    SoftAssertions softly = new SoftAssertions();
    softly.assertThat(softly.assertThat(result)).isNotNull();
    softly.assertThat(result.size()).isEqualTo(5);
    softly.assertThat(result.get(0).getCode()).isEqualTo("BK001");
    softly.assertAll();
}
```

JUnit 5 Assumptions

```
@Test
void assumptionThat() {
    String someString = "Some string";
    assumingThat(
            someString.equals("Some string"),
            () -> assertEquals(11, someString.length())
    );
}
```

JUnit 5 Testing for Exceptions

```
@Test
void shouldThrowException() {
    Throwable exception = assertThrows(UnsupportedOperationException.class, () -> {
        throw new UnsupportedOperationException("Not supported");
    });
    assertEquals(exception.getMessage(), "Not supported");
}
```

JUnit 5 Parameterized Tests

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
</dependency>
```

```
@ParameterizedTest(name="#{index} - Test with Argument={0}")
@ValueSource(ints = {8,4,2,6,10})
void test_int_arrays(int arg) {
    System.out.println("arg => "+arg);
    assertTrue(arg % 2 == 0);
}
```

JUnit 5 Parameterized Tests - II

```
@ParameterizedTest
@CsvSource({
        "Peter, admin, 1",
        "John, author, 2",
        "Martin, subscriber, 3"
})
void testWith_CsvSource(String name, String role, long id) {
    System.out.println("testWith_CsvSource: name => "+name+"; role =>
"+role+"; id => "+id);
    assertTrue(name.length() >= 0);
    assertTrue(id >=1 && id <=3);
    assertTrue(!role.isEmpty());
}
```

JUnit 5 Parameterized Tests - III

```
@ParameterizedTest
@CsvFileSource(resources = "/users-data.csv", numLinesToSkip = 1)
void testWith_MethodSource(String name, String role, long id) {
    System.out.println("name => "+name+"; role => "+role+"; id => "+id);
    assertTrue(name.length() >= 0);
    assertTrue(id >=1 && id <=3);
    assertTrue(!role.isEmpty());
}
```

JUnit 5 Dynamic Tests using @TestFactory

- **DynamicTest** is a test generated during runtime
- DynamicTests are generated by a factory method annotated with the ***@TestFactory*** annotation
- A ***@TestFactory*** method cannot be static or *private* and must return a *Stream*, *Collection*, *Iterable*, or *Iterator of DynamicTest* instances. Otherwise a *JUnitException* is thrown
- DynamicTests are executed id different way than the standard @Tests and do not support lifecycle callbacks
- DynamicTests differ from the parameterized tests because they support full test lifecycle, while parametrized tests do not
- JUnit 5 prefers extensions over features principle

JUnit 5 Dynamic Tests Example

```
@TestFactory
Collection<DynamicTest> dynamicTestsCollection() {
    return Arrays.asList(
            DynamicTest.dynamicTest("Add test",
                    () -> assertEquals(5, Math.addExact(2, 3))),
            DynamicTest.dynamicTest("Multiply Test",
                    () -> assertEquals(15, Math.multiplyExact(5, 3))));
}

@TestFactory
Stream<DynamicTest> dynamicTestsStream() {
    return IntStream.iterate(0, n -> n + 5).limit(10)
            .mapToObj(n -> DynamicTest.dynamicTest("testMultipleOfFive_" + n,
                    () -> assertTrue(n % 5 == 0)));
}
```

Test Doubles: Mocks and Stubbs

- Stub: a dummy piece of code (fake) that lets the test run, but you don't care what happens to it. Stub can never fail the test. The asserts the test uses are always against the class under test. A stub is an object used to fake a method that has pre-programmed behavior. You may want to use this instead of an existing method in order to avoid unwanted side-effects (e.g. a stub could make a fake fetch call that returns a pre-programmed response without actually making a request to a server).
- Mock: a dummy piece of code (fake), that you VERIFY is called correctly as part of the test. Additionally mocks can be constructed using a convenient library like Mockito or jMock. A mock has pre-programmed behavior as well as pre-programmed expectations. If these expectations are not met then the mock will cause the test to fail (e.g. making fake fetch call that returns pre-programmed response without calling server which would expect e.g. the first argument to be "http://localhost:3008/" otherwise test would fail.)

Test Doubles: Dummies and Spies

- Dummy: an object passed around but never actually used. Usually they are just used to fill parameter lists.

- Spy: a dummy piece of code, that intercepts some calls to a real piece of code, allowing you to verify calls without replacing the entire original object. Spies are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent (also called Partial Mock).

Why Mockito

- StackOverflow community voted Mockito the best mocking framework for java.
- Top 10 Java library across all libraries
- In late 2013 there was an analysis made of 30.000 GitHub projects.
- Dan North, the originator of Behavior-Driven Development wrote this back in 2008: "We decided during the main conference that we should use JUnit 4 and Mockito because we think they are the future of TDD and mocking in Java"

Mockito Features

- Mocks concrete classes as well as interfaces
- Little annotation syntax sugar - @Mock
- Verification errors are clean - click on stack trace to see failed verification in test; click on exception's cause to navigate to actual interaction in code. Stack trace is always clean.
- Allows flexible verification in order (e.g: verify in order what you want, not every single interaction)
- Supports exact-number-of-times and at-least-once verification
- Flexible verification or stubbing using argument matchers (anyObject(), anyString() or refEq() for reflection-based equality matching)
- Allows creating custom argument matchers or using existing hamcrest matchers

Adding Mockito to Maven Project

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.1.0</version>
    <scope>test</scope>
</dependency>


<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>4.0.0</version>
    <scope>test</scope>
</dependency>
```

Verify Interactions

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

// mock creation
List mockedList = mock(List.class);

// using mock object - it does not throw any "unexpected interaction" exception
mockedList.add("one");
mockedList.clear();

// selective, explicit, highly readable verification
verify(mockedList).add("one");
verify(mockedList).clear();
```

Stub Method Calls

```
// you can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

// stubbing appears before the actual execution
when(mockedList.get(0)).thenReturn("first");

// the following prints "first"
System.out.println(mockedList.get(0));

// the following prints "null" because get(999) was not stubbed
System.out.println(mockedList.get(999));
```

Real Example

```
// setup
var updated = copyUser(NEW_USER);
updated.setId(SAMPLE_ID);
updated.setRole(role);
when(mockUserRepo.update(any(User.class))).thenReturn(updated);
when(mockUserRepo.findById(any(Long.class))).thenReturn(Optional.of(updated));

// call
updated.setRole(role);
var actual = userService.updateUser(updated);

// assert
assertEquals(role, actual.getRole());
assertEquals(SAMPLE_ID, actual.getId());

// verify repo method called
verify(mockUserRepo).update(any(User.class));
```

Using Spies

```
public class Test{
        //Instance for spying is created by calling constructor explicitly:
        @Spy
        Foo spyOnFoo = new Foo("argument");
        //Instance for spying is created by mockito via reflection (only default constructors supported):
        @Spy Bar spyOnBar;
        private AutoCloseable closeable;
        @Before
        public void init() {
            closeable = MockitoAnnotations.openMocks(this);
        }
        @After
        public void release() throws Exception {
            closeable.close();
        }
        ...
}
```

Resources

- Software testing в Wikipedia - http://en.wikipedia.org/wiki/Software_testing
- Test-driven development в Wikipedia - http://en.wikipedia.org/wiki/Test-driven_development
- Test Driven Development wiki - http://c2.com/cgi/wiki?TestDrivenDevelopment
- Junit 4 - https://github.com/junit-team/junit/wiki
- JUnit 4 Tutorial - http://www.vogella.com/articles/JUnit/article.html
- JUnit 5 User Guide - https://junit.org/junit5/docs/current/user-guide/
- JUnit 5 Dynamic Tests – https://www.baeldung.com/junit5-dynamic-tests
- JUnit 5 Tutorial - https://javabydeveloper.com/junit-5-tutorial/

Unit Testing with TestNG

What is TestNG?

- TestNG is a testing framework inspired from JUnit and NUnit, but introducing some new functionalities that make it more powerful and easier to use
- Allows creating custom argument matchers or using existing Hamcrest / AssertJ matchers
- TestNG is designed to cover all categories of tests: unit, integration, end-to-end, functional, etc...

```
public class SimpleTest {
    @BeforeClass
    public void setUp() {
        // code that will be invoked when this test
is instantiated
    }
    @Test(groups = { "fast" })
    public void aFastTest() {
        System.out.println("Fast test");
    }
    @Test(groups = { "slow" })
    public void aSlowTest() {
        System.out.println("Slow test");
    }

}
```

TestNG Features

- Annotations.
- Run your tests in arbitrarily big thread pools with various policies available (all methods in their own thread, one thread per test class, etc...).
- Test that your code is multithread safe.
- Flexible test configuration.
- Support for data-driven testing (with @DataProvider).
- Support for parameters.
- Powerful execution model (no more TestSuite).
- Supported by a variety of tools and plug-ins (Eclipse, IDEA, Maven, etc...).
- Embeds BeanShell for further flexibility.
- Default JDK functions for runtime and logging (no dependencies).
- Dependent methods for application server testing.

Maven Dependency

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.1.0</version>
    <scope>test</scope>
</dependency>
```

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD