

Software Testing

Vasil Vasilev,
Kiril Gavrilov
October, 2014

Public

Agenda

Software Testing Overview

JUnit framework

TDD Demo



Software Testing Overview

What is unit testing and why we need it

Basics of Software Testing

Overview – WHY

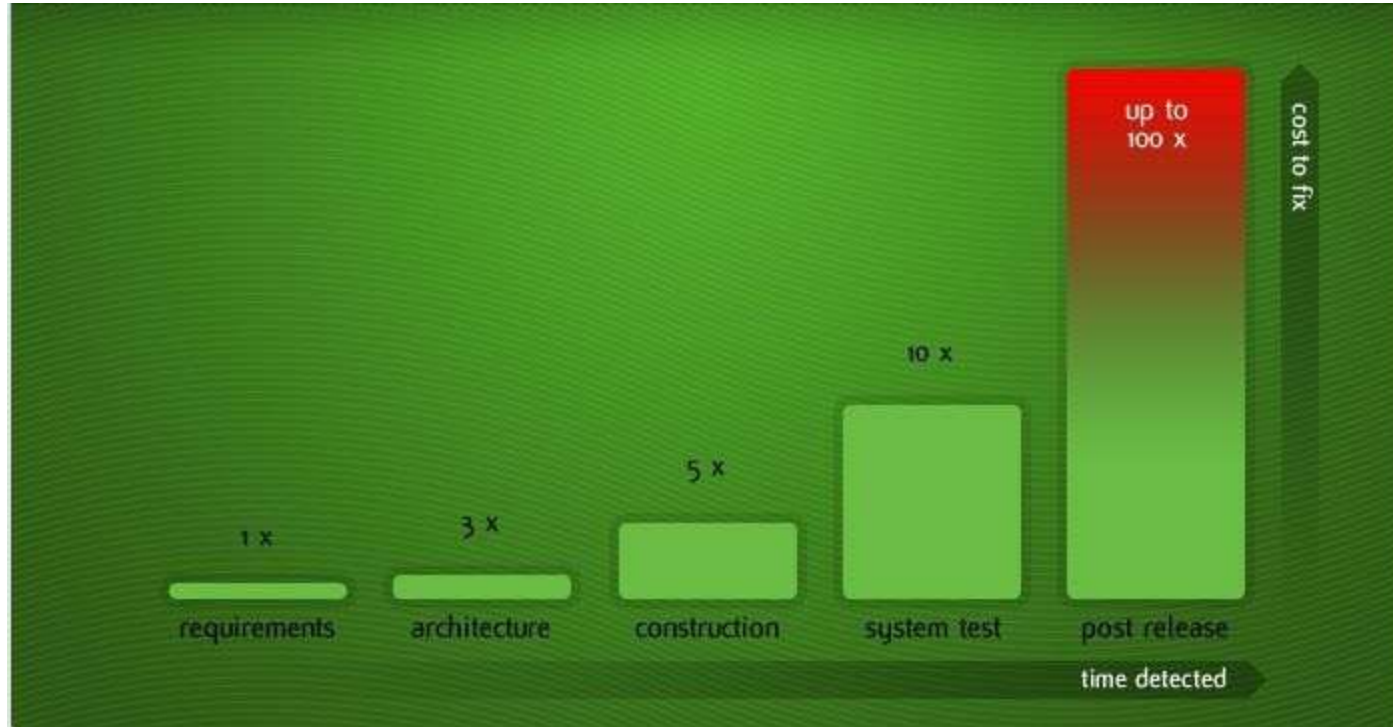
Ariana 5 - trying to stuff a 64-bit number into a 16-bit space



Source: https://www.youtube.com/watch?v=gp_D8r-2hww

Overview – WHEN (1)

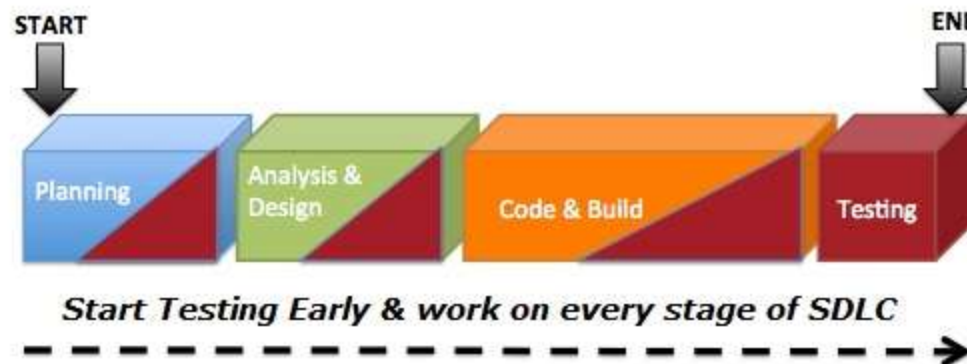
As soon as possible 😊



Overview – WHEN (2)

Software project phases are:

- **Planning** – plan the acceptance testing (product definition)
- **Analysis & Design** – design tests for all use-cases (architect)
- **Code & Build** – write unit tests for all software units (developer)
- **Testing** – integration testing on top of the already created unit test (quality assurance, they should be involved from planning)



Overview – Unit testing

What is unit test:

- A *unit test* is a piece of code written by a developer that executes a specific functionality in the code to be tested.
- A unit test targets a small unit of code, e.g., a method or a class, (local tests).
- Unit tests ensure that code works as intended. They are also very helpful to ensure that the code still works as intended in case you need to modify code for fixing a bug or extending functionality, so they are used as a safety net.

Software Testing Terms (1)

Some important definitions:

- *Productive code* (a.k.a. *code under test*) – it is the code that implements the user requirements and fulfills the customer's use cases.
- The percentage of productive code which is tested by unit tests is typically called *test coverage*. Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests.
- *Test Driven Development* (TDD) is a methodology in which the test code is written upfront the productive code in a way that once the written test is satisfied, the use-case is done.

Software Testing Terms (2)

More important definitions:

- The *test fixture* is a fixed state of the software under test used as a baseline for running tests
- An *integration test* has the target to test the behavior of a component or the integration between a set of components.
- The term *functional test* is sometimes used as synonym for integration test
- *Performance tests* are used to benchmark software components in a repeatable way



JUnit framework

Test

Assertions

TestCase

TestSuite

Execution order

Basic annotations used in JUnit

JUnit

What it is

Why we (developers) need test frameworks?

- Test implementation and test execution goods out of the box
- Common testing knowledge – easy to develop and support

JUnit is the standard for testing framework in Java world

- Typically a JUnit test is a `@Test` annotated method contained in a class which is only used for testing. This is called a **TestCase** class.
- In widespread use is to use the name of the class under test and to add the "Test" suffix to the test class
- For the test method names it is recommend to use the word "should" in the test method name, as for example "ordersShouldBeCreated" or "menuShouldGetActive"

JUnit

Test example

```
@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));
}
```

JUnit

Assert statements (1)

JUnit provides static methods in the Assert class to test for certain conditions:

- `fail(String)` – Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The String parameter is optional.
- `assertTrue([message], boolean condition)` – Checks that the boolean condition is true.
- `assertFalse([message], boolean condition)` – Checks that the boolean condition is false.

JUnit

Assert statements (2)

- `assertEquals([String message], expected, actual)` – Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
- `assertEquals([String message], expected, actual, tolerance)` – Test that float or double values match. The tolerance is the number of decimals which must be the same.
- `assertNull([message], object)` – Checks that the object is null.
- `assertNotNull([message], object)` – Checks that the object is not null.
- `assertSame([String], expected, actual)` – Checks that both variables refer to the same object.
- `assertNotSame([String], expected, actual)` – Checks that both variables refer to different objects.

JUnit

Execution order

- JUnit assumes that all test methods can be executed in an arbitrary order. Well-written test code should not assume any order, i.e., tests should not depend on other tests.
- As of JUnit 4.11 you can use an annotation to define that the test methods are sorted by method name, in lexicographic order.
- To activate this feature, annotate your test class with:

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

- If you have several test classes, you can combine them into a *test suite*. Running a test suite will execute all test classes in that suite in the specified order.

JUnit

Test suite example

```
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 {

}
```

Available JUnit annotations (1)

Annotation	Description
<code>@Test</code> <code>public void method()</code>	The <code>@Test</code> annotation identifies a method as a test method.
<code>@Test (expected = Exception.class)</code>	Fails if the method does not throw the named exception.
<code>@Test(timeout=100)</code>	Fails if the method takes longer than 100 milliseconds.
<code>@Before</code> <code>public void method()</code>	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
<code>@After</code> <code>public void method()</code>	This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.

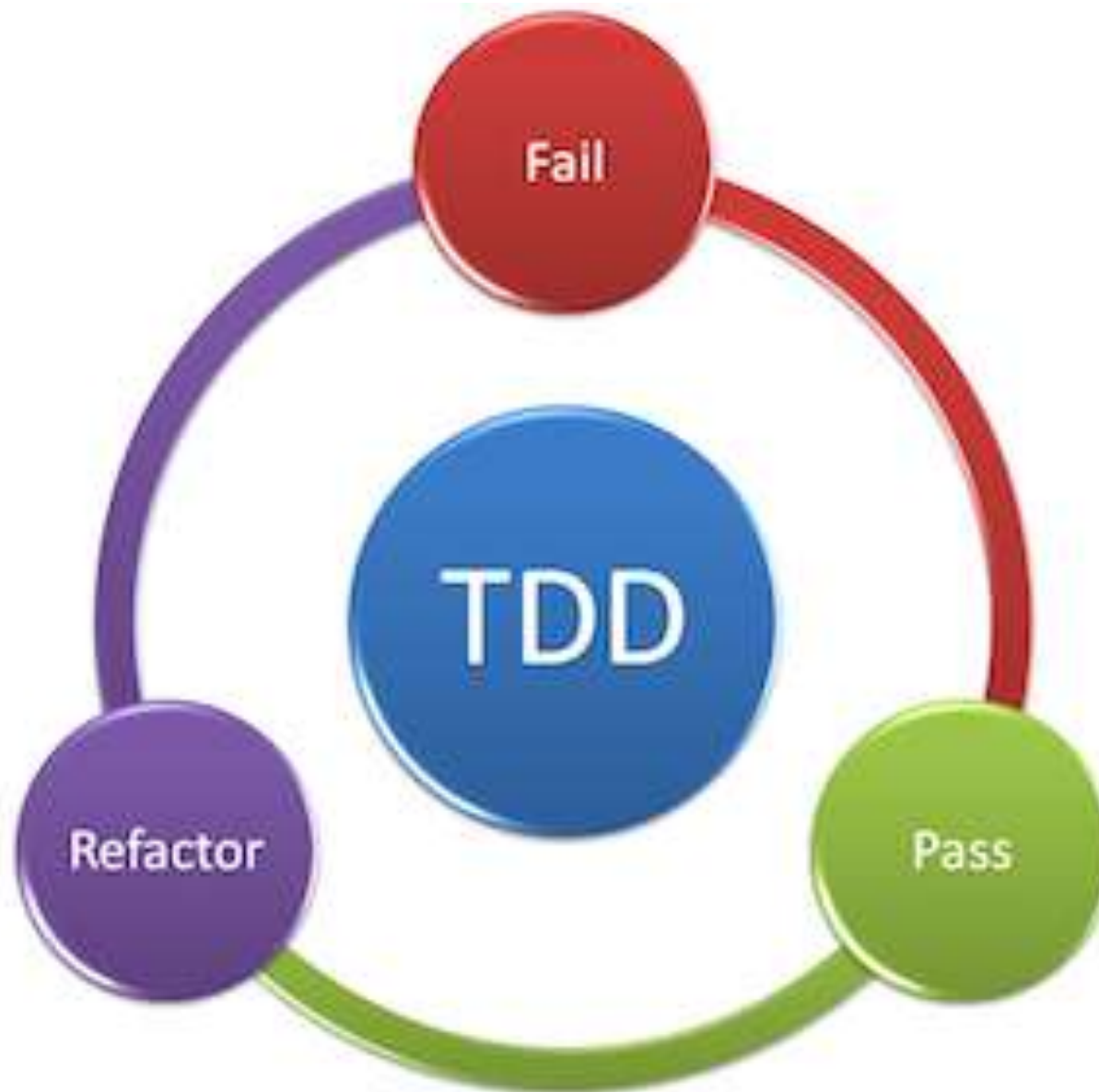
Available JUnit annotations (2)

Annotation	Description
<code>@BeforeClass</code> <code>public static void method()</code>	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit.
<code>@AfterClass</code> <code>public static void method()</code>	This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit.
<code>@Ignore</code>	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.

Какво е Test Driven Development?



Мантрата на TDD





Next...



Thank you

Contact information:

Vasil Vasilev
Kiril Gavrilov

SAP Labs Bulgaria
Sofia, bul.Boris III, 136A

Used sources

Ariana 5 - <http://www.around.com/ariane.html>
https://www.youtube.com/watch?v=gp_D8r-2hwk

Why testing early - <http://www.softwaretestingclass.com/why-testing-should-start-early-in-software-development-life-cycle/>

JUnit - <http://junit.org/>

JavaDoc for JUnit - <http://junit.sourceforge.net/javadoc/overview-summary.html>

Unit Testing with JUnit - <http://www.vogella.com/tutorials/JUnit/article.html>