

Module 1-14

Unit Testing

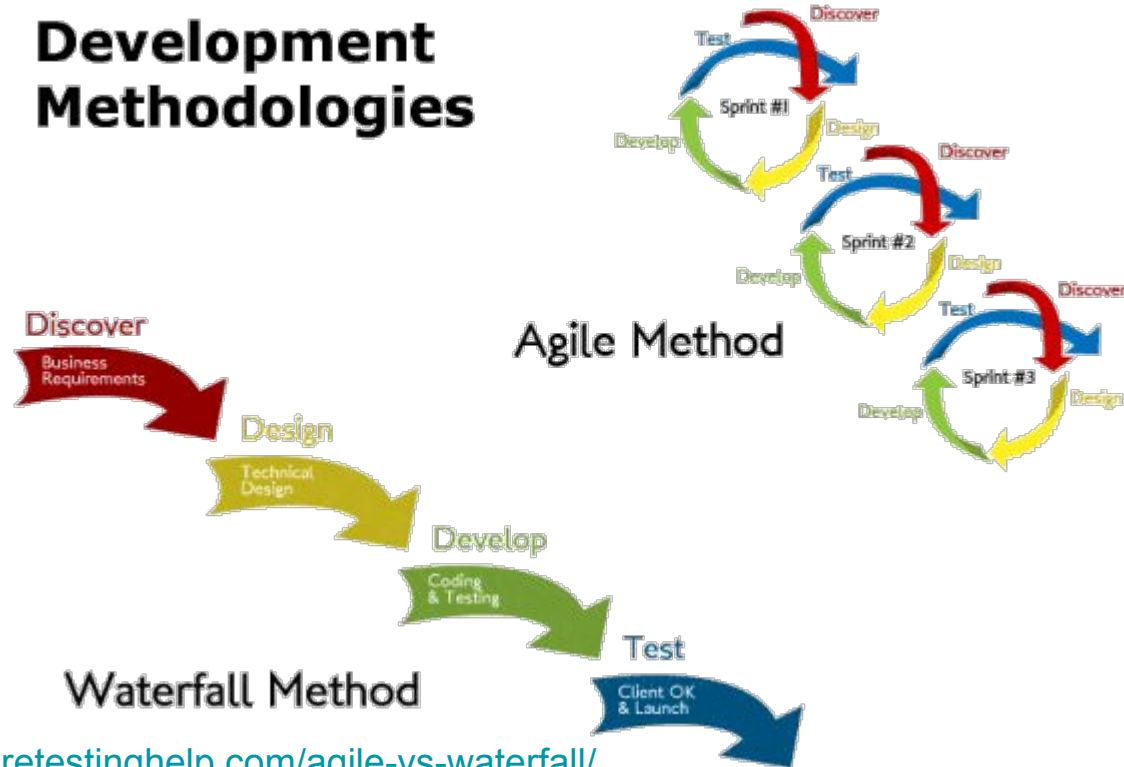
Module 1 Day 14

Can you?

- ... list the pros and cons associated with Manual versus Automated testing
- ... state the difference between Exploratory and Regression testing
- ... state the difference between Unit, Integration and Acceptance testing
- ... create and run Unit tests
- ... choose the proper asserts from an xUnit framework
- ... describe boundary cases and how to spot what the boundary cases are in a piece of code

SDLC - Software Development Life Cycle

Development Methodologies



Testing

Goes without saying... we need a way to test the code we've written.

Testing is a critical part of EVERY SDLC methodology.

The sooner you, as a developer, test, the sooner you identify problems and can move to QA, UAT, and Production.

Manual Testing vs Automated Testing

- Historically, tests were written on a third party tool (i.e. Excel) with a script a tester should follow. The results are recorded.
 - This is a very error prone manual process.
- Over time, testing frameworks were introduced so that we could write code that tests code in your system.
 - This made testing more automated.
 - However, the quality of the tests now partially depends on the developer's knowledge of the testing framework.

Types of Testing

- **Unit Testing:** Tests the smallest units possible (i.e. methods of a class).
- **Integration Testing:** Tests how various units or parts of the program interact with each other.
 - It can also be used to validate some external dependencies like database systems or API's.
- **User Acceptance Testing:** Tests the functionality from the end user's perspective. It can be conducted by a non-technical user.

Other Types of Testing

- **Security Testing:** Is our data safe from unauthorized users?
- **Performance Testing:** it works with 1 user, what about a million?
- **Platform Testing:** Works great on my laptop, what if I pull up the app from my phone?

Unit Testing in Java: Introduction

The most commonly used testing framework in Java is JUNIT.

- JUNIT is written in Java and will leverage all the concepts you've learned so far: declaring variables, calling methods, instantiating objects.
- All related tests can be written in a single test class containing several methods, each method could be a test.
- Each method should contain an assertion, which compares the result of your code against an expected value.

Unit Testing in Java: Assertions

An assertion is the result of a comparison between an actual value of an expected value. Supposed we have a Java method that returned the following:

```
public static boolean divBy2(int i) {  
    return i%2 == 0;  
}
```

Assertion 1: If I run `divBy2(4)` the result of invoking the method should be true.

If `divBy2(4)` returns false, then the assertion has failed.

Assertion 2: If I run `divBy2(5)` the result of invoking the method should be false.

If the method is invoked and the result is false, then the assertion has failed.

Unit Testing in Java: Production Code vs Test Code

- Production code refers to the actual code for your project.
- Test code is the code that is designed to test Production Code

Unit Testing in Java: Example

Production Code

```
package te.examples.testingexamples;

public class MyApp {

    public boolean divBy2(int number) {
        return number%2==0;
    }

    public String concatenator(String [] wordArray) {

        String output = "";

        for (String word : wordArray) {
            output += word;
        }

        return output;}}}
```

These two are tests designed to check if divBy2 is working property.

Test Code

```
// A lot of imports up top, removed for brevity
public class TestContainingClass {

    @Test
    public void threeDivByTwoShouldReturnFalse() {

        MyApp app = new MyApp();
        boolean actualResult = app.divBy2(3);
        boolean expectedResult = false;

        Assert.assertEquals(expectedResult, actualResult);

    }

    @Test
    public void fourDivByTwoShouldReturnTrue() {

        MyApp app = new MyApp();
        boolean actualResult = app.divBy2(4);
        boolean expectedResult = true;

        Assert.assertEquals(expectedResult, actualResult);

    }

}
```

Unit Testing in Java: Anatomy of Test Method

Let's take a closer look at a test method and what happens inside it:

We are using an `@Test` annotation to indicate this method is a test.

Tests are typically void methods, they follow the same syntax rules as regular methods.

We need to bring in the test collaborators, in this case an instance of the class `MyApp`

We run any methods in the collaborator that we want to test, obtain the actual result and compare against what we are expecting.

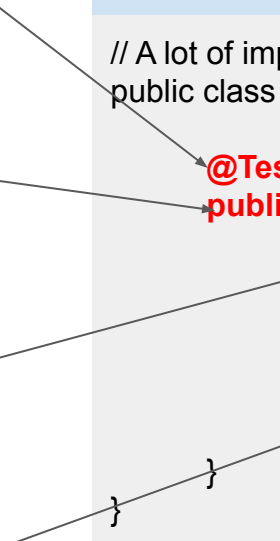
Test Code

```
// A lot of imports up top, removed for brevity
public class TestContainingClass {

    @Test
    public void threeDivByTwoShouldReturnFalse() {

        MyApp app = new MyApp();
        boolean actualResult = app.divBy2(3);
        boolean expectedResult = false;

        Assert.assertEquals(expectedResult, actualResult);
    }
}
```



Unit Testing in Java: Multiple Tests

A testing class can contain multiple tests. The same production method can be called and tested as many times as needed.

This class contains two tests.

A light blue rectangular callout box with the text "This class contains two tests." is positioned to the left of the code. Two arrows originate from the right side of the box: one points to the first **@Test** annotation and the other points to the second **@Test** annotation.

```
public class TestContainingClass {  
    @Test  
    public void threeDivByTwoShouldReturnFalse() {  
        // test content  
    }  
    @Test  
    public void fourDivByTwoShouldReturnTrue() {  
        // test content  
    }  
}
```

Unit Testing in Java: Before & After

You can specify that certain pieces of code be run before and after a test.

```
public class TestContainingClass {  
  
    @Before  
    public void setUp() throws Exception {  
  
        System.out.println("Test starting.");  
    }  
  
    @After  
    public void tearDown() throws Exception {  
  
        System.out.println("Test complete.");  
    }  
  
    @Test  
    public void threeDivByTwoShouldReturnFalse() {  
        // Test content.  
    }  
  
    @Test  
    public void fourDivByTwoShouldReturnTrue() {  
        // Test content.  
    }  
}
```

Anything in the **@Before** block will run right before a test.

Anything in the **@After** block will run right after the test.

So the order of operations is:

1. run setup()
2. Run threeDivByTwo... test
3. run tearDown()
4. run setup()
5. Run fourDivByTwo... test
6. run setup()