Module 1-14

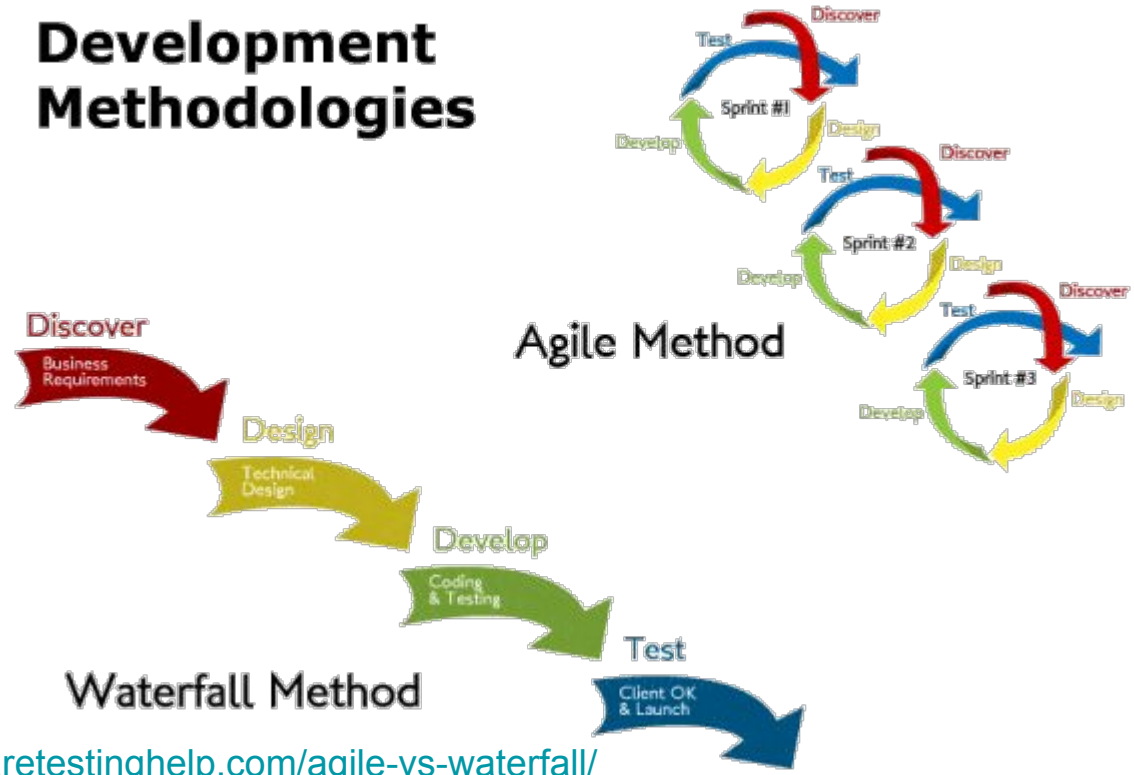
Unit Testing

Module 1 Day 14

Can you?

- … discuss the differences between Waterfall and Agile SDLC Methodologies
- ... 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



https://www.softwaretestinghelp.com/agile-vs-waterfall/

Testing

It should go without saying; but, we need a way to test the code we've written.

Testing is a critical part of EVERY SDLC methodology and every project.

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

Manual Testing vs Automated Testing

- Historically, tests were written out as a series of steps on a third party tool (i.e. Excel) creating a script for a tester to follow. The tester then follows the script and manually records the results.
  - There is a large margin of error due to human interpretation
  - This a very error prone manual process

- Eventually, testing frameworks were introduced that allowed developers to write to test the code in applications.
  - Testing became automated and reliably repeatable
  - The quality of the testing now depends on the test developer’s skill and knowledge of the testing framework

Types of Testing

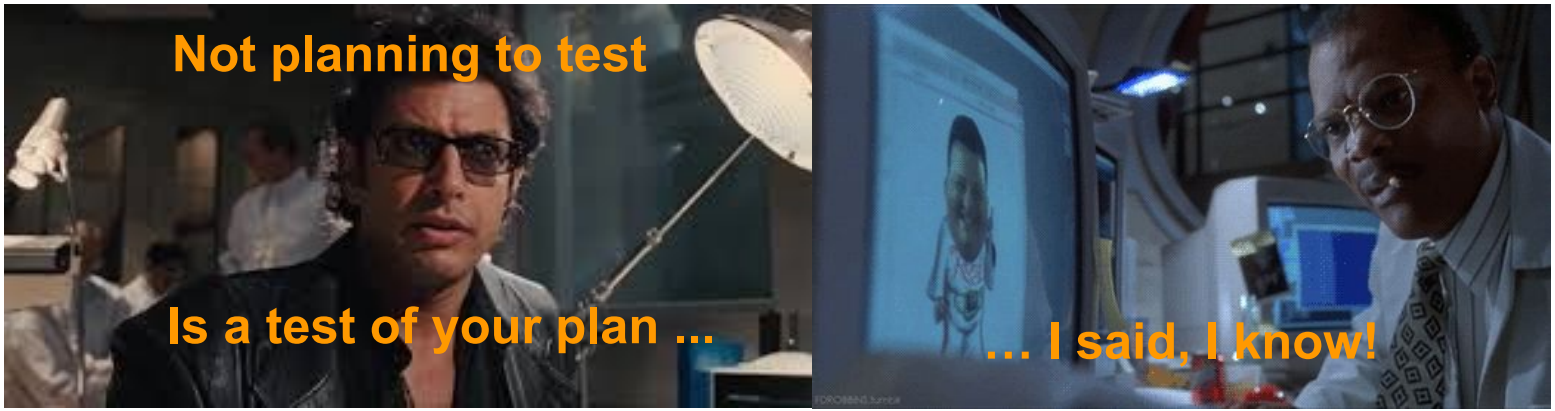
- **Unit Testing**: Tests the smallest units possible (i.e. methods of a class).

- **Integration Testing:** Tests how various units or parts of an application interact with each other.
  - It can also be used to validate external dependencies
    - Database
    - APIs

- **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 typically represents a single 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 a comparison between an actual value of an expected value. Supposed we have a Java method that returned the following:

Application Code

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

Testing Assertions

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 divBy(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

  ApplicationClass.java

- **Test code** is the code that is designed to test Production code

  ApplicationClassTest.java

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 concatArray(String [] wordArray) {

                String output = "";

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

                return output;}}
```

Test Code

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

        @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

```
/** Required Imports Removed For Sake of Example **/
public class MyAppTest{

    @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.

```
public class MyAppTest{

    @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 MyAppTest {

        @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 ***each*** test.

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

The MyAppTest Execution Sequence is:

1. run setUp()

2. Run threeDivByTwo test

3. run tearDown()

4.run setup()

5. Run fourDivByTwo test

6. run tearDown()

Let’s Get to the Code!