JUnit Essentials

CS585 Software Verification and Validation
http://cs585.yusun.io
January 7, 2015

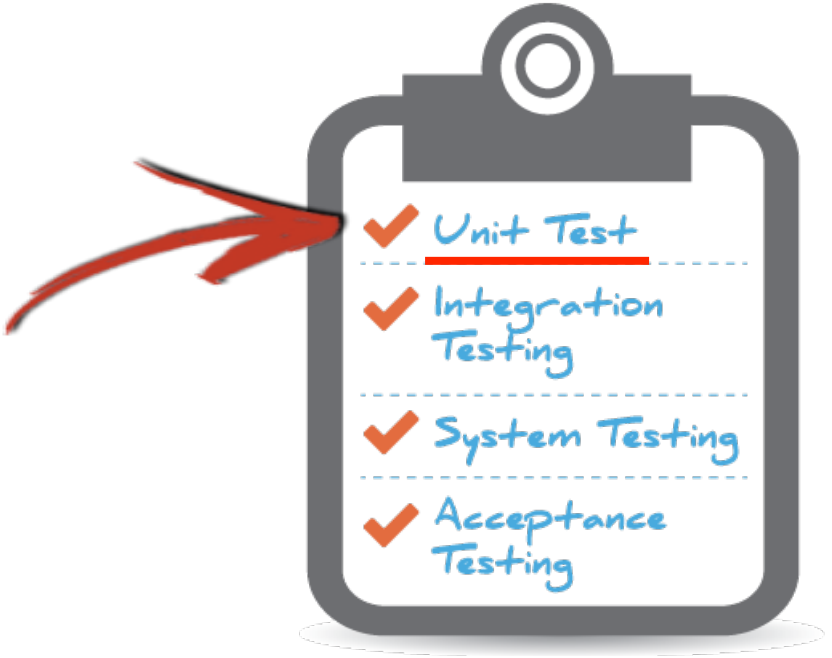
Yu Sun, Ph.D.
http://yusun.io
yusun@cpp.edu

Announcements

- Decide your project topic ASAP

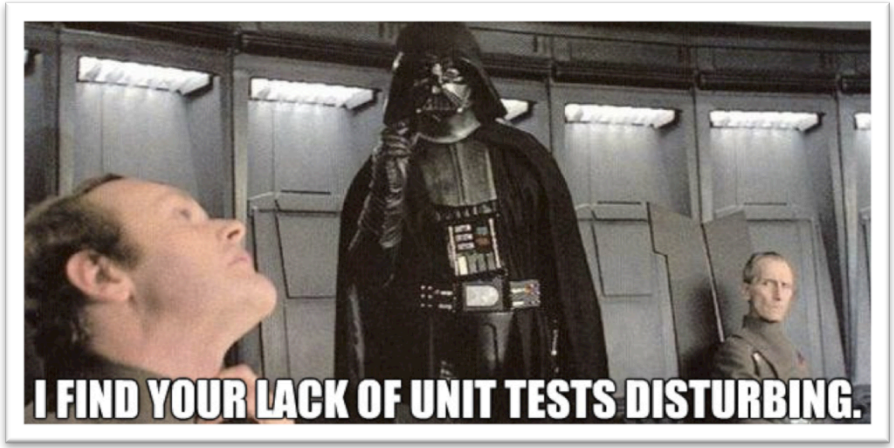
The Most Fundamental Testing Step

- The smallest test unit
- Test every single function



A MUST-HAVE Skill for Developers

- You are required to write unit test for every change you made
- There is no way that you can skip the process, because your code will be reviewed by your peer developers

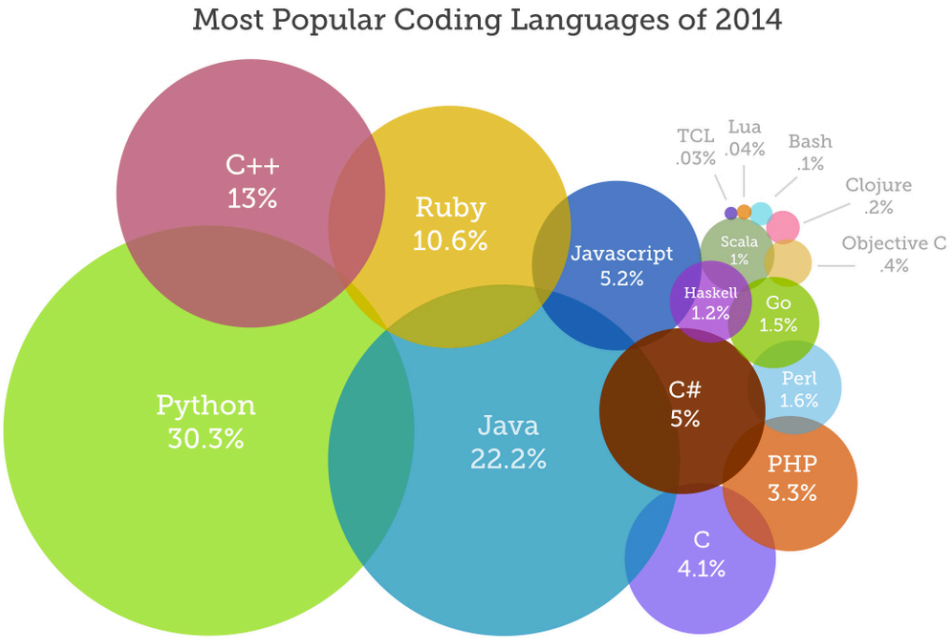


Writing Good Unit Tests is Non-Trivial

- Write testable code
- High test coverage
- Use test patterns
- Handle tricky testing scenarios

Never Ignore Java

- A complete general-purpose language
- Platform-independent
- Web/Mobile/Desktop
- Widely used in the industry
- New and cool features



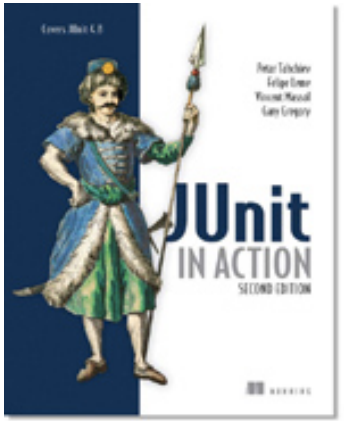
From JUnit to xUnit

- JUnit is a modern and mature testing framework
- Learning JUnit helps learning xUnit
  - ASP, C++, C#, Eiffel, Delphi, Perl, PHP, Python, REBOL, Smalltalk, and Visual Basic

JUnit

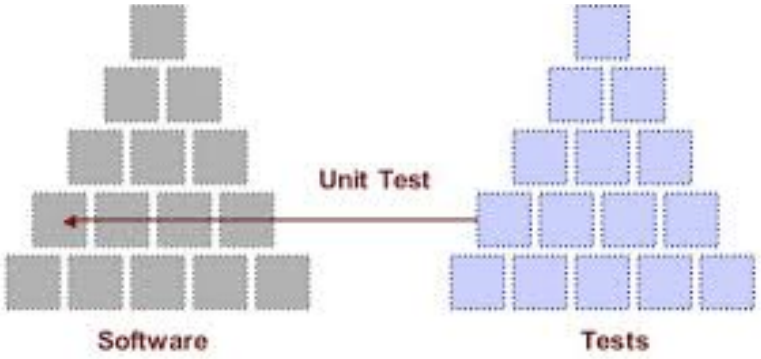
JUnit Resources

- http://junit.org
- Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in Action*. Manning Publications Co., 2010.
- Latest version: JUnit 4.12
- Source code: https://github.com/junit-team/junit



Unit Test – Definition

- A *unit test* examines the behavior of a distinct unit of work
- Within a Java application, the "distinct unit of work" is often (but not always) a single method



How to Test without JUnit

- We code, we compile, we run, and we test
- When test, we
  - click on a button
  - add a record
  - delete a record
  - …

JUnit from Scratch – Calculator Example

Listing 1.1 The test calculator class

```
public class Calculator {
   public double add(double number1, double number2) {
      return number1 + number2;
   }
}
```

JUnit from Scratch – A Simple Test

Listing 1.1 The test calculator class

```
public class Calculator {
   public double add(double number1, double number2) {
      return number1 + number2;
   }
}
```

Listing 1.2 A simple test calculator program

```
public class CalculatorTest {
   public static void main(String[] args) {
      Calculator calculator = new Calculator();
      double result = calculator.add(10,50);
      if (result != 60) {
         System.out.println("Bad result: " + result);
      }
   }
}
```

JUnit from Scratch – A Better Test

Listing 1.3 A (slightly) better test calculator program

```
public class CalculatorTest {

   private int nbErrors = 0;

   public void testAdd() {
      Calculator calculator = new Calculator();
      double result = calculator.add(10, 50);
      if (result != 60) {
         throw new IllegalStateException("Bad result: " + result);
      }
   }
```

① Explicitly report the error using Java exception

JUnit from Scratch – A Better Test (cont.)

```
    public static void main(String[] args) {
        CalculatorTest test = new CalculatorTest();
        try {
            test.testAdd();
        }
        catch (Throwable e) {
            test.nbErrors++;
            e.printStackTrace();
        }
        if (test.nbErrors > 0) {
            throw new IllegalStateException("There were " + test.nbErrors
                + " error(s)");
        }
    }
}
```

② A simple framework to count and summarize the tests

Unit Test Best Practices

- Each unit test should run independently of all other unit tests
- The framework should detect and report errors test by test
- It should be easy to define which unit tests will run

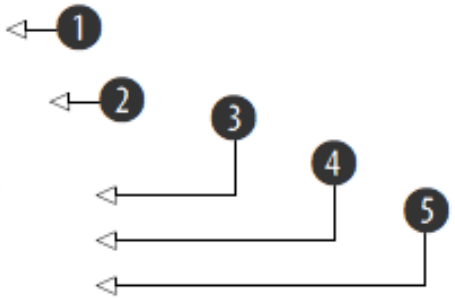
The Process should be Automated

- In 1997, Erich Gamma and Kent Beck created JUnit 1.0

JUnit from Scratch – The JUnit Solution

Listing 1.4 The JUnit `CalculatorTest` program

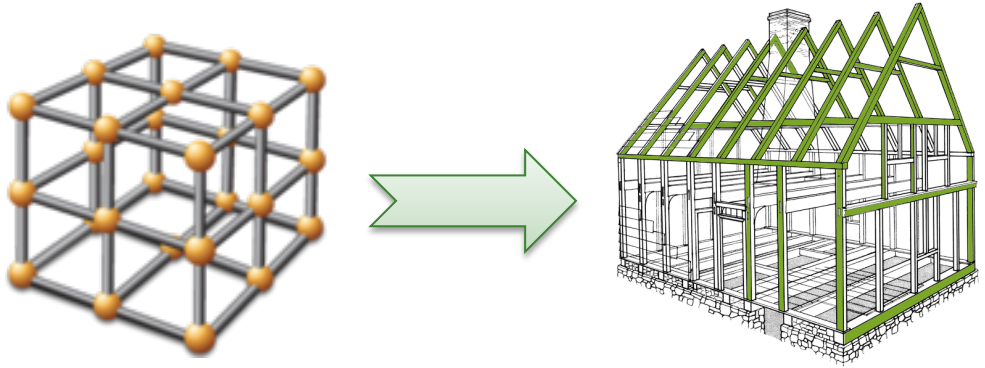
```
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {

   @Test
   public void testAdd() {
      Calculator calculator = new Calculator();
      double result = calculator.add(10, 50);
      assertEquals(60, result, 0);
      }
}
```



① JUnit has some naming conventions
② @Test indicates the unit test method
③ Create a new instance – to be independent
④ Execute the method to be tested
⑤ Check the result using JUnit

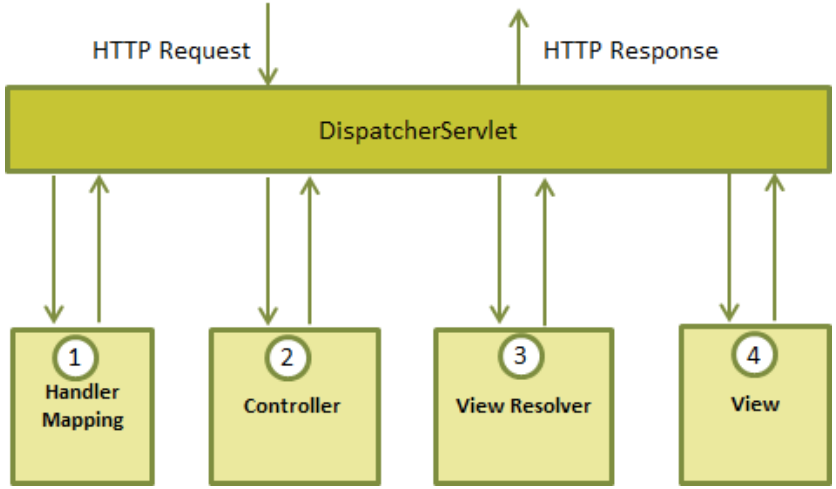
JUnit is a Software Framework

- A *framework* is a semi-complete application
- It provides a reusable, common structure to share among applications
- Developers incorporate the framework into their own application and extend it to meet their specific needs

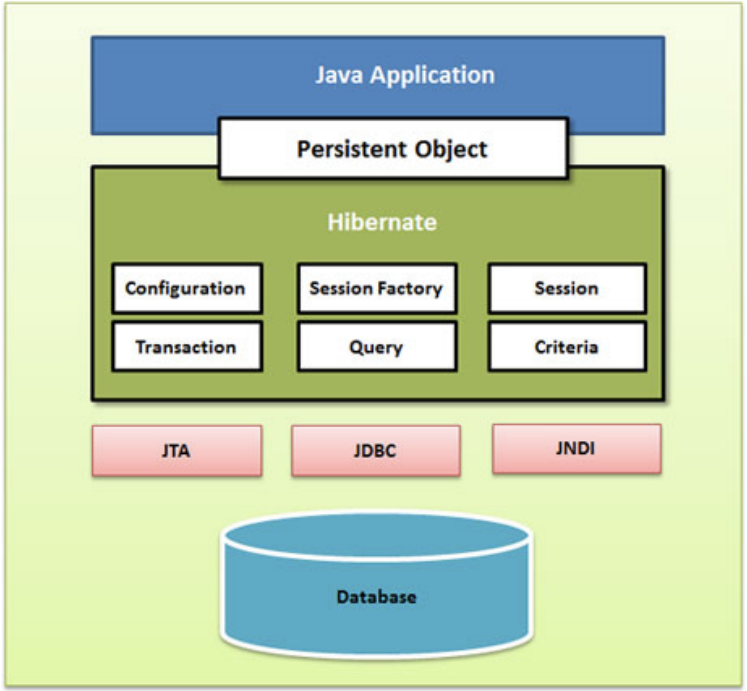


Software Framework Examples

- Spring MVC



- Hibernate



Core JUnit - @Test

- Annotate the test methods
  - Public
  - No arguments
  - No return

Core JUnit – Assert

assertXXX method	What it's used for
`assertArrayEquals("message", A, B)`	Asserts the equality of the A and B arrays.
`assertEquals("message", A, B)`	Asserts the equality of objects A and B. This `assert` invokes the `equals()` method on the first object against the second.
`assertSame("message", A, B)`	Asserts that the A and B objects are the same object. Whereas the previous `assert` method checks to see that A and B have the same value (using the `equals` method), the `assertSame` method checks to see if the A and B objects are one and the same object (using the == operator).
`assertTrue("message", A)`	Asserts that the A condition is true.
`assertNotNull("message", A)`	Asserts that the A object isn't null.

Core JUnit – Test Runner

- Test runner executes the test methods

```
@RunWith (SpringJUnit4ClassRunner.class)
```

```
@RunWith (Parameterized.class)
```

```
@RunWith (JUnit38ClassRunner.class)
```

Core JUnit – Test Suite

- The *Suite* is a container used to gather tests for the purpose of grouping and invocation

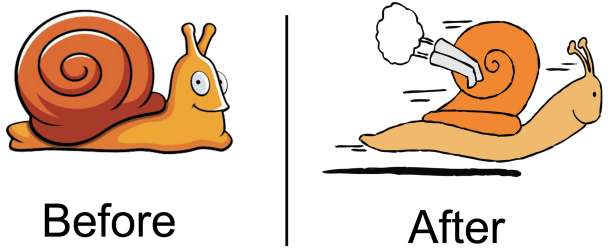
```
[...]
@RunWith(value=Suite.class)
@SuiteClasses(value = {TestCaseB.class})
public class TestSuiteB {
}

[...]
@RunWith(value = Suite.class)
@SuiteClasses(value = {TestSuiteA.class, TestSuiteB.class})
public class MasterTestSuite{
}
```

Core JUnit - @Before/@After

- The @Before and @After annotated methods are executed right before/after the execution of each one of your @Test methods and regardless of whether the test failed or not
- You can have as many of these methods as you want, but beware that if you have more than one of the @Before/@After methods, *the order of their execution is not defined*
- Extract the common logic, like instantiating your domain objects and setting them up in some known state
- The method must be *public*

Core JUnit - @BeforeClass/@AfterClass

- The methods annotated with @BeforeClass and @AfterClass will get executed only once, before/after all of your @Test methods
- As with the @Before and @After annotations, you can have as many of these methods as you want, and again the order of the execution is unspecified
- The method must be *public* and *static*



Core JUnit – Test Exceptions

Listing 3.14 Testing methods that throw an exception

```
public class TestDefaultController
{
[...]
   @Test(expected=RuntimeException.class)
   public void testGetHandlerNotDefined()
   {
      SampleRequest request = new SampleRequest("testNotDefined");

      //The following line is supposed to throw a RuntimeException
      controller.getHandler(request);
   }

   @Test(expected=RuntimeException.class)
   public void testAddRequestDuplicateName()
   {
      SampleRequest request = new SampleRequest();
      SampleHandler handler = new SampleHandler();

      // The following line is supposed to throw a RuntimeException
      controller.addHandler(request, handler);
   }
}
```

Core JUnit – Timeout

Listing 3.15 Timeout tests

```
[...]
public class TestDefaultController
{
[...]
   @Test(timeout=130)
   public void testProcessMultipleRequestsTimeout()
   {
     Request request;
     Response response = new SampleResponse();
     RequestHandler handler = new SampleHandler();

     for(int i=0; i< 99999; i++)
     {
        request = new SampleRequest(String.valueOf(i));
        controller.addHandler(request, handler);
        response = controller.processRequest();
        assertNotNull(response);
        assertNotSame(ErrorResponse.class, response.getClass());
     }
   }
```

Core JUnit – @Ignore

Listing 3.16 Ignoring a test method in JUnit 4.x

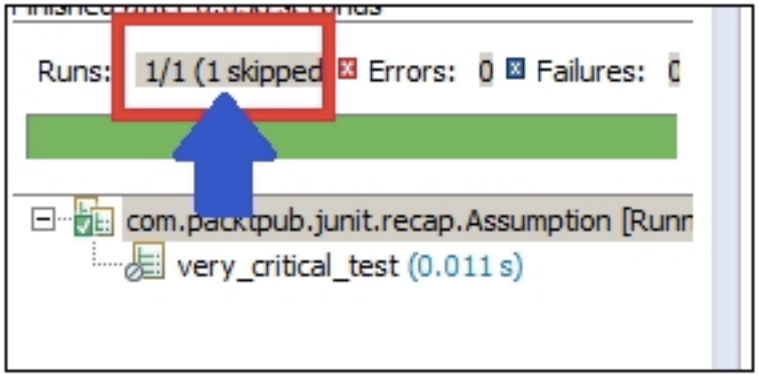
```
[...]
@Test(timeout=130)
@Ignore(value="Ignore for now until we decide a decent time-limit")
public void testProcessMultipleRequestTimeout()
{
    [...]
}
```

- Always specify a reason for skipping a test

Core JUnit – Assume

- Skip the test if the assumption is not satisfied

```
Assume.assumeFalse(isSonarRunning);
```



Core JUnit – Hamcrest

- Declaratively specify simple matching rules

```
@Before
   public void setUpList() {
          values = new ArrayList<String>();
      values.add("x");
      values.add("y");
      values.add("z");
   }

@Test
public void testWithoutHamcrest() {
   assertTrue(values.contains("one")
           || values.contains("two")
           || values.contains("three"));
   }
}
```

Core JUnit – Hamcrest

- Declaratively specify simple matching rules

```
@Before
   public void setUpList() {
          values = new ArrayList<String>();
      values.add("x");
      values.add("y");
      values.add("z");
   }
```

```
@Test
public void testWithHamcrest() {
   assertThat(values, hasItem(anyOf(equalTo("one"), equalTo("two"),
                  equalTo("three"))));
}
}
```

Core JUnit – Hamcrest

Core	Logical
`sameInstance`	Tests object identity.
`notNullValue, nullValue`	Tests for null values (or non-null values).
`hasProperty`	Tests whether a JavaBean has a certain property.
`hasEntry, hasKey, hasValue`	Tests whether a given Map has a given entry, key, or value.
`hasItem, hasItems`	Tests a given collection for the presence of an item or items.
`closeTo, greaterThan, greaterThanOrEqual, lessThan, lessThanOrEqual`	Test whether given numbers are close to, greater than, greater than or equal to, less than, or less than or equal to a given value.
`equalToIgnoringCase`	Tests whether a given string equals another one, ignoring the case.
`equalToIgnoringWhiteSpace`	Tests whether a given string equals another one, by ignoring the white spaces.
`containsString, endsWith, startWith`	Test whether the given string contains, starts with, or ends with a certain string.

Core JUnit – Hamcrest

Core	Logical
`anything`	Matches absolutely anything. Useful in some cases where you want to make the assert statement more readable.
`is`	Is used only to improve the readability of your statements.
`allOf`	Checks to see if all contained matchers match (just like the && operator).
`anyOf`	Checks to see if any of the contained matchers match (like the || operator).
`not`	Traverses the meaning of the contained matchers (just like the ! operator in Java).
`instanceOf, isCompatibleType`	Match whether objects are of compatible type (are instances of one another).

Core JUnit – Test Order

- By default, JUnit executes methods with random order
- @FixMethodOrder
  - MethodSorters.JVM
  - MethodSorters.NAME_ASCENDING
  - MethodSorters.DEFAULT

Core JUnit – Rules

- Timeout Rule

```
public class TimeoutTest {

    @Rule
    public Timeout globalTimeout =  new Timeout(20);

    @Test
    public void testInfiniteLoop1() throws InterruptedException{
      Thread.sleep(30);
    }
    @Test
    public void testInfiniteLoop2() throws InterruptedException{
      Thread.sleep(30);
    }
}
```

Core JUnit – Rules

- ExpectedException Rule

```
public class ExpectedExceptionRuleTest {

    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void throwsNothing() {
    }
    @Test
    public void throwsNullPointerException() {
      thrown.expect(NullPointerException.class);
      throw new NullPointerException();
    }
}
```

Core JUnit – Rules

- TemporaryFolder Rule: allows the creation of files and folders that are guaranteed to be deleted when the test method finishes (whether it passes or fails)

```
public class TemporaryFolderRuleTest {

  @Rule
  public TemporaryFolder folder = new TemporaryFolder();

  @Test
  public void testUsingTempFolder() throws IOException {
    File createdFile = folder.newFile("myfile.txt");
    File createdFolder = folder.newFolder("mysubfolder");
  }
}
```

Core JUnit – Rules

- ErrorCollector Rule: allows the execution of a test to continue after the first problem is found

```
public class ErrorCollectorTest {

   @Rule
   public ErrorCollector collector = new ErrorCollector();

   @Test
   public void fails_after_execution() {
        collector.checkThat("a", equalTo("b"));
        collector.checkThat(1, equalTo(2));
        collector.checkThat("ae", equalTo("g"));
   }
}
```

Best Practices – Test One Object at a Time

- Make fine granularity unit test
- It helps to isolate problems

Best Practices – Choose Meaningful Names

- Following *XXXTest* pattern for each test class
- Follow *testXXX* pattern for each test method
- Long and verbose names are OK

Best Practices – Explain the Failure Reason

- Use *assertXXX(String message, …);*

```
assertNotNull("Must not return a null response", response);
assertEquals("Response should be of type SampleResponse",
             SampleResponse.class, response.getClass());
```

Best Practices – One UT Equals One @Test

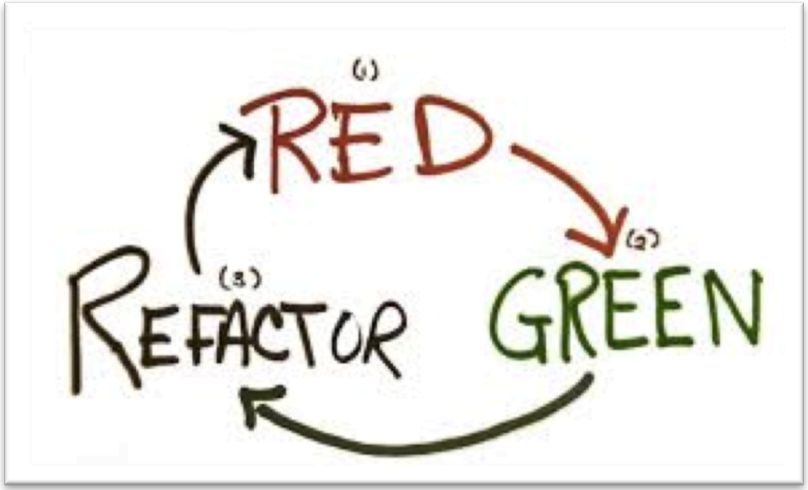
- Don't put several tests into one method; otherwise, really hard to read and understand
- One assertion per @Test method
- Don't write test method without using assertions

Best Practices – Test Anything that could Fail

- Test everything

Best Practices – Let Test Improve Your Code

- A test case is a user of your code. It's only when using code that you find its shortcomings
- Listen your tests and *refactor* your code so that it's easier to use



Why do we need Unit Tests?

- Allow greater test coverage than functional tests
- Increase team productivity
- Detect regressions and limit the need for debugging
- Changing and refactoring with confidence
- Improve implementation
- Document expected behavior
- Enable code coverage and other metrics

Cannot be Agile without Unit Tests