

Test-Driven Development

JUnit

EECS 2311 - Software Development Project

Click to edit Master text styles

Second level

Third level

Fourth level

Fifth level

Wednesday, January 24, 2018

Unit Testing

- Testing the internals of a class
- Black box testing
 - Test public methods
- Classes are tested in isolation
 - One test class for each application class

Test – Driven Development

- TDD is a software development approach whereby you write your test cases **before** you write any implementation code
- Tests drive or dictate the code that is developed
- An indication of “intent”
 - Tests provide a specification of “what” a piece of code actually does
 - Some might argue that “tests are part of the documentation”

TDD Stages

1. Write a single test.
2. Compile it. It should not compile because you have not written the implementation code
3. Implement **just enough** code to get the test to compile
4. Run the test and see it **fail**
5. Implement **just enough** code to get the test to pass
6. Run the test and see it **pass**
7. Refactor
8. Repeat

JUnit

- JUnit is a framework for writing and running tests
 - Written by Erich Gamma (of Design Patterns fame) and Kent Beck (creator of XP methodology)
 - Uses Java features such as annotations and static imports

Terminology

- A **test fixture** sets up the data (both objects and primitives) that are needed for every test
 - Example: If you are testing code that updates an employee record, you need an employee record to test it on
- A **unit test** is a test of a *single* class
- A **test case** tests the response of a single method to a particular set of inputs
- A **test suite** is a collection of test cases
- A **test runner** is software that runs tests and reports results

Structure of a JUnit test class

- To test a class named **Fraction**
- Create a test class **FractionTest**

```
import org.junit.*;  
import static org.junit.Assert.*;  
public class FractionTest  
{  
    ...  
}
```

Test fixtures

- Methods annotated with `@Before` will execute before every test case
- Methods annotated with `@After` will execute after every test case

```
@Before  
public void setUp() {...}  
@After  
public void tearDown() {...}
```


Class Test fixtures

- Methods annotated with `@BeforeClass` will execute once *before* all test cases
- Methods annotated with `@AfterClass` will execute once *after* all test cases
- These are useful if you need to allocate and release expensive resources once

Test cases

- Methods annotated with `@Test` are considered to be test cases

```
@Test
public void testadd() {...}
@Test
public void testToString() {...}
```

What JUnit does

- For *each* test case **t**:
 - JUnit executes all `@Before` methods
 - Their order of execution is not specified
 - JUnit executes **t**
 - Any exceptions during its execution are logged
 - JUnit executes all `@After` methods
 - Their order of execution is not specified
- A report for all test cases is presented

Within a test case

- Call the methods of the class being tested
- Assert what the correct result should be with one of the provided **assert methods**
- These steps can be repeated as many times as necessary
- An assert method is a JUnit method that performs a test, and throws an **AssertionError** if the test fails
 - JUnit catches these exceptions and shows you the results

List of assert methods 1

- `assertTrue(boolean b)`
`assertTrue(String s, boolean b)`
 - Throws an `AssertionError` if *b* is False
 - The optional message *s* is included in the Error
- `assertFalse(boolean b)`
`assertFalse(String s, boolean b)`
 - Throws an `AssertionError` if *b* is True
 - All assert methods have an optional message

Example: Counter class

- Consider a trivial “counter” class
- The constructor creates a counter and sets it to zero
- The **increment** method adds one to the counter and returns the new value
- The **decrement** method subtracts one from the counter and returns the new value
- An example and the corresponding JUnit test class can be found on the course website

List of assert methods 2

- `assertEquals(Object expected,
Object actual)`
- Uses the `equals` method to compare the two objects
- Primitives can be passed as arguments thanks to autoboxing
- Casting may be required for primitives
- There is also a version to compare arrays

List of assert methods 3

- `assertSame(Object expected,
Object actual)`
 - Asserts that two references are attached to the same object (using `==`)
- `assertNotSame(Object expected,
Object actual)`
 - Asserts that two references are not attached to the same object

List of assert methods 4

- `assertNull(Object object)`
Asserts that a reference is null
- `assertNotNull(Object object)`
Asserts that a reference is not null
- `fail()`
Causes the test to fail and throw an `AssertionError`
 - Useful as a result of a complex test, or when testing for exceptions

Testing for exceptions

- If a test case is expected to raise an exception, it can be noted as follows

```
@Test(expected = Exception.class)
public void testException() {
    //Code that should raise an exception
    fail("Should raise an exception");
}
```

The assert statement

- A statement such as
`assert boolean_condition;`
will also throw an `AssertionError` if the *boolean_condition* is false
- Can be used instead of the Junit `assertTrue` method

Ignoring test cases

- Test cases that are not finished yet can be annotated with `@Ignore`
- JUnit will not execute the test case but will report how many test cases are being ignored

JUnit in Eclipse

- JUnit can be downloaded from github
- If you use Eclipse, as in this course, you do not need to download anything
- Eclipse contains wizards to help with the development of test suites with JUnit
- JUnit results are presented in an Eclipse window

JUnit 5

- JUnit 5 was released last year
- Eclipse support for JUnit 5 was only recently released
- You can use JUnit 5 for the course if you want
- You can also write JUnit 4 style test cases if you have JUnit 5 installed
 - Include junit-vintage in your build path

JUnit 4 vs. JUnit 5

- Some annotations have been updated
 - @BeforeClass → @BeforeAll
 - @Before → @BeforeEach
 - @AfterClass → @AfterAll
 - @After → @AfterEach
 - @Ignore → @Disabled
- Exceptions can be asserted with **assertThrows**

JUnit 5 new features

- **assertAll()** – tests a number of assertions together
- **assertTimeout()** – test that a piece of code will finish within a particular timeframe
- Assumptions – Running the test case only if the assumption holds
- Nested test cases
- Tagging test cases

Hello World demo

- Run Eclipse
- File -> New -> Project, choose Java Project, and click Next. Type in a project name, e.g. ProjectWithJUnit.
- Click Next
- Click Create New Source Folder, name it test
- Click Finish
- Click Finish

Create a class

- Right-click on ProjectWithJUnit
Select New -> Package
Enter package name, e.g. **eeecs2311.week3**
Click Finish
- Right-click on eeecs2311.week3
Select New -> Class
Enter class name, e.g. **HelloWorld**
Click Finish

Create a class - 2

- Add a dummy method such as
`public String say() { return null; }`
- Right-click in the editor window and select Save

Create a test class

- Right-click on the HelloWorld class
Select New -> Junit Test Case
- **Change** the source folder to test as opposed to src

Create a test class

- Check to create a setup method
- Click Next
- Check the checkbox for the say method
 - This will create a stub for a test case for this method
- Click Finish
- Click OK to “Add JUnit 4 library to the build path”
- The HelloWorldTest class is created
- The first version of the test suite is ready

Run the test class - 1st try

- Right click on the HelloWorldTest class
- Select Run as -> JUnit Test
- The results appear in the left
- The automatically created test case fails

Create a better test case

- Declare an attribute of type HelloWorld
`HelloWorld hi;`
- The setup method should create a HelloWorld object
`hi = new HelloWorld();`
- Modify the testSay method body to
`assertEquals("Hello World!", hi.say());`

Run the test class - 2nd try

- Save the new version of the test class and re-run
- This time the test fails due to expected and actual not being equal
- The body of the method `say` has to be modified to `return "Hello World!";` for the test to pass

Create a test suite

- Right-click on the `eeecs2311.week3` package in the test source folder
- Select New -> Class. Name the class **AllTests**.
- Modify the class text so it looks like class AllTests for the Counter example on the course website (keep the package declaration)
- Change CounterTest to HelloWorldTest
- Run with Run -> Run As -> JUnit Test
- Add more test classes separated by commas

Lab Task

- Assume the Counter class is modified as follows:
 - A reset method is added to change the counter value to 0.
 - Overloaded versions of increment and decrement are added. They receive an int as an argument to inc/dec the counter by that amount
- **Each student** must implement and test these methods. You must have at least 5 test cases.
- In the lab on Monday, you must present your test cases to the TA and demonstrate running them