Test-Driven Development
JUnit

CSE 2311 - Software Development Project

Click to edit Master text styles
Second level
Third level
Fifth level

Tuesday, January 20, 2015

YORK UNIVERSITÉ UNIVERSITY

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 http://junit.sourceforge.net/
- 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

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. **eecs2311.week3**
  Click Finish
- Right-click on eecs2311.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

- Import the class under test
```
import eecs2311.week3.HelloWorld;
```
- 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 eecs2311.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
- Change CounterTest to HelloWorldTest
- Run with Run -> Run As -> JUnit Test
- You can easily add more test classes
- Homework on next slide

Homework

- Each team member must write at least 5 test cases for one of the classes that you have already developed
- In the lab on Monday, you must present your test cases to the TA and demonstrate running them

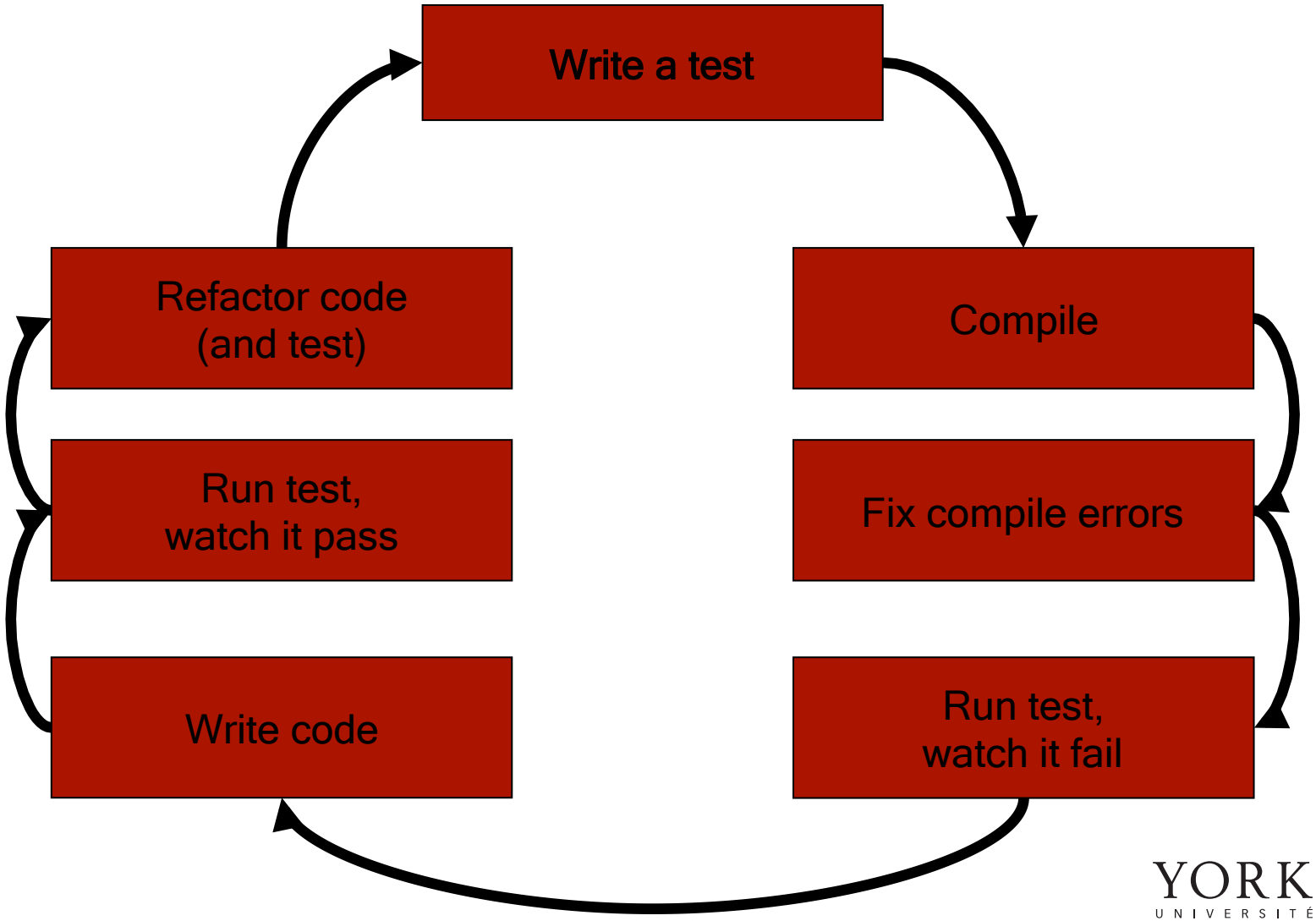
More on TDD

- Before you write code, think about what it will do.
- Write a test that will use the methods you haven't even written yet.

- A test is not something you "do", it is something you "write" and run once, twice, three times, etc.
  - It is a piece of code
  - Testing is therefore "automated"
  - Repeatedly executed, even after small changes

- *The following TDD slides are based on a slide set by Craig Murphy*

TDD Stages



Why TDD?

- Programmers dislike testing
  - They will test reasonably thoroughly the first time
  - The second time however, testing is usually less thorough
  - The third time, well..
- Testing is considered a “boring” task
- Testing might be the job of another department / person
- TDD encourages programmers to maintain an exhaustive set of repeatable tests
  - Tests live alongside the Class/Code Under Test (CUT)
  - With tool support, tests can be run selectively
  - The tests can be run after every single change

Summary

- TDD does not replace traditional testing
  - It defines a proven way that ensures effective unit testing
  - Tests are working examples of how to invoke a piece of code
    - Essentially provides a working specification for the code
- No code should go into production unless it has associated tests
  - Catch bugs before they are shipped to your customer
- No code without tests
- Tests determine, or dictate, the code

Summary

- TDD means less time spent in the debugger

- TDD negates fear
  - Fear makes developers communicate less
  - Fear makes developers avoid repeatedly testing code
    - Afraid of negative feedback

Summary

- TDD promotes the creation of a set of “programmer tests”
  - Automated tests that are written by the programmer
  - Exhaustive
  - Can be run over and over again
- TDD allows us to **refactor**, or change the implementation of a class, without the fear of breaking it
  - TDD and refactoring go hand-in-hand
- With care, [some] User Acceptance Tests can be codified and run as part of the TDD process

Resources

- JUnit: http://junit.sourceforge.net
- NUnit: http://www.nunit.org
- CSUnit: http://www.csunit.org

XP approach to testing

- In the Extreme Programming approach
  - Tests are written before the code itself
  - If the code has no automated test cases, it is assumed not to work
  - A testing framework is used so that automated testing can be done after every small change to the code
    - This may be as often as every 5 or 10 minutes
  - If a bug is found after development, a test is created to keep the bug from coming back

XP consequences

- Fewer bugs
- More maintainable code
- The code can be refactored without fear
- Continuous integration
  - During development, the program *always works*
  - It may not do everything required, but what it does, it does right