Testing

CMSC 202

Overview

- What is software testing?
- What is unit testing?
- Why/When to test?
- Intro to JUnit
- What makes a good test?
- What to test?

What is Software Testing?

- Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results

— William Hetzel
"The Complete Guide to Software Testing"

Types of Software Testing

- Unit Testing
  - Verifies the functionality of a specific chunk of code, usually at the function/class level
- Integration Testing
  - Testing of combined modules as a whole
- System Testing
  - Tests fully integrated system against requirements
- System Integration Testing
  - Testing between multiple systems

Unit Testing

- A unit test is a piece of code written by a developer that exercises a very small, specific area of functionality in the code being tested
- Usually a unit test exercises some particular method in a particular context

— Andy Hunt & Dave Thomas
“Pragmatic Unit Testing”

Unit Testing

- Also known as component testing
- In OOP, typically ensures that method/class works as designed
- Written by developers to test their code
  - Also known as white box testing

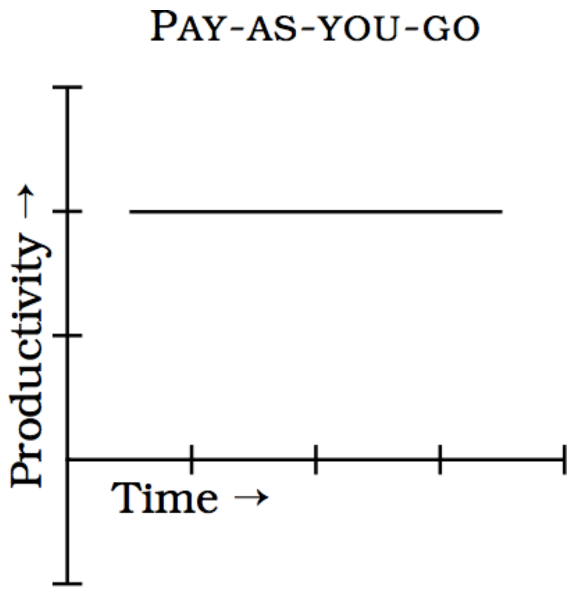
Why Test?

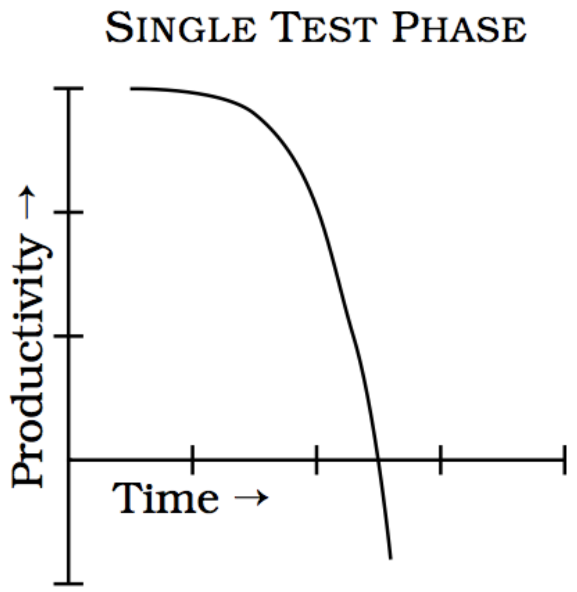
- You wouldn’t do this without a safety net
- Why develop your code without one?

When to Test?

- How many of you write almost all of your code and then write some tests…
  - To fulfill project requirements?
  - To exercise and test your code?
- How many of you incrementally write tests to exercise code as your write it?
- Anyone write the tests first?

Pay Now or Pay Later





- It’s cheaper in the long run to “pay as you go”
- Minimizes trying to solve many problems at once at the end of your development cycle

Test Driven Development

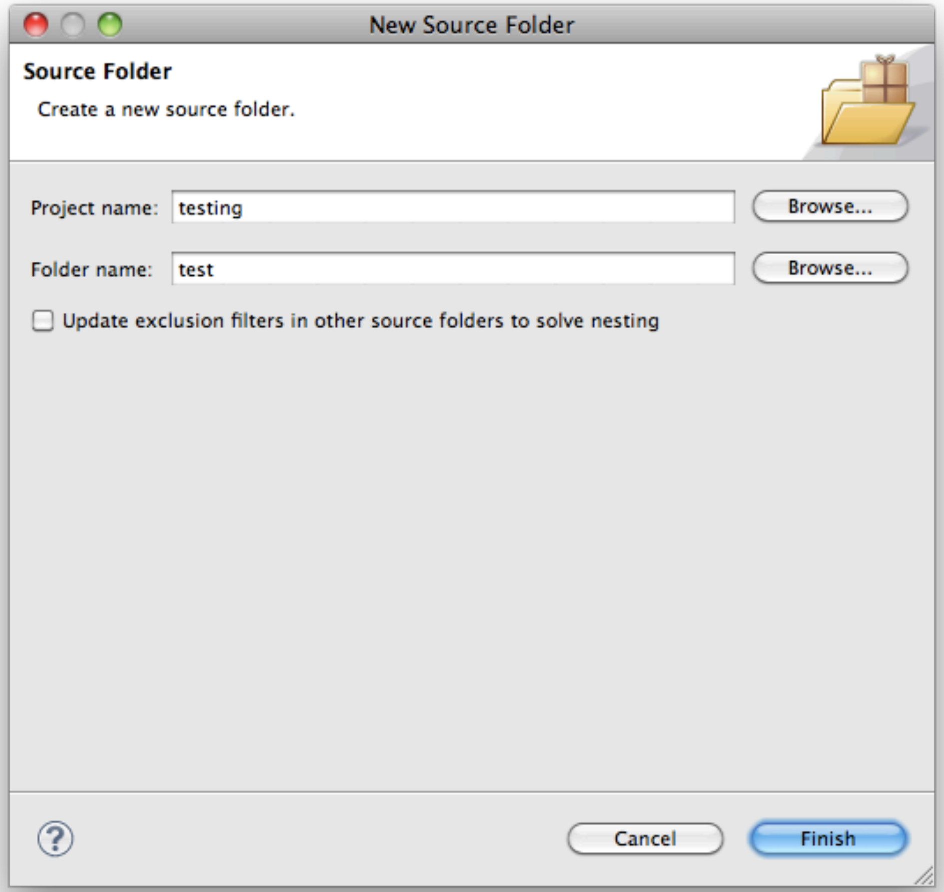
- Test Driven Development (TDD) takes this "pay early" approach a step further by requiring that you write the tests before writing non-test code…
    1. Add test
    2. Run tests, new tests should fail
    3. Write code to satisfy tests
    4. Re-run tests, all tests should pass
    5. Refactor as needed
    6. Repeat

Unit Testing with JUnit

- JUnit is a widely used unit testing framework for Java written by Erich Gamma & Kent Beck
- JUnit support is integrated into many popular Java IDEs including Eclipse and NetBeans
- Instead of testing a code in its main, we're going to create special JUnit aware classes to test our classes

Test Folder

- To keep things tidy, let's create a separate source folder to house the JUnit test classes
- To do so, right click in package explorer and select New → Source Folder and name it "test"



New Source Folder

Source Folder

Create a new source folder.

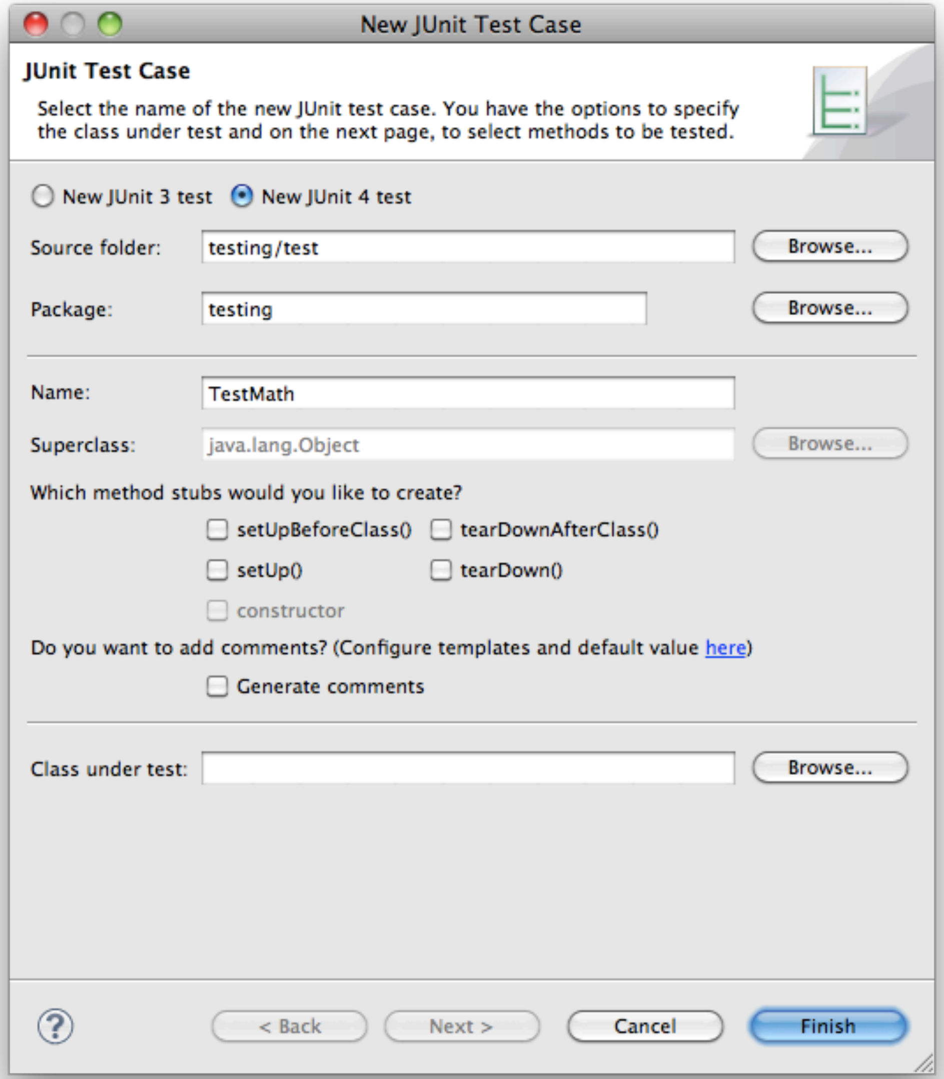
Project name: testing

Folder name: test

☐ Update exclusion filters in other source folders to solve nesting

Creating a JUnit Test

- To create a new JUnit test in Eclipse, first select the test folder, then…
- File → New → JUnit Test Case



JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test [✓ New JUnit 4 test]

Source folder: testing/test

Package: testing

Name: TestMath

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()
☐ setUp() ☐ tearDown()
☐ constructor

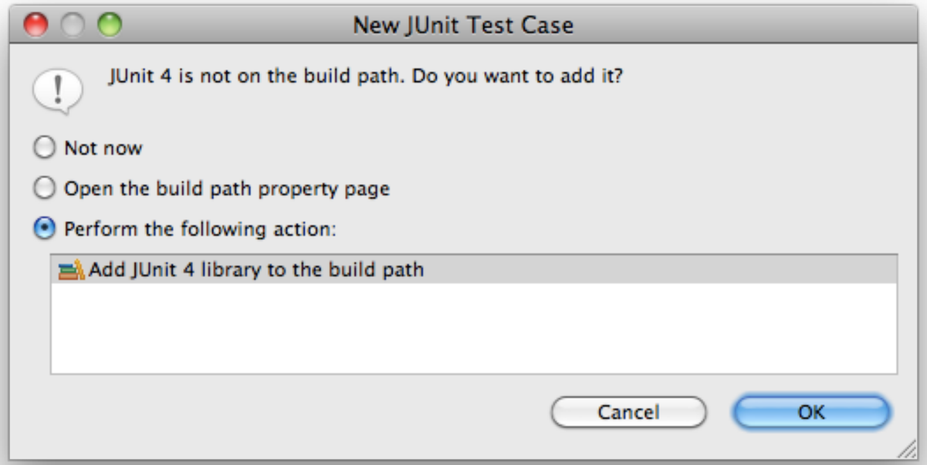
Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test:

Adding JUnit to the Classpath

- If you're adding your first test case, Eclipse will automatically allow you to add the library to the classpath…



New JUnit Test Case

JUnit 4 is not on the build path. Do you want to add it?

- ( ) Not now
- ( ) Open the build path property page
- [✓ Perform the following action:]

Add JUnit 4 library to the build path

@Test Annotation

- Test methods are identified by the @Test annotation before the method declaration

```
@Test
public void someTestMethod() { }
```

- This tells JUnit that the method should be executed as a test case
- To use @Test, you'll need to import Test…

```
import org.junit.Test;
```

JUnit Assert Class

- The Assert class is the primary mechanism for identifying success/failures in JUnit
- It provides many static methods that are used to test various conditions
- To utilize the class, you'll need to import…

```
import org.junit.Assert;
```

JUnit Assert Class Methods

- Methods for verifying trueness/falseness…

```
public static void assertTrue(boolean condition);
public static void assertFalse(boolean condition);
```

- Methods for testing nullness/non-nullness…

```
public static void assertNotNull(Object object);
public static void assertNull(Object object);
```

JUnit Assert Class Methods

- Checking objects, integer types (byte, char, int, long) and floating point types (float, double) for equality…

```
public static void assertEquals(Object expected,
                                Object actual);
public static void assertEquals(integer_types expected,
                                integer_types actual);
public static void assertEquals(float_types expected,
                                float_types actual,
                                float_types delta);
```

JUnit Assert Class Methods

- Methods for comparing arrays of elements for equality…

```
public static void assertArrayEquals(integer_type[] expecteds,
                                     integer_type[] actuals);
public static void assertArrayEquals(Object[] expecteds,
                                     Object[] actuals);
```

- See the javadocs for a complete listing…
  - http://junit.sourceforge.net/javadoc/org/junit/Assert.html

Testing Java's Math Class

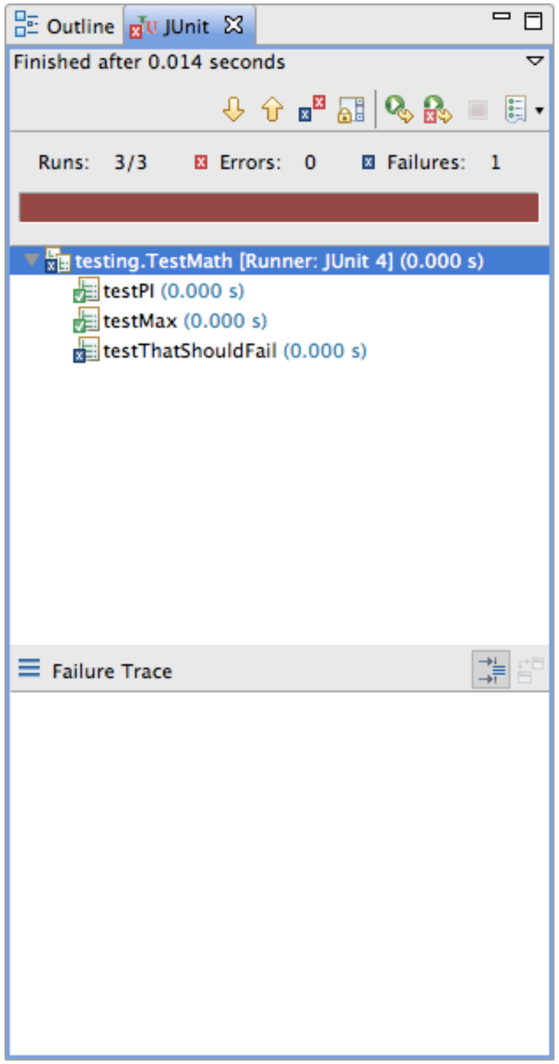
```java
package testing;

import org.junit.Assert;
import org.junit.Test;

public class TestMath {
    @Test
    public void testPI() {
        Assert.assertEquals(3.1415, Math.PI, .0001);
    }
    @Test
    public void testMax() {
        Assert.assertEquals(16, Math.max(5, 16));
    }
    @Test
    public void testThatShouldFail() {
        Assert.assertEquals(16, Math.min(5, 16));
    }
}
```

Running a Test

- To run a test in Eclipse simply right click and select Run As → JUnit Test
- Note that your class doesn't have a main, this is actually okay, as JUnit is doing some magic behind the scenes…
  - Basically it's finding all methods that have a @Test annotation and runs each of them independently and records the Assert result

The JUnit View

- The JUnit view shows all of the tests as executed by JUnit
  - Errors indicate something went wrong during the test (e.g. exception)
  - Failures are a result of a failing assert statement
- Errors and/or failures indicate an issues with the test or a problem (bug) in the code



Finished after 0.014 seconds

Runs: 3/3   Errors: 0   Failures: 1

- testing.TestMath [Runner: JUnit 4] (0.000 s)
  - testPI (0.000 s)
  - testMax (0.000 s)
  - testThatShouldFail (0.000 s)

Properties of Good Unit Tests

- Now that we know how to write a basic test, what are things we aim for in good tests?
  - Repeatable
    - Should be able to be re-run producing the same results (avoid randomness, getting current time, etc.)
  - Independent
    - Only test one feature (method) at a time (per JUnit test method)
    - Tests should not be dependent upon one another
  - Provide Value
    - Testing simple getters/setters is probably not a good use of time
  - Thorough
    - Tests all class invariants, pre/post conditions, edge cases

Thoroughness

- In order for your tests to be thorough, you need to check for several things…
  - General Correctness
  - Boundary Conditions
  - Error Conditions

General Correctness

- These are the so-called easy tests to write
- These test the “general” cases

Boundary Conditions

- Ordering
  - Does various ordering affect the outcome?
- Range
  - zero, minimum, maximum, positive #s, negative #s
- Existence
  - Null values for reference parameters?
  - Empty things…
    - Collections (e.g. Arrays)
    - Strings
- Cardinality
  - Expected number of items?

Error Conditions

- Are the right exceptions getting raised under the right conditions?
- I/O issues…
  – Missing files
  – Unreadable files
  – Empty files

Testing for Exceptions in JUnit

- You can create a test that checks that an exception is thrown by modifying the @Test attribute as so…

```
@Test(expected=SomeException.class)
public void testThatRaisesException() {
    // foo should throw an exception if arg is negative
    SomeObject.foo(-1);
}
```

Running a Suite of Tests

- Ideally you'd have test classes corresponding to most (if not all) of your classes
- Rather than running each test separately you can run a whole suite of tests like so…

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
   TestFoo.class,
   TestBar.class,
   TestBaz.class
})
public class RunAllTests { }
```

Exercise

- Identify test cases for the following method…

```
public static int largest(int[] list) { }
```

- What tests might we have for each of the following areas?
  - General Correctness
  - Boundary Conditions
  - Error Conditions

A Buggy Implementation

- How many of your tests failed on the following buggy implementation of largest?

```
public static int largest(int[] list) {
    int max = Integer.MAX_VALUE;
    for(int i = 0; i < list.length - 1; i++) {
        if(list[i] > max) {
            max = list[i];
        }
    }
    return max;
}
```

General Correctness

```
@Test
public void testLargestInMiddle() {
    int[] array = new int[] {1, 2, 5, 3, 4};
    Assert.assertEquals(5, Statistics.largest(array));
}
```

Ordering

```
@Test
public void testLargestAtBack() {
    int[] array = new int[] {1, 2, 3, 4, 5};
    Assert.assertEquals(5, Statistics.largest(array));
}

@Test
public void testLargestAtFront() {
    int[] array = new int[] {5, 4, 3, 2, 1};
    Assert.assertEquals(5, Statistics.largest(array));
}
```

Range

```
@Test
public void testLargestNegativeNumbers() {
    int[] array = new int[] {-1, -2, -3, -4, -5};
    Assert.assertEquals(-1, Statistics.largest(array));
}

@Test
void testLargestAcrossZero() {
    int[] array = new int[] {-2, 2, 0, -1, 1};
    Assert.assertEquals(2, Statistics.largest(array));
}

@Test
void testLargestBigNumbers() {
    int[] array = new int[] { Integer.MAX_VALUE - 2,
            Integer.MIN_VALUE, Integer.MAX_VALUE, 0 };
    Assert.assertEquals(Integer.MAX_VALUE, Statistics.largest(array));
}
```

Existence/Error Conditions

```java
@Test(expected=IllegalArgumentException.class)
public void testNullList() {
   int[] array = null;
   Assert.assertEquals(-1, Statistics.largest(array));
}

@Test(expected=IllegalArgumentException.class)
public void testEmptyList() {
   int[] array = new int[] { };
   Assert.assertEquals(-1, Statistics.largest(array));
}
```

A Much Improved largest Method

```java
public static int largest(int[] list) {
    if(list == null) {
        throw new IllegalArgumentException("list cannot be null");
    } else if (list.length == 0) {
        throw new IllegalArgumentException("list cannot be empty");
    }

    int max = Integer.MIN_VALUE;
    for(int i = 0; i < list.length; i++) {
        if(list[i] > max) {
            max = list[i];
        }
    }
    return max;
}
```

Additional Resources

- Pragmatic Unit Testing in Java with JUnit
  - Free Introduction chapter
  - Free testing Summary cheat-sheet
- JUnit Test Infected: Programmers Love Writing Tests