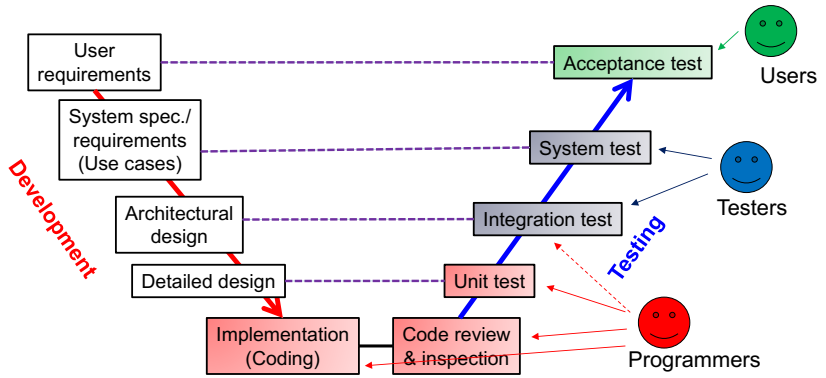
Unit Testing

Unit Tests

- Verify that each *program unit* works as it is intended and expected along with the system specification.
  - Units to be tested: classes (methods in each class) in OOPLs



Who Does it?

- You!
  - as a programmer
- Test cases are written *as programs* from a programmer's perspective.
  - A test case describes a test to verify a tested class in accordance with the system specification.
- Programmers and unit testers are no longer separated in most (both large-scale and small-scale) projects as
  - Useful tools has made unit testing a lot easier and less time-consuming.
  - Programmers can write the best test cases for their own code in the least amount of efforts.

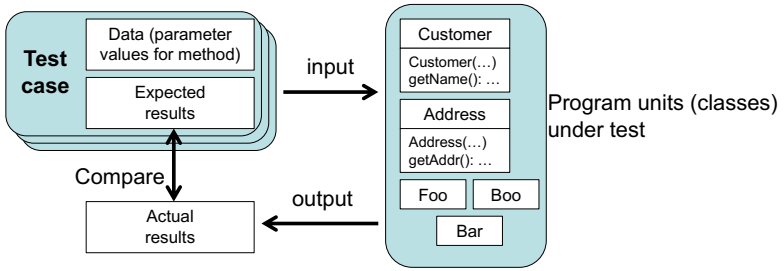
What to Do in Unit Testing?

- 4 tests (test types)
  - We will focus on 3 of them: *functional*, *structural* and *confirmation* tests.

	Functional test	Non-functional test	Structural test	Confirmation test
Acceptance test				
System test				
Integration test				
Unit test	**X (B-box)**		**X (W-box)**	**X (Reg test)**
Code rev&insp.				

Functional Test in Unit Testing

– Ensure that each method of a tested class successfully performs a set of specific tasks.
  - Each test case confirms that a method produces the expected output when given a known input.
    – Black-box test
  - Well-known techniques: equivalence test, boundary value test



Structural Test in Unit Testing

- Verify the structure of each class.
- Revise the structure, if necessary, to improve maintainability, flexibility and extensibility.
  – White-box test

- To-dos
  – Refactoring
  – Use of design pattern
  – Control flow test
  – Data flow test

Confirmation Test in Unit Testing

- Re-testing
- Regression testing

Suggested Reading

- Nick Jenkins, *Software Testing Primer*.
  – http://www.nickjenkins.net/
  – http://bit.ly/2xuofin

Unit Testing with JUnit

JUnit

- A unit testing framework for Java
  - Defines the format of each test case
    - Test case: a program to verify a method(s) of a given class with a set of inputs/conditions and expected results.
  - Provides APIs to write and run test cases
  - Reports test results
  - Makes testing as easy and automatic as possible.
- Version 4.x, http://junit.org/junit4/
- Integration with Ant and Eclipse (and other IDEs)
  - <junit> and <junitreport> for Ant

Test Classes and Test Methods

- Test class
  - A public class that has a set of "test methods"
  - Common naming convention: XYZTest
    - XYZ is a class under test.
  - One test class for one class under test
- Test method
  - A public method in a test class.
    - No parameters
    - No values returned ("void")
    - Can have a "throws" clause
  - Annotated with @Test
    - org.junit.Test
  - One test method implements one test case.

Assertions

- Each test method verifies one or more *assertions*.
  - An assertion is a statement that a predicate (boolean function/expression) is expected to be always true at a particular point in code.
    - `String line = reader.readLine();`
      Assertion: `line != null`
    - `String str = foo.getPassword();`
      Assertion: `str.length() > 6`
- In JUnit, running unit tests means verifying assertions described in test methods.

An Example

- Class under test

```
public class Calculator{
 public float multiply(float x,
                       float y){
   return x * y;
 }
 public float divide(float x,
                     float y){
   if(y==0){ throw
    new IllegalArgumentException(
     "division by zero");}
   return x/y;
 }
]
```

- Test class

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

public class CalculatorTest{
 @Test
 public void multiply3By4(){
   Calculator cut = new Calculator();
   float actual = cut.multiply(3,4);
   float expected = 12;
   assertThat(actual, is(expected); }

 @Test
 public void divide3By2(){
   Calculator cut = new Calculator();
   float actual = cut.divide(3,2);
   float expected = 1.5f;
   assertThat(actual, is(expected)); }

@Test(expected=illegalArgumentException.class)
 public void divide5By0(){
   Calculator cut = new Calculator();
   cut.divide(5,0); }
}
```


Key APIs

- `org.junit.Assert`
  - Used to define an assentation and verify if it holds
- `org.hamcrest.CoreMatchers`
  - Provides a series of *matchers*, each of which performs a particular matching logic.

Key Annotations

- @Test
  - org.junit.Test
  - JUnit runs test methods that are annotated with @Test.
- @Ignore
  - org.junit.Ignore
  - JUnit ignores test methods that are annotated with @Ignore
    - No need to comment out the entire test method.

Static Imports

- `Assert` and `CoreMatchers` are typically referenced through *static import*.
  - `import static org.junit.Assert.*;`
    `import static org.hamcrest.CoreMatchers.*;`
  - With static import
    - `assertThat(actual, is(expected);`
      - asserting that "actual" is equal to "expected"
      - `assertThat()` is a static method of `Assert`.
      - `is()` is a static method of `CoreMatcher`.
        - Performs a particular matching logic.
  - With normal import
    - `Assert.assertThat(actual, CoreMatchers.is(expected));`

JUnit and Hamcrest

- Hamcrest provides many useful matchers (incl. `CoreMatchers.is()`) for JUnit
  - http://hamcrest.org/JavaHamcrest/
  - junit.jar and hamcrest-core.jar available from http://junit.org
    - Both are available in Eclipse (and other IDEs) by default.

Principles in Unit Testing

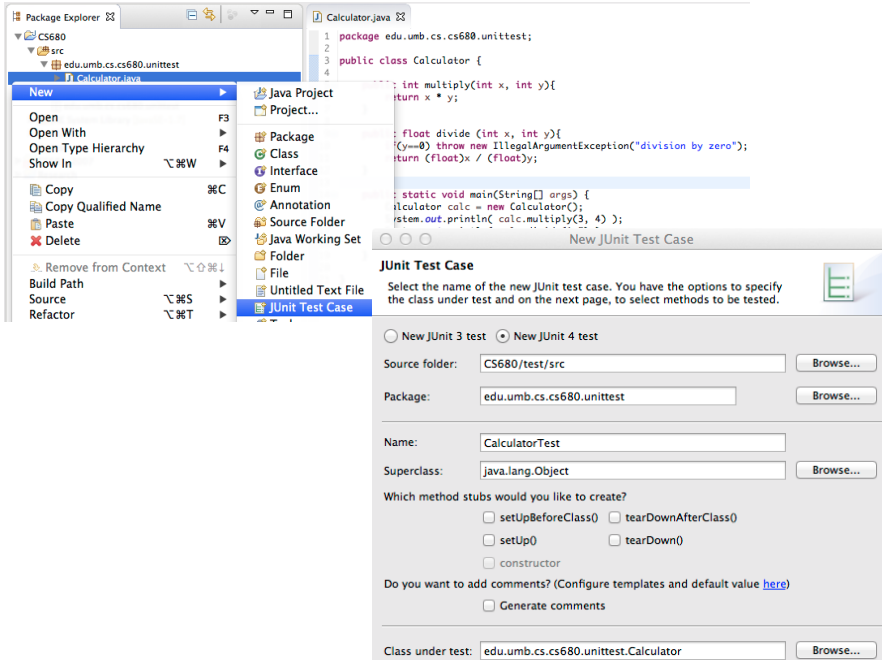
- Define one or more *fine-grained specific* test cases (test methods) for each method in a class under test.
- Give a *concrete/specific* and *intuitive* name to each test method.
  - e.g. “divide5by4”
  - Avoid something like “testDivide”
- Use specific values and conditions, and detect design and coding errors.
  - Be detail-oriented. The devil resides in the details!
- No need to worry about redundancy in/among test methods.

- Write simple, short, easy to understand test cases
  - Try to write many simple test cases, rather than a fewer number of complicated test cases.
    - Make it clear what is being tested for everyone.
    - Avoid a test case that perform multiple tasks.
    - You won’t feel bothered/overwhelmed by the number of test cases as far as they have intuitive names.
      - e.g. “divide5by4”

Test Suite with JUnit

- A set of test classes
  - ~/code/projectX/ [project directory]
    - build.xml
    - src [source code directory]
      - edu/umb/cs/cs680/hw01/Foo.java
      - edu/umb/cs/cs680/hw02/Boo.java
    - bin [byte code directory]
      - edu/umb/cs/cs680/hw01/Foo.class
      - edu/umb/cs/cs680/hw02/Boo.class
    - test [a test suite; a set of test classes]
      - src
        - » edu/umb/cs/cs680/hw01/FooTest.java
        - » edu/umb/cs/cs680/hw02/BooTest.java
      - bin
        - » edu/umb/cs/cs680/hw01/FooTest.class
        - » edu/umb/cs/cs680/hw02/BooTest.class

- Do not mix up test code and production code (e.g. tested code)
- Production code should not include tests in many projects.



Things to Test

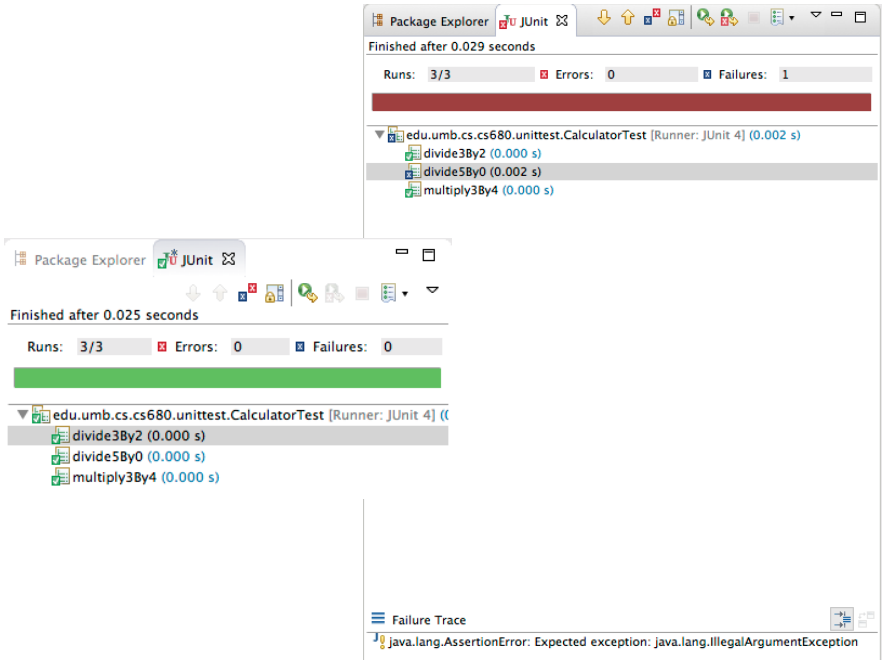
- Methods
- Exceptions
- Constructors

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

public class StudentTest{
  @Test
  public void constructorWithName(){
    Student cut = new Student("John");
    assertThat(cut.getName(), is("John"));
    assertThat(cut.getAge(), is(nullValue()));
    assertThat(cut.getEmailAddr(), is(nullValue()));
  }
 @Test
  public void constructorWithoutName(){
    Student cut = new Student();
    ...
}
```


Test Runners

- How to run JUnit?
  - From command line
    - `java org.junit.runner.JUnitCore edu.umb.cs.cs680.CalculatorTest`
    - `java org.junit.runner.JUnitCore edu.umb.cs.cs680.FooTest, edu.umb.cs.cs680.BooTest`
  - From IDEs
    - Eclipse, etc.
  - From Ant
    - <junit> task
- How to run unit tests?
  - Test runners
    - org.junit.runners.JUnit4 (default runner)



HW 1 – Step 1

- Implement `Calculator`
  - Package: `edu.umb.cs.cs680.hw01`
- Follow the directory structure shown in Slide 20.
  - `<proj dir>/src/edu/umb/cs/cs680/hw01/Calculator.java`
  - `<proj dir>/bin/edu/umb/cs/cs680/hw01/Calculator.class`
- Use Ant to build and run `Calculator`
  - Set up the directory where `Calculator.class` is placed.
    - `<proj dir>/bin/edu/umb/cs/cs680/hw01`
  - Set up CLASSPATH
    - `<proj dir>/bin`
  - Compile `Calculator.java` and generate `Calculator.class` to `<proj dir>/bin/edu/.../hw01`
  - Run `Calculator.class`

Automatic Build

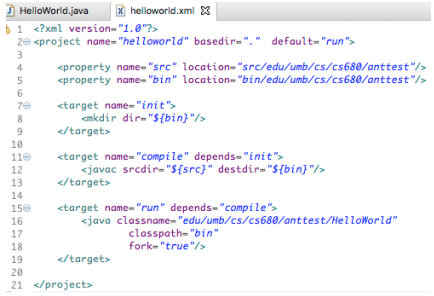
- Use Ant (http://ant.apache.org/) to build all of your Java programs in every coding HW.
  - Learn how to use it, if you don't know that.
  - Turn in *.java and a build script (e.g. build.xml).
    - Turn in a ***single*** build script (build.xml) that
      - configures all settings (e.g., class paths, a directory of source code, a directory to generate binary code),
      - compiles all source code from scratch,
      - generates binary code (*.class files), and
      - runs compiled code
    - DO NOT include absolute paths in a build script.
      - You can assume my OS configures a right Java API (JDK/JRE) Jar file (in its env setting).
    - DO NOT turn in byte code (class files).
    - DO NOT use any other ways for configurations and compilation.
      - Setting paths manually with a GUI (e.g., Eclipse)
      - Setting an output directory manually with a GUI
      - Clicking the "compile" button manually

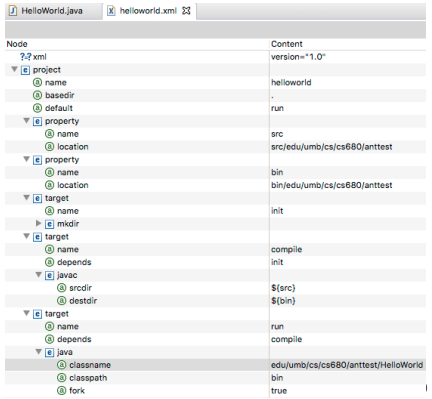
  - I will simply type "ant" (on my shell) in the directory where your build.xml is located and see how your code works.
    - You can name your build script as you like.
      - No need to name it build.xml.
      - I will type: ant -f abc.xml
    - If the "ant" command fails, I will NOT grade your HW code.
  - Fully automate configuration and compilation process to
    - speed up your configuration/compilation process.
    - remove potential human-made errors in your configuration/compilation process.
    - Make it easier for other people (e.g., code reviewers, team mates) to understand your code/project.

- Use the ANT_HOME and PATH environment variables to specify the location of the "ant" command (i.e., ant.sh or ant.bat)
  - ANT_HOME
    - Reference the top directory of an Ant distribution
      - e.g. Set ~/code/ant/apache-ant-1.9.7 to ANT_HOME
  - PATH
    - Reference the location of "the ant" command
      - e.g., Set ${ANT_HOME}/bin to PATH
  - c.f. http://ant.apache.org/manual/
- You can assume my machine/OS configures ANT_HOME and ANT properly.
  - This way, your build.xml can run on my machine/OS.

Ant in Eclipse

- You can use Ant that is available in your IDE (e.g. Eclipse).
  - However, I will run your build script on a shell.
  - Make sure that your build script works on a shell.





HW 1 – Step 2

- Implement **`CalculatorTest`**
  - Package: **`edu.umb.cs.cs680.hw01`**
  - Define extra test methods in addition to **`multiply3By4()`**, **`divide3By2()`**, **`divide5By0()`**.
    - e.g., A float number times a float number
      - **`Multiple2_5By5_5()`**
    - e.g., A float number over a float number
      - **`Multiple2_5By5_5()`**
- Follow the directory structure shown in Slide 20.
  - <proj dir>/**src**/edu/umb/cs/cs680/hw01/Calculator.java
  - <proj dir>/**bin**/edu/umb/cs/cs680/hw01/Calculator.class
  - <proj dir>/**test**/src/edu/umb/cs/cs680/hw01/CalculatorTest.java
  - <proj dir>/**test**/bin/edu/umb/cs/cs680/hw01/CalculatorTest.class

- Use Ant to build and run **`Calculator`** and **`CalculatorTest`**
  - Set up the directory where **`Calculator.Class`** is placed.
    - **`<proj dir>/bin/edu/umb/cs/cs680/hw01`**
  - Set up the directory where **`CalculatorTest.class is placed.`**
    - **`<proj dir>/test/bin/edu/umb/cs/cs680/hw01`**
  - Set up CLASSPATH
    - **`<proj dir>/bin`**
    - **`<proj dir>/test/bin`**
    - Junit.jar and hamcrest-core.jar
  - Compile **`Calculator.java`** and generate **`Calculator.class`** to **`<proj dir>/bin/edu/.../hw01`**
  - Compile **`CalculatorTest.java`** and generate **`CalculatorTest.class`** to **`<proj dir>/test/bin/edu.../hw01`**
  - Run **`CalculatorTest.class`** with JUnit
  - Run **`Calculator.class`**

- Run JUnit from Ant. Use <junit> task in Ant.
  - c.f. JUnit documentations (API docs, user manual, etc.)
- No need to save test results in files. Just print them out on a shell
  - e.g., <formatter type="plain" usefile="false" /> in <junit>
- Keep your build.xml *platform independent*.
  - Set the location of junit.jar to the environment variable JUNIT
    - e.g., JUNIT → ~/.p2/pool/plugins/org.junit_4.xxx/junit.jar
    - Use a relative path.
  - Set the location of hamcrest-core.jar to the environment variable HAMCREST_CORE
    - e.g., HAMCREST_CORE → ~/.p2/pool/plugins/org.hamcrest.core_1.xxxx.jar
    - Use a relative path.

- Reference the two environment variables to configure CLASSPATH in your build script.
  - <property environment="env"/>
    <path id="test.classpath">
      …...
      <pathelement path="${env.JUNIT}" />
      <pathelement path="${env.HAMCREST_CORE}" />
    </path>

    <javac …>
      ...
      <classpath refid="test.classpath"/>
      …
    </javac>

    <junit …>
      ...
      <classpath refid="test.classpath"/>
      …
    </junit>

HW Submission

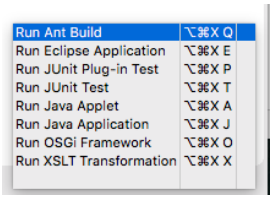
- Submit me an archive file (.rar, .tar.gz, .7z, etc.) that contains
  - build.xml
  - "src" sub-directory
  - "test/src" sub-directory
  - **DO NOT send me binary files (.class and .jar files)**
  - Avoid a .zip file
  - Send the archive file to **umasscs680@gmail.com** from **your "primary" email address**
  - Or, place it somewhere online (e.g. at G Drive) and email me a link to it.
- Deadline: Feb 27 midnight

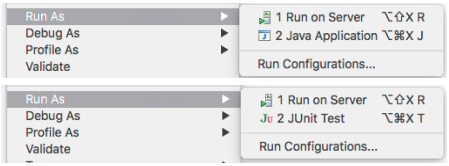
Suggestions

- Start as early as possible.
- Go step by step
  - If you are not familiar with Ant…
    - compile and run your code for Step 1 without using Ant first and then use Ant to build the code
  - Step 1 and then Step 2
- Expect "Death by XML"
  - You may need to spend a few hours, a half day or even a full day to make your build.xml run correctly.
  - That's a part of your learning curve...

Useful Keyboard Shortcuts in Eclipse

- Run as a Java App
  - Alt + Cmd + X, (and then) J
- Run as a JUnit Test
  - Alt + Cmd + X, (and then) T
- Run as an Ant script
  - Alt + Cmd + X, (and then) Q





Just in case…

- Run
  - Cmd + Shift + F11
  - Runs the current (open) Java class if it has main().
    - If not, runs the last launched class.
  - Starts JUnit and run test cases if a test class is open.
- Debug
  - Cmd + F11
- Quick fix
  - Cmd + 1
- Code assist
  - Ctrl + Space
- Organize import statements
  - Ctrl + Shift + O

Useful Eclipse Plugins

- Quick JUnit
  - https://marketplace.eclipse.org/content/quick-junit
  - https://github.com/kompiro/quick-junit
  - Cmd/Ctrl + 9
    - Move from a tested class (XYZ) to its test class (XYZTest), and vice versa.
      - If a test class is not defined yet, pop up a wizard to do so.
  - Cmd/Ctrl + 0
    - Run a test class with JUnit
    - Easier to type/remember than Shift+Cmd+X → T
    - Run a test method with JUnit if a cursor is placed in the test method.
  - Cmd/Ctrl + Shift + 0
    - Run a test class with the debugger

- JUnit Helper
  - https://marketplace.eclipse.org/content/junit-helper
  - https://github.com/seratch/junithelper
  - ALT + 9
    - Generates a test class for a tested class based on a template.
      - static imports, a test class and test methods with template-based bodies

```
public class SavingsAccount{
 public SavingsAccount(
              float balance){...}
 public float computeTax(){...}
}
```

```
public void computeTax_(){
  float balance = 0.0F;
  SavingsAccount target =
       new SavingsAccount(balance);
  float actual = target.computeTax();
  float expected = 0.0F;
  assertThat(actual,
            is(equalTo(expected)));}
```