

Testing

JUnit

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Monday, Mar. 6, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

1 Blackbox Practice Problems

Homework Assignment

- 1 Programming assignment 2 is posted on your lab Canvas page.

Recall

- ▶ The goal of testing is to find failures
- ▶ A failure is when the observed does not match the expected
- ▶ A fault is the algorithmic or mechanical cause of the failure
- ▶ A successful test finds a failure
- ▶ A failure lets us know a fault exists
 - ▶ It does not tell us the exact location
 - ▶ We still have to find it and fix it
- ▶ Testing can prove that faults exist, but no amount of testing will prove that faults do not exist

Primitive Testing

- ▶ Write `main` as a *command interpreter* with console input/output, so user (tester) provides inputs and observes actual results
- ▶ Tester compares actual results with allowed/expected results by *inspection*
- ▶ **Pros:**
 - ▶ Simple
 - ▶ Easy
 - ▶ Intuitive
- ▶ **Cons:**
 - ▶ Tedious
 - ▶ Error-Prone
 - ▶ Not Automated
- ▶ We should **NOT** do this.

Some Automated Testing

- ▶ Write `main` to contain sets of inputs and expected results in “parallel arrays” of argument values and expected results
 - ▶ **E.g.**, input array: `[1, 2, 3]` and expected result array: `[1, 4, 9]`, if testing a `square` method
- ▶ Loop in `main` compares actual results with allowed/expected results
- ▶ **Pros:**
 - ▶ Better, primarily because the process is now far more automatic.

Remaining Problems

- ▶ Can you store the inputs and outputs in an array?
- ▶ Classes make this difficult
 - ▶ Multiple method calls to set up input space
 - ▶ Multiple methods to call to check output
- ▶ What if there are multiple possible outputs?
 - ▶ A 2D array of outputs?
- ▶ This method doesn't scale with complexity

Remaining Problems

- ▶ Another drawback of this approach is that, if there are multiple allowed results for the given arguments, mere equality checking with actual results *does not work*
 - ▶ Recall the `aFactor` method; what happens if we write in the `results` array that the “expected” result is 6, when any of 1, 2, 3, or 6 (and maybe other results) are *also* allowed?

Serious Testing: JUnit

- ▶ **JUnit** is an industry-standard “framework” for testing Java code
 - ▶ A **framework** is one or more components with “holes” in them, i.e., some missing code
 - ▶ Programmer writes classes following particular conventions to fill in the missing code
 - ▶ Result of combining the framework code with the programmer’s code is a complete product
- ▶ For this course, we are going to use **JUnit 4**

Generic Example

- ▶ Suppose you are testing a method `M` in a class `C`

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

public class CMTest {

    @Test
    public void test1327M() {
        ...
    }

    ...
}
```

- ▶ A **static import** allows you to call `static` methods in `org.junit.Assert` without qualifying names (see, e.g., `assertEquals` in upcoming code).
- ▶ A **test plan** or **test fixture** is a `public class` with one method per test case.

Generic Example

Test Case

```
@Test
public void test1327M() {
    ...
}
```

- ▶ Each test case is a **public void** method with no parameters
 - ▶ JUnit will run this code, not us!
 - ▶ JUnit won't pass parameters or look at returned values
- ▶ Each test case has an **@Test** annotation just before it.
- ▶ There is an easy way to make a new test case:
 - 1 copy/paste from an existing test case
 - 2 edit slightly

Vocabulary Review

- ▶ **Test Case:**

- ▶ Exercises a single unit of code, normally a method (and a test case normally makes one call to that method)
- ▶ Test cases should be *small* (i.e., should test one thing)
- ▶ Test cases should be *independent* of each other
- ▶ Has input and expected output
- ▶ **In JUnit:** a `public` method that is annotated with `@Test`

- ▶ **Test Fixture/Plan:**

- ▶ Exercises a single class
- ▶ Is a collection of *test cases*
- ▶ **In JUnit:** a class that contains `@Test` methods

New Vocabulary

- ▶ **(JUnit) Assertion:**

- ▶ A claim that some `boolean`-valued expression is `true`; normally, a comparison between expected and actual results (i.e., the `equals` method says they are equivalent)

- ▶ **Passing a Test Case:**

- ▶ All JUnit assertions in the test case are `true` when the test case is executed (and no error occurred to stop program execution)

- ▶ **Failing a Test Case:**

- ▶ Some JUnit assertion in the test case is `false` when the test case is executed

Execution Model

- ▶ Separate instances (objects) are created from the JUnit test fixture
 - ▶ JUnit creates one instance per test case (!)
- ▶ **Implication:**
 - ▶ Do not rely on order of test cases
 - ▶ Test case listed first in JUnit test fixture is not guaranteed to be executed first

JUnit Assertions

- ▶ Two most useful `static` methods in `org.junit.Assert` to check actual results against allowed results:
 - 1 `assertEquals(Object expected, Object actual);`
 - 2 `assertTrue(boolean expression);`
- ▶ There is rarely a reason to use any of the dozens of other assertion static methods in `org.junit.Assert`

JUnit Assertions

assertEquals

```
assertEquals(Object expected, Object actual);
```

- ▶ **Recall:** Every class extends the `Object` class
 - ▶ So this can take in any non-primitive data type
- ▶ How can it check if they are equal without knowing the data type?
 - ▶ Use the `equals` method!

JUnit Assertions

`assertEquals` - doubles

- ▶ In JUnit 4, `assertEquals(double expected, double actual)` has been replaced with:

`assertEquals(double expected, double actual, double epsilon)`

- ▶ We can use `epsilon` to define an acceptable error
 - ▶ `assertEqual` will return `true` as long as $\text{abs}(\text{actual} - \text{expected}) < \text{epsilon}$
 - ▶ Use a constant for `epsilon`
- ▶ This handles the fact that we lose precision with `double` values at very small decimal places.
- ▶ We could expect a method to return 2.0 and it actually would return 2.000000000000001, which would be marked as a failed test case.

Comparing Objects

- ▶ Testing with objects can be difficult
- ▶ Want to be able to see the state of the object
 - ▶ May not be publicly available
- ▶ Using `equals` can be problematic
 - ▶ The only way to create the expected object may be to **use the method you are trying to test** due to information hiding
 - ▶ The fault would affect both your expected output and your actual output!
- ▶ **A Solution:**
 - ▶ Use the `toString()` method of the object to convert it to a `String`, and create a `String` representation of the expected object
 - ▶ Compare the strings for equivalence

Timed Tests

- ▶ What if you're worried about an infinite loop?
 - ▶ Parameterize `@Test` with a **timeout**: number of milliseconds before the test case is terminated for running too long
 - ▶ **Ex:** `@Test(timeout=100)`
- ▶ **Problem:** How do you know what is long enough for a test case to run?
 - ▶ **Overestimate!**

Best Practices

- ▶ Keep JUnit test fixtures in the same project as the code, but in a separate source folder (for this course: regular code in “src”, test classes/fixtures in “test”)
 - ▶ Tests are then included when project is “built”
 - ▶ Helps keep test fixtures consistent with other code

Best Practices

- ▶ Name test fixtures consistently
 - ▶ **Ex:** class `CTest` (or `TestC`) tests class `C`
- ▶ Name test cases consistently
 - ▶ **Ex:** method `test_13_7_foo` (or `test_foo_13_7`) tests method `foo` with inputs 13 and 7
- ▶ Examples:
 - ▶ **Bad:**¹ `test_mortgage_57`
 - ▶ **Good:**
`test_loanApproved_lowRate_highDownPay_HighDtoI`

¹On Unix, when a test case fails, we only get the name of the method!

Best Practices

- ▶ Try to only test one method per test case
- ▶ If a test case fails, it does not give us any information about which assert statement failed or why.
 - ▶ We are just given the name of the test case that failed
 - ▶ If we have multiple method calls, the fault could exist in any method
- ▶ **Problem:**
 - ▶ Often we have to call multiple methods
 - ▶ Need to call the constructor, or methods to set private data members
- ▶ **Solution:**
 - ▶ Create test cases that also call those methods individually
 - ▶ If the constructor test case fails, then fix that before you worry about the others
 - ▶ Some even separate each method into it's own JUnit code file and run them in that order

Magic Numbers

- ▶ Often times we will use “Magic Numbers” in our test cases
 - ▶ Hard code input and output
 - ▶ But “Magic Numbers” are bad!
- ▶ Couldn't I have the method that calls my method take in input?
 - ▶ Then call it with different input?
 - ▶ Avoids “Magic Numbers” and a lot of copying and pasting
 - ▶ **Don't do this!**
- ▶ **Recall:** When a JUnit test case fails, all we get is the name of the method
 - ▶ We wouldn't get the input that caused it to fail!

Remember

- ▶ Our JUnit code is not production code
- ▶ It is a framework to help us automate testing
- ▶ We don't have the same best practices as we do with our normal code
 - ▶ So we use “Magic Numbers”
 - ▶ We don't factor out common functionality

Recommended Test Case Style

```
@Test
public void test_13_27_myFunction() {
    // Declare and initialize input variables
    int input1 = 13;
    int input2 = 27;

    // Declare and initialize expected output
    int expected = 37;

    // Declare Object and set up space
    MyClass myVar = new MyClass();
    ...

    // run method and get output
    int actualOutput = myVar.myFunction(input1, input2);

    //Compare expected and actual
    assertEquals(expected, actualOutput);
}
```

Recommended Test Case Style

```
@Test
public void test_140_anotherM() {
    /*
     * Set up variables and call method under test
     */
    ...

    /*
     * Assert that values of variables match expectations
     * Random code below, just for an example
     */
    assertEquals(nExpected, n);
    assertEquals(7, k);
}
```

Note: Sometimes, you can write the expected value directly.

- ▶ JUnit is a framework that helps us automate unit testing
 - ▶ We can test our code after any and all changes
 - ▶ Keep the bar green!
 - ▶ Don't put any errors into the system
- ▶ Takes a lot of work to set it up, but saves us the work of manually running all the test cases

JUnit on School of Computing Unix Machines

Running JUnit on School of Computing Unix Machines

- ▶ The different versions of JUnit are wildly different.
- ▶ We are using JUnit 4 in this course
 - ▶ The School of Computing's Unix machines have JUnit 3 and 4 installed on them
 - ▶ JUnit 4 code is much easier, and will not run on the JUnit 3 installation
 - ▶ Make sure to install JUnit 4, not JUnit 5 on your machine
 - ▶ That way you can easily transfer code
- ▶ The graders will use JUnit 4 on the school machines, so it is important to use the right version

Running JUnit on School of Computing Unix Machines

- ▶ Put your JUnit test file in the same package/directory as your Java code
 - ▶ This is different from IntelliJ, as IntelliJ mirrors the package file structure in “src” and “test”, making it easier to remove test cases
 - ▶ We could set up a whole “test” directory, but for simple programs like ours this is fine.
- ▶ You can still compile your code without JUnit and run it normally, just don't include the JUnit code file
 - ▶ **Example:** Lab Code
 - ▶ `javac cpsc2150/banking/MortgageApp.java`
`cpsc2150/banking/models/Customer.java`
`cpsc2150/banking/models/Mortgage.java ...`
 - ▶ We could now run it with `java MortgageApp` as we usually would

Running JUnit on School of Computing Unix Machines

- ▶ If you want to run your JUnit code, you need to include the JUnit code file you wrote, and the JUnit jar file
 - ▶ Add the JUnit jar file to your `classpath`
 - ▶ A jar file is a Java Archive file
 - ▶ Like a zip file, but filled with compiled Java code and meta data
- ▶ Classpath?
 - ▶ One of the arguments to our compiler
 - ▶ We can add other libraries and folders to our `classpath`
 - ▶ Java will search for libraries that we import by looking in the folders specified in the `classpath`.
 - ▶ We haven't needed to do this because we've only imported standard Java libraries that are included with the compiler

Running JUnit on School of Computing Unix Machines

- ▶ Classpath?

- ▶ `-cp <path>`

- ▶ Our JUnit path is: `./usr/share/java/junit4.jar`

- ▶ This is only the path for our school computers. If you install it yourself, it's a different directory.

- ▶ `javac -cp ./usr/share/java/junit4.jar
cpsc2150/banking/models/TestMortgage.java
cpsc2150/banking/models/Mortgage.java ...`

- ▶ **Note:** Don't copy and paste!

- ▶ It is possible that symbols are encoded differently than Unix.
(e.g. -)

- ▶ Instead type it out manually.

- ▶ Make sure you don't forget the `“./”` before the path to JUnit.

Running JUnit on School of Computing Unix Machines

- ▶ Now we can run our JUnit tests on SoC Unix
- ▶ We need to use the `classpath` again and include the JUnit executable
 - ▶ `java -cp ./usr/share/java/junit4.jar org.junit.runner.JUnitCore cpsc2150.banking.models.TestMortgage`
 - ▶ Now instead of calling `MortgageApp` as the entry point, we are calling `JUnitCore` as our driver, and passing in our compiled `TestMortgage` as an argument
- ▶ **Note:**
 - ▶ There is a documented shortcut on the school machines that just involves calling “`junit`” as a command
 - ▶ It calls JUnit 3
 - ▶ There were major changes made from 3 to 4, with some important syntax differences
 - ▶ So our code won't work with JUnit 3

JUnit Testing and Makefiles

- ▶ We have commands to compile and run our JUnit tests on Unix
- ▶ Makefiles allow us to compile and run programs on Unix
- ▶ So we can add this functionality to our Makefiles
- ▶ Add two new targets:
 - 1 `test`:
 - ▶ Use the command to compile our JUnit code
 - 2 `runtest`:
 - ▶ Use the command to execute JUnit tests
- ▶ Now our testing can happen from our Makefiles as well!

Resources

- ▶ *JUnit in Action, Second Edition* (Petar Tahchiev, et. al., 2010)
 - ▶ <https://proquest.safaribooksonline.com/book/software-engineering-and-development/software-testing/9781935182023>
- ▶ *IntelliJ Testing*
 - ▶ <https://www.jetbrains.com/help/idea/testing.html>
 - ▶ Look over this to help you with this and next week's lab

Homework Assignment

- 1 Programming assignment 2 is posted on your lab Canvas page.

Summary

- 1 Serious Testing: JUnit
- 2 Generic Example
- 3 New Vocabulary
- 4 Execution Model
- 5 JUnit Assertions
- 6 Timed Tests
- 7 Best Practices
- 8 Running JUnit on School of Computing Unix Machines