CSC 335 – Testing & Documentation

Dr. Jonathan Misurda
jmisurda@cs.arizona.edu

Software Testing

- Testing can be done by execution or inspection
- Execution means a program is run with an input designed to exercise the program in a certain way
  - The input is called a **test case**
- Since a single run may not capture the full intended program behavior, multiple test cases can be assembled into a **test suite**

Testing Software

- How do you know when your code is correct?
  - It compiles?
  - It runs?
  - It produces a result?
  - That result is right?

Some Types of Testing

- **Unit Testing**
  - Test one module (e.g., Java class) at a time
  - Separate tests that each test a single aspect of the module
- **Integration Testing**
  - Test multiple modules together
  - Test full features
- **Alpha Testing**
  - Test feature-complete program within testing organization
- **Beta Testing**
  - Use outside testers

Software Testing

- Process of assessing the functionality of a program through execution or analysis
- Try to reveal software **faults**
  - A portion of a program's source code that is incorrect
- Before they manifest themselves as **failures**
  - The inability of a system or component to perform its required functions within specified performance requirements

IEEE Standard Glossary of Software Engineering Terminology

Unit Testing

- **Black Box Testing**
  - Tests the behavior of the program with input/output sets
  - Tests are written with the original specification of the behavior in mind, ignoring the details of the (eventual) implementation
  - Implementation may not even exist yet…
- **White Box Testing**
  - Uses the control structure of the program to design test cases
  - Must be written after the implementation

Goals of *Black Box Testing*

- Find incorrect or missing functions
- Find interface errors
- Find errors in data structures or external data base access
- Locate behavioral or performance errors
- Identify initialization and termination errors
- Detect off-by-one errors

Why Use JUnit?

- With a succinct group of tests for your code, you are able to run them all to make sure the changes you just made didn’t break something that was working previously
- This kind of testing is called **regression testing**

Goals of *White Box Testing*

1. Exercise all logical decisions on their true and false sides
2. Execute all loops to their boundaries
3. Guarantee all independent paths have been exercised
4. Exercise all internal data structures to, and beyond, their boundaries (much harder)

Assertions

- JUnit uses the notion of a runtime assertion
- An **assertion** is a statement that enables you to test your assumptions about your program.
- Each assertion contains a boolean expression that you believe will be true when the assertion executes.
  - If it is false, the system will throw an error.
  - By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors.

What is JUnit?

- JUnit is a tool for the unit testing of Java code
- http://www.junit.org/ is an open source testing framework
- JUnit provides an easy framework to quickly test all your code at once
  - Write a separate unit test as a class, in the same language as the production code (in our case, Java)
  - Works as an Eclipse plugin, meaning the Eclipse IDE runs your tests, and shows a side window telling whether your tests passed (Green Bar) or where they failed (Red Bar and stack trace)

Java’s Built-In assert

- A programmer does not actually need JUnit to create assertions
- Include assertions in your code using the following syntax:
  - `assert (boolean);`
- Ideally, all methods should use Java’s built-in assert to check all preconditions on entry to a method and all postconditions on exit from a method. In reality this is often hard to do, especially for postconditions.

How to Use Java's `assert`

- Assertions are disabled by default in Java, so when an assertion is false, nothing usually happens!
- You must enable assertions by running Java with the `-enableassertions` command line argument.
  - When an assertion is false an `AssertionError` exception will be thrown.
- `AssertionError` exceptions can be handled just like any other exception using try/catch blocks or throws declarations. If they are unhandled, they will halt your program's execution.

TDD Can Help Design

- TDD makes you think about what you want or need before you write the code
  - Pick class and method names
  - Decide what arguments might be needed
  - Decide return types
  - Write the class, constructors, instance variables, and methods to make the what you want work
- The unit tests are there to verify the code works and to identify errors when **refactoring**

JUnit's Assertions

- `void assertEquals(boolean expected, boolean actual)`
- `void assertTrue(boolean condition)`
- `void assertFalse(boolean condition)`
- `void assertNotNull(Object object)`
- `void assertNull(Object object)`
- Lots more like: `assertThrows, assertSame, assertAll, assertTimeout, assertArrayEquals, …`

Structure of the Regression Suite

- The regression suite is
  - a collection of *unit tests*, each one of a single class, which may reference other classes, and
  - *integration tests*, which test a set of classes
- Each test is a collection of individual *test cases*
- Some of the test cases are *black box tests*; the rest are *white box tests* (this should be indicated in the class comments)
- Each test has the same general structure

Test-Driven Design

- TDD turns traditional development around
  - Write test code before the methods
  - Do so in very small steps
  - Refuse to add any code until a test exists for it
- You can run the test any time
  - And you should do so quite often…

Test Cases in Java

- A test case is a single class, a JUnit test, whose name has a suffix of "`Test`".
- A test case is an ordered collection of test methods.
- JUnit does not guarantee order.
  - The best approach is to have each method independent of the others, so that they can be executed in any order.
  - If any tests are order-dependent, it is best to have one method calling sub-methods (that don't have an `@Test` annotation), in the needed order.

Structure of a Test Case

- `@Before` method
  - This method is run before any test methods. It sets up the environment.
  - Consists of setters, followed by getters, followed by assertion(s) that ensure that the setup was successful.
- `@Test` method(s)
  - The test methods are ordered by comprehensiveness of further preconditions.
- `@After` method
  - This method is guaranteed to run after the test methods, even if they throw an exception. It cleans up the environment.
  - Consists of setters. No assertions should be present.

Structural Testing

- Consider code that contains an if/else statement:

```
if(x < 100) {
        //do something
}
else {
        //do something else
}
```

- What kind of test cases would we want to generate?

@Test Method(s)

- Tests part of the functionality under test.
- Consists of further setup (same structure as above), followed by getters and one or more assertions.
- The assertion(s) are the core part of the test method.
- Can repeat, as long as precondition of tested method continues to be met

Coverage

- We would want to try to **cover** all of our code using test input
  - So we'd want some value for x < 100 and one for greater
- This would allow us to have 100% **coverage** of the *paths* of execution through our program
- Any additional test cases wouldn't add anything new here, even if the values were "interesting"

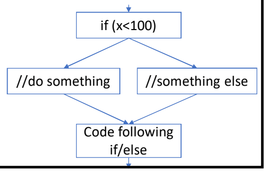
Test Suite Quality

- How do we know we have done a good job of testing?
- We would like to have some measure of how good our test cases actually exercise the code
- One way is to use the structure of the program to guide us

Path Coverage

- Use test cases to ensure that every possible path through our program's execution is exercised during testing
- We can consider our program's flow-chart representation to see where we get this notion:

```
if(x < 100) {
    //do something
}
else {
    //do something else
}
```



Path Coverage

- 100% path coverage turns out to be impossible for any realistic program as the number of paths can be infinite (loop 1 time, loop 2 times, loop n times, etc.)
- So we use weaker coverage criteria out of necessity
  - But weaker means we have less confidence in our test suite being "complete"

Branch Coverage

- **Branch coverage** tests how many of the branches/jumps/control transfers within the code have been taken
- In our previous example, 2 out of 3 branches were taken, giving a 66% branch coverage
  - This is a better measure than statement coverage of our test suite quality
  - Still doesn't test every iteration of a loop, we only need to go through 1 iteration to get full branch coverage

Statement/Node Coverage

- A very weak, but common coverage criteria is to instead measure the number of source code **statements** that were covered
- This means every line of code has run during testing, but not every interaction of code has been exercised

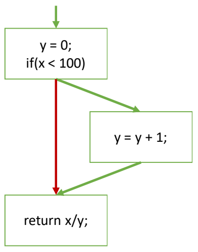
Coverage Tools

- There are many coverage tools for Java
  - Including the one I wrote for my thesis
- We can use eclEMMA under eclipse to do statement and branch coverage and evaluate the quality of our test suite
- We can also use the feedback to reduce the number of test cases (if a test case doesn't test a new statement, branch, or path it doesn't need to exist)

Statement/Node Coverage

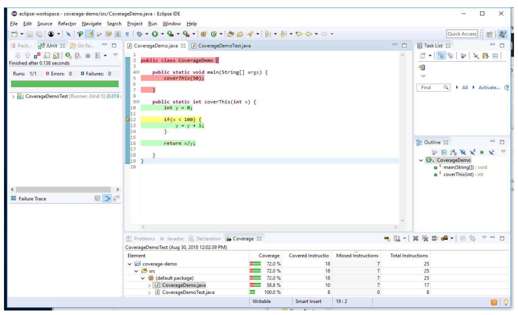
- With x = 50, this code has 100% statement coverage:

```
y = 0;
if(x < 100) {
        y = y + 1;
}
return x/y;
```



- But only has 50% path coverage

eclEMMA


eclipse-workspace - coverage-demo/src/CoverageDemo.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Pack... JUnit Git Re...
CoverageDemo.java CoverageDemoTest.java
Task List
Find All Activate...
Outline
Failure Trace
Problems Javadoc Declaration Coverage
Writable Smart Insert 19 : 2

Finished after 0.138 seconds

Runs: 1/1 Errors: 0 Failures: 0

CoverageDemoTest [Runner: JUnit 5] (0.019

```
public class CoverageDemo {

    public static void main(String[] args) {
        coverThis(50);

    }

    public static int coverThis(int x) {
        int y = 0;

        if(x < 100) {
            y = y + 1;
        }

        return x/y;

    }
}
```

CoverageDemo
main(String[]) : void
coverThis(int) : int

CoverageDemoTest (Aug 30, 2018 12:02:39 PM)

Element	Coverage	Covered Instructio	Missed Instructions	Total Instructions
coverage-demo	72.0 %	18	7	25
src	72.0 %	18	7	25
(default package)	72.0 %	18	7	25
CoverageDemo.java	58.8 %	10	7	17
CoverageDemoTest.java	100.0 %	8	0	8