Developer Testing

Unit Testing
Logging
Assertions
ANT
Debugging
Testing practices

- Testing is the most popular quality-improvement activity
  - *Regression Testing*: The automated repetition of previously executed tests cases for the purposes of finding defects in software that previously passed the same set of tests earlier in the design cycle.
- Best practices include continuous testing at multiple development points:
  - *Unit testing*: The testing of a method/routine, a complete class, or small program (with a few of classes) in isolation from the more complete system. These tests can be performed before the application is complete.
  - *Application testing*:
    - *Component Testing*: Testing (in isolation) of a complete package (or other logical component) of a large program developed by multiple programmers/teams
    - *Integration Testing*: Testing the combined execution of two or more packages, components, other or subsystems.
    - *System Testing*: Testing the full application for security, resource loss, timing, and other issues that can’t be tested at lower levels of integration.
Unit Testing

- *Unit testing*: The testing of a complete class, method/routine, or small program in isolation from the more complete system.
  - The single most important testing tool
  - The *focus* for testing for novice programmers
  - You don't test the complete program that you are developing; you test the classes and methods in isolation

- Methodology:
  - Each test checks a single method or a set of cooperating methods in the same class
  - Each test is developed as a test harness (a simple class)
  - The Test harness feeds parameters to the methods being tested and verifies the proper behavior/result
  - Test harnesses allow unit tests to be easily reused later in the development cycle (regression testing)
What should we test?

- A best practices unit test includes:
  - Positive Tests
    - Tests that case that we expect to work do, in fact, work as expected
    - Tests for expected inputs
  - Negative Tests
    - Tests for cases that we expect to fail do, in fact, handle the error appropriately
    - Test for unexpected inputs
- Pitfall: It is very common for inexperienced programmers to *only* conduct positive tests. Negative tests, testing that what should go wrong, indeed does go wrong, and does so in a well defined manner – is crucial for a good test procedure.
Test Coverage

- **Black-box testing**
  - test functionality without consideration of internal structure of implementation
  - tests for boundary conditions, problem inputs, expected behavior, etc.
  - Example: I test for 0, -1, -100000, because I _don’t_ know if they are all part of the same logic path

- **White-box testing** (The box is really transparent!)
  - take internal structure into account when designing tests
    - Example, if there is an if/else statement, then test both paths
    - Example: I don’t need to test -1, -2, … because I know that my if statement tests if (n <= 0)!
  - Make sure that each possible path of flow control is exercised

- Which is better?

- **Test coverage**: percentage measure of how many parts of a program have been tested (or how much of the input space has been tested)
Example: Approximate Square Rooter

- Algorithm to compute the approximate square root of \( a \):
  - Guess an initial value \( x \) that is “close” to but greater than the square root (say \( x = a \))
  - Actual square root lies between \( x \) and \( a/x \)
  - Take midpoint \((x + a/x) / 2\) as a better guess
- Repeat until two successive approximations are “very close” to each other
- When should we design tests?
  - Before/After implementation?

Figure 1: Approximating a Square Root
What should we test?

- **Positive Tests:**
  - Things we know to be true
    - `squareRootOf(4) == 2`
    - `squareRootOf(100) = 10`
    - `squareRootOf(1) == 1`
  - Things that must be true as a postcondition
    - The return value squared should be “close” to the parameter passed
      
      ```
      n*n - MyMath.DELTA <= squareRootOf(n) <= n*n +MyMath.DELTA;
      Math.abs( Math.power(n, 0.5) - squareRootOf(n) ) <= MyMath.DELTA
      ```

- **Negative Tests:**
  - Things we expect to be problems
    - `squareRootOf(0) == ?`
    - `squareRootOf(-1) == ?`

- **Consider Boundary Conditions:**
  - What about `n = 0, 1, -1, Double.MAX_VALUE, Double.MIN_VALUE, Double.NaN, Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY`?
Example: Naïve implementation of Square Root Approximator

```java
// This class has a Naive Implementation of a square root approximator
public class MyMath {
    public static final double DELTA = 0.001;

    public static double squareRootOf (double n) {
        double approximation = n;
        double lastApproximation;

        do {
            lastApproximation = approximation;
            approximation = (n/approximation + approximation)/2.0;
        } while ( Math.abs(lastApproximation - approximation) > DELTA  ) ;

        return approximation;
    } // end method squareRootOf

    public static void main (String[] args) {
        System.out.println( squareRootOf (4) );
    } // end method main
}
```

Wright State University, College of Engineering
Dr. T. Doom, Computer Science & Engineering
import junit.framework.*;

public class MyMathTest extends TestCase {

    public MyMathTest (String testName) {
        super (testName);
    }

    /**
     * Test of squareRootOf method, *
     * of class unittesting.MyMath.
     */
    public void testSquareRootOf () {
        System.out.println("squareRootOf");
        double n, actualResult, expectedResult;
        boolean theTest = true;
        String errorMessage = "";

        errorMessage = "Testing square root 4.0 ~= 2.0";
        n = 4.0;
        expectedResult = 2.0;
        actualResult = MyMath.squareRootOf(n);
        theTest = Math.abs(expectedResult - actualResult) < MyMath.DELTA;
        assertTrue(errorMessage, theTest);

        errorMessage = "Testing square root 0.0 ~ 2.0";
        n = 0.0;
        expectedResult = Math.sqrt(n);
        actualResult = MyMath.squareRootOf(n);
        theTest = Double.isNaN(actualResult);
        assertTrue(errorMessage, theTest);

        errorMessage = "Testing square root -1";
        n = -1.0;
        expectedResult = Math.sqrt(n);
        actualResult = MyMath.squareRootOf(n);
        theTest = Double.isNaN(actualResult);
        assertTrue(errorMessage, theTest);

        // more tests ...
    } // end method testSquareRootOf
} // end class MyMathTest

errorMessage = "Testing square root" + Double.MIN_VALUE;

n = Double.MIN_VALUE;
expectedResult = Math.sqrt(n);
actualResult = MyMath.squareRootOf(n);
theTest = Math.abs(expectedResult - actualResult) < MyMath.DELTA;
assertTrue(errorMessage, theTest);

errorMessage = "Testing square root 0"

n = 0.0;
actualResult = MyMath.squareRootOf(n);
theTest = Double.isNaN(actualResult);
assertTrue(errorMessage, theTest);

errorMessage = "Testing square root -1"

n = -1.0;
actualResult = MyMath.squareRootOf(n);
theTest = Double.isNaN(actualResult);
assertTrue(errorMessage, theTest);

// more tests ...
} // end method testSquareRootOf
} // end class MyMathTest
Creating a test harness using JUnit

JUnit helps automate regression testing by using a specific testing framework

- JUnit tools will create the default test harness for you
  - In Netbeans, left click on class, select Tools/Create JUnit tests
  - You need to *exercise* the methods yourself

By creating a complete test harness, you guarantee that your code continues to work as promised even after changes to your (or others!) code.

- This is the essence of regression based unit testing

JUnit tests can be performed by class, package, or application

- In Netbeans, select Run/Run tests to test a package
- In Netbeans, left click test class and Run File to test just one class

For more about JUnit: [http://www.junit.org/](http://www.junit.org/)
You found an error! Now what?!?

- Most common ways to track down errors:
- Manual walkthrough
  - Print out the code and logically deduce the source of the error
- Print statements
  - Eg: System.out.println(“….”);
  - Drawback: Need to remove them when testing is complete, stick them back in when another error is found
  - Use a Boolean constant to easily turn on/off debugging information
  - Use Logging to easily turn on/off debugging information
- Use Assertions
- Use Dynamic Debugging
Logging

- When tracing execution flow, the most important events are entering and exiting a method
- At the beginning of a method, print out the parameters
- At the end of a method, print out the return value
- Logging messages can be deactivated when testing is complete

```java
import java.util.logging.*;
public class HelloWorld {
    private static Logger theLogger;
    private String theMessage;

    public HelloWorld(String message) {
        theMessage = message;
    } // end constructor

    public void sayHello() {
        theLogger.info("Hello logging!"); //use Logger to debug
        System.out.println(theMessage);
    } // end method sayHello

    public static void main(String[] args) {
        theLogger = Logger.getLogger(HelloWorld.class.getName());
        HelloWorld hello = new HelloWorld("Hello world!");
        HelloWorld.theLogger.setLevel(Level.OFF); //turn logger off
        hello.sayHello();
    } // end method main
} // end class HelloWorld
```
Run-Time Debugger

- Advantages:
  - you can stop the program at breakpoints
  - you can examine the contents of memory (including the heap and stack) at leisure
  - you can modify memory contents on the fly for dynamic testing
  - step through or into code line by line.

- Downsides:
  - Can be very time-consuming
  - Debuggers generally not available for concurrent programs