Spring 2012 Course: CS 498, Section DM

Software Testing

Problem Set 1
Assigned: January 29, 2012
Due: Thursday, February 9, 2012 (at 12:30pm, beginning of lecture)

This problem set covers the material from Chapter 1 of the textbook, basics of JUnit and statement coverage, and basics of JPF. There are six problems, worth a total of 120 points. You need only 100 points to get the maximum score for this problem set, which accounts for 15% of the final grade. If you have more than 100 points, you will get credit for the future problem sets and the project.

For the problems labeled **[MP]**, you will have to submit your code through SVN. The detailed instructions will be provided on the cs498dm mailing list when TSG sets up an appropriate SVN repository. For the other problems, you can include the solutions as text files in SVN, or you can bring hand-written/printed solutions to the lecture. Some starting files for the MP problems are available at **http://mir.cs.illinois.edu/~marinov/sp12-cs498dm/pset1.zip**.

Discussing problems on the cs498dm mailing list is allowed and encouraged! However, you should write your solutions individually.

Problem 1 [20 points]: [MP] (Based on the **trityp** example tested interactively in the lecture.) Consider the class **trityp** from **problem1** in pset1.zip.
(a) [4 points]: Write four JUnit *passing* tests for the **Triang** method.
(b) [4 points]: Change the **trityp** class to enable partial testing of the string output (rather than just the integer value that encodes the triangle type).
(c) [4 points]: Write four JUnit *failing* tests where the code returns a string different from what you would expect. (We already had several examples in the lecture, so just encode them in JUnit.)
(d) [4 points]: Write four JUnit tests that provide inputs directly to the **main** method (rather than to the **Triang** method that computes the result); these tests need not check the output.
(e*) [4 points]: Write generic code that can check the output of the **main** method (capture **System.out**) and then add specific output checks to the four tests you wrote in the previous part. (***** Hard question.)

Problem 2 [20 points]: [MP] (Based on Exercise **2** after Section **1.1**.) The following exercise is intended to encourage you to think of testing in a more rigorous way than you may be used to. The exercise also hints at the strong relationship between specification clarity, faults, and test cases.
(a) [4 points]: Write a Java method **union** with the signature

public static Vector union(Vector a, Vector b)

The method should return a **Vector** of objects that are in either of the two argument **Vector** objects. Describe in code comments how you interpreted the notion of union.
(b) [4 points]: Upon reflection, you may discover a variety of defects and ambiguities in the given assignment. In other words, ample opportunities for faults exist. Identify at least *eight* potential faults.
(c) [4 points]: Write in JUnit a set of test cases that you think would have a reasonable chance of revealing the faults you identified above. Document a rationale for each test in your test set. If possible, characterize all of your rationales in some concise summary. Run your tests against your implementation.
(d) [4 points]: What statement coverage do your tests achieve? You can use (Ecl)Emma or a similar tool to measure coverage, and it need not be 100%. Is 100% statement coverage feasible for your code?
(e) [4 points]: Rewrite the method signature to be precise enough to clarify the defects and ambiguities identified earlier. You may wish to illustrate your specification with examples drawn from your test cases.

Problem 3 [20 points]: [MP] (Based on an old exercise after Section **1.2**.) Consider the class **Count** from **problem3** in pset1.zip.

(a) [4 points]: Complete this program by modifying the **main** method to contain exactly one call to the **numZero** method. The arguments for the call can be read from the command line (parsing **args**) or from standard input (see the **trityp.java** example in **problem1**).

(b) [4 points]: This program contains a fault. What is it? Does executing the program necessarily result in either incorrect output or in failure?

(c) [4 points]: Write a JUnit test case that results in failure. Verify by executing this test case. For the test case that results in failure, identify some error state. Be sure to describe the complete state.

(d) [4 points]: Write another JUnit test case that does not result in failure. Verify by executing this test case.

(e) [4 points]: Write a JUnit test suite to execute every *statement* in the method **numZero**. Discuss whether you wrote a *minimal* set, i.e., a set such that if any one test was removed, the remaining tests would no longer execute every statement.

Problem 4 [20 points]: (Exercise **3** after Section **1.2**.) Consider the following faulty method and a test case that results in failure:

```
public int countPositive(int[] x) {
   // Effects: If x == null throw NullPointerException
   // else return the number of positive elements in x.
   int count = 0;
   for (int i = 0; i < x.length; i++) {
      if (x[i] >= 0) {
         count++;
      }
   }
   return count;
}
// test: x = [-4, 2, 0, 2]
// expected = 2
```

(a) [4 points]: Identify the fault.

(b) [4 points]: If possible, identify a test case that does **not** execute the fault.

(c) [4 points]: If possible, identify a test case that executes the fault, but does **not** result in an error state.

(d) [4 points]: If possible identify a test case that results in an error, but **not** a failure. Hint: Don't forget about the program counter.

(e) [4 points]: For the given test case, identify the first error state. Be sure to describe the complete state.

Problem 5 [20 points]: This exercise asks you to try out basics of some parts of Java PathFinder (JPF). JPF can be used to explore non-deterministic choices, including various thread interleavings or explicit non-determinism. For example, JPF's library call **Verify.getInt(int min, int max)** introduces an explicit non-deterministic choice point that can return values between **min** and **max**, inclusively. Write a program that has some thread interleavings or explicit non-deterministic choices. You can start from the example code used in the lecture about JPF and modify it slightly but do *not* use the exact code from the slides.

(a) [4 points]: Run JPF on your program. How many states did JPF explore? (If you have more than 1000 states, your program is too big, so write a smaller one.) How many times did JPF backtrack in the exploration? How many bytecode instructions did JPF execute?

(b) [4 points]: Run JPF for some non-default search strategy. How does it affect the number of states explored and the number of backtracks performed?

(c) [4 points]: Run JPF with some listener. What additional information does it provide?

(d) [4 points]: Run JPF with **DebugJenkinsStateSet**. What additional information does it provide?

(e) [4 points]: Run some JPF feature that is *not* a search strategy or listener. What does it provide?

Problem 6 [20 points]: The basic functionality that JPF has is to execute a Java program as any regular JVM does. This exercise asks you to test whether JPF gives the same output as a regular JVM. You will write Java programs with various features, run them on both JPF and JVM, and compare the outputs. You will submit the programs you wrote, the script/code you used for comparing outputs, and description of any unexpected outputs you may obtain.

(a) [5 points]: Choose a Java library that you are not very familiar with (e.g., reflection or I/O). Write or download five *small* programs that use this library. (If you download code, write where you got it from.) Test if JPF gives the same output as JVM for those programs.

(b) [5 points]: Select some code with JUnit tests from the previous problems in this problem set. Run these JUnit tests on JPF. Is the output the same as when running on JVM?

(c) [5 points]: Choose some Java code slightly larger than the examples from the previous problems (say, choose a program with 5-10 classes). This can be code that you wrote previously or code that you download from the Internet. (Again, if you download code, write where you got it from.) Test if JPF gives the same output as JVM for this code.

(d) [5 points]: Java language version 7 was recently released, and it offers several new features and enhancements, listed at **http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html**. Choose one of these new features. Write or download five *small* programs that use this feature. (As always, if you download something from somewhere, write where you got it from.) Test if JPF gives the same output as JVM for those programs.