Spring 2012 Course: CS 498, Section DM

Software Testing

Problem Set 3
Assigned: March 30, 2012
Due: Thursday, April 5, 2012 (by 12:30pm, beginning of class)

This problem set covers the material from Chapter 5 of the textbook and some 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]**, commit your code into SVN. For the non-MP problems, you can either include your solutions in SVN (if so, please use simple ASCII documents to write your solutions) or you can bring your (hand-written or printed) solutions to the lecture. **Important note1:** do not collaborate on any of these problems, especially on the MP parts! If you have questions, please feel free to schedule a meeting with Darko or email him the questions. **Important note2:** you're welcome to use any online resource you want as a help, including the book web page (http://cs.gmu.edu/~offutt/softwaretest/), but if you copy some solution/idea from somewhere, please do state the original source. Some of these online resource have bugs themselves. If you do not state the original source but have exactly the same bugs (which would be unlikely), you will get negative points!

Problem 1 [15 points]: (This is a modified version of exercises after Section **5.1.2**.)
Consider the Stream BNF (page 171, Section 5.1.1, also see slides) and the string "B 10 08.27.98".
(a) [3 points]: Give three valid and three invalid mutants of the above string.
(b) [3 points]: Specify three mutation operators for the Stream grammar (not Stream strings).
(c) [3 points]: For each operator, discuss whether it always produces a valid mutant, always produces an invalid mutant, or could produce either.
(d) [3 points]: For each of your mutants from part **(a)**, describe whether and how it can be generated using your mutation operators from part **(b)**.
(e) [3 points]: Compute the number of mutants (if it is too hard to compute the exact number, then approximate) that your operators can produce from the above string.

Problem 2 [25 points]: [MP] Write a program that generates random tests in JUnit format for a given class. See SVN for more details (search for **TODO**).

Problem 3 [5 points]: (This is a modified version of exercises after Section **5.1.2**.)
(a) [2 points]: Define mutation score.
(b) [3 points]: How is the mutation score related to coverage from Chapter 1 (pages 16-20)? What are the test requirements for mutation coverage? Which requirements are infeasible?

Problem 4 [25 points]: [MP] (This is a modified version of Exercise **6** after Section **5.2**.) For the method **power,** given in SVN, do the following:
(a) [16 points]: Define at least eight non-equivalent mutants for **power**. If possible, use for each mutant a different operator from those effective mutation operators listed on pages 182-185, Chapter 5. If not possible, discuss why.
(b) [3 points]: Write a set of test inputs (not in JUnit format) that strongly kills all mutants.
(c) [6 points]: Define at least three equivalent mutants for **power**.
See SVN for details on writing your mutants and tests (search for **TODO**).

Problem 5 [30 points]: (This is a modified version of Exercise **3** after Section **5.2**.) This problem considers the **TestPat** class from page 56, Chapter 2. (This is the same class from Problem 4 in Problem Set 2.) Consider two mutants for this class:

(A) while (isPat == false && isub + patternLen - 1 < subjectLen) // Node 3
    while (isPat == false && isub + patternLen - **0** < subjectLen) // Mutant A

(B) isPat = false; // Node 8
    isPat = **true**; // Mutant B

For each mutant, do the following:

(a) [3 points]: If possible, find a test input that does not satisfy reachability for the mutant (i.e., a test input whose execution does **not reach** the mutation).

(b) [3 points]: If possible, find a test input that satisfies reachability but **not infection** for the mutant (i.e., a test input that reaches the mutation but does not result in a modified state).

(c) [3 points]: If possible, find a test input that satisfies infection but **not propagation** for the mutant (i.e., a test input that reaches the mutation and results in a modified state, but this state does not propagate to the return value).

(d) [3 points]: If possible, find a test input that kills the mutant weakly but not strongly. (Hint: consider a brief answer to this question.)

(e) [3 points]: If possible, find a test input that strongly kills the mutant.

If any of the above is not possible, discuss **why**; simple "no" does not suffice. (Note about points: each of the ten parts A.a, A.b… A.e, B.a, B.b… B.e is worth 3 points.)

Problem 6 [20 points]: [MP]** Java Pathfinder (JPF) can be used to explore various sources of non-determinism such as thread interleavings in concurrent code and explicit non-determinism introduced with the JPF's library call **Verify.getInt(int min, int max)** that returns values between **min** and **max**, inclusively. The goal of this problem is to use such library calls to create sequential test sequences for the **Account** class from **Problem 2**. Recall that each JUnit test creates several objects and invokes methods on them, and then finally asserts that some properties should hold.

In particular, we will focus on the **Account** class and its methods **deposit** and **withdraw**. For each method, we will use only a finite, specified set of parameter **values**. (The provided sample code uses "1" and "2" as parameters, but the code that you write should work not only for those two parameters but for all possible **values** arrays; in other words, your code should be generic and not rely on the particular, provided array, e.g., you should use **values.length** rather than 2 where appropriate.) The goal is to explore method sequences that have a specified number of method calls using these parameters. In general, we can be checking some properties for each such sequence. In this MP, the goal is to output one JUnit test for each different sequence.

Your specific task is to change the **AccountDriver** class provided in SVN so that running JPF on this class generates a sequence of JUnit tests as described above. The program creates **Account** objects through execution of a sequence of method calls. At the same time, the program creates the string/code necessary to reconstruct those objects as part of a JUnit test. (In principle, as each **Account** object is created, the program should compare the state of that object against previously generated objects, and it should print JUnit tests only for those objects that have new, previously unseen states. You need not worry about that in this problem.)

Note that the **main** method accepts a single argument. This argument is the length of the method sequence (i.e., the number of **deposit** and **withdraw** method calls) that should be generated. Your solution should be general and able to run correctly for sequence lengths greater than or equal to 1.

(This problem is very hard, much harder than any previous we had. Please attempt this last only if you have time and only if you didn't already do well on your project and previous problem sets.)**