Spring 2015 Course: CS 498, Section DM

Software Testing

Machine Problem 5
Assigned: April 22, 2015 (by 11:59:59pm AoE time)
Due: Wednesday, April 29, 2015 (by 11:59:59pm local time)

This MP covers more material from chapter 5 (and a bit of 4) of the textbook. You will obtain some initial files and submit your added and modified files through SVN. If you add some unnecessary files (e.g., target), you will get negative points. Your SVN directory is as before. You are strongly encouraged to set up Jenkins to run your MP5 code on your VM even if you develop the code on another machine.

There are five problems (and an easy, no problem task), worth a total of 120 points. As before, you need only 100 points to get the maximum score for this MP, i.e., if you have 100 points on each MP, you get A+ for the MP portion of the course.

You cannot collaborate on the first four problems, but members of each team should collaborate on Problem 5, without teams collaborating with one another! You should not share your solutions or code on Piazza! If you share your solutions, we will have to give negative points. Write your solutions individually, without discussing specific code with others. If you use some online resource, you must credit the original source. Many online resources have bugs, and we hope to show you some.

No Problem [5 points]: If you notice some bugs related to the course (in book, slides, code, tests, text...) any time during the course, please report them to Darko, and you can get extra credit!

Problem 1 [25 points]: (*) Write a program that generates random tests in JUnit format for a given class. For more details, search for TODO in SVN. Note that you can get credit for writing tests and setting up Jenkins for this problem. Please do NOT put your NetID or other personally identifiable information in the code or tests; we may share the code and tests with the entire class.

Problem 2 [20 points]: (This is a modified version of Exercise 6 after Section 5.2.) For the method power, given in SVN, do the following:
(a) [8 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.
(b) [6 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 3 [20 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”. Write answers in a file called prob3.txt in your mp5 directory.
(a) [4 points]: Give three valid and three invalid mutants of the above string.
(b) [4 points]: Specify three mutation operators for the Stream grammar (not Stream strings).
(c) [4 points]: For each operator, discuss whether it always produces a valid mutant, always produces an invalid mutant, or could produce either.
(d) [4 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) [4 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 4 [30 points]: This problem comes in two variants: V1 and V2.
If you’re taking the course for 3 hours of credit, choose to do only one variant, V1 (x) or V2.
If you’re taking the course for 4 hours of credit, do both variants, V1 and V2.

V1 (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 that you “know and love” from several other problems.) 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

Write answers in a file called prob4.txt in your mp5 directory. 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.)

V2 part 1: (This is a modified version of Exercise 1 after Section 4.1, page 159; the entire text is here.) Answer the following questions for the method search below:

public static int search(List list, Object e)
// Effects: If list or e is null, throw NullPointerException
// else if the element e is in the list, return an index of e in the list; else return -1
// for example, search([3,3,1], 3) returns either 0 or 1 while search([1,7,5], 2) returns -1

Base your answers on the following characteristic partitioning “Location of e in list”:
Block 1: e is first entry in list
Block 2: e is last entry in list
Block 3: e is in some position other than first or last

(f) [3 points]: “Location of e in list” fails the disjointness property. Give an example that illustrates this.
(g) [3 points]: “Location of e in list” fails the completeness property. Give an example that shows this.
(h) [4 points]: Describe the input domain for the method search.
(i) [4 points]: Supply one or more new partitions that capture the intent of “Location of e in list” but do not suffer from any disjointness or completeness problems. Do NOT use only partitions with two blocks.

V2 part 2: Consider the following specification for a sort method:

public static void sort(int[] s)
// Precondition: s != null
// Effects: sort elements of s (smaller elements first)

Suppose that sort is checked in the following way, with the code provided in SVN:
(1) Make a copy t of s.
(2) Apply the method sort to s.
(3) Verify that every element in s is also an element in t.
(4) Verify that every element in t is also an element in s.
(5) Verify that s[i] <= s[i+1] for appropriate values of i.
(j) [4 points]: Do we need to test \texttt{sort} with \texttt{s} being \texttt{null}? What is an expected “output” in that case? Provide \texttt{explanation} for your answers; simple “yes” or “no” is not sufficient.

(k) [4 points]: The provided code that implements the checks has a few problems with array index bounds. Identify and correct those problems.

(l) [4 points]: Find initial and “sorted” values for \texttt{s} such that all of the given checks (with correct implementations for array index bounds) succeed, but the sorted version of \texttt{s} is still wrong. Propose an additional check or modify existing checks to detect this (counter) example with “sorted” values.

(m) [4 points]: Write a specification for the \texttt{binarySearch} method provided in SVN and implement a simple generic check for that method.

\textbf{Problem 5 [20 points]:} The goal of this problem is to help you make more progress on your project. You will commit the files in your \texttt{sp15-cs498dm/_projects/netids/mp5/} directory.

(a) [5 points]: For some part of the project you are testing (it can be the same as in MP4 Problem 5 or in \texttt{progress.txt}, or something new), create a grammar that defines the inputs.

(b) [5 points]: Based on your grammar, prepare test cases that satisfy some coverage criterion. Write your test cases in a format used by your project. Discuss what criteria are appropriate to use. How many test inputs would there be in “Production Coverage” for the grammar that you created?

(c) [5 points]: Manually construct at least five mutants for the code under test, and prepare test cases that kill all those mutants. Note that each test case should not only provide the input to the code but also check the output.

(d) [5 points]: Write a simple text file with several bug reports that you could potentially submit for the project, but do not submit those reports yet. Make sure to include everything that should go in a bug report (inputs given to the program, observed actual output, and expected output).