Spring 2015 Course: CS 498, Section DM

Software Testing

Machine Problem 2
Assigned: February 24, 2015 (by 11:59:59pm AoE time)
Due: Wednesday, March 4, 2015 (by 11:59:59pm local time)

This MP covers more material from chapter 2 of the textbook and somewhat advanced usage of Jenkins. 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 negatives points. As before, your SVN directory is https://subversion.ews.illinois.edu/svn/sp15-cs498dm/netid, where netid is your NetID. You are required to use the department-provided virtual machines (VMs) for some parts of this MP, but you are NOT required to use these VMs for all other parts. However, you should set Jenkins to run your MP2 code on your VM even if you develop the code on another machine (e.g., on your laptop)! Your primary VM remains cs498dm-XyA.cs.illinois.edu. Remember, to access your VM, you should be on the university network. If you don’t use SSH keys, you are not being efficient.

There are four problems, worth a total of 120 points. 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. If you have more than 100 points, you will get credit for the future homework assignments and the project.

Discussing these problems on Piazza is allowed and encouraged, but do 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. However, you can use any online resource, including the book web page (http://cs.gmu.edu/~offutt/softwaretest/) for help, but if you copy some solution/idea, please do credit the original source. Many online resources have bugs; if you credit no source but have exactly the same bugs (which would be rather unlikely), you will get negative points!

General reminder: 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 [30 points]: This problem asks you to implement the algorithm for finding prime paths for a given graph, as shown in the book and on the slides. Some initial files are provided in SVN, but you may add more files. Remember to add all the files you need (using Jenkins in part 1(a) helps you with that) and not to add any files you don’t need (e.g., consider “svn stat” or “svn propset svn:ignore -F”).

(a) [5 points]: Create a Jenkins job called “mp2” to run your solution as you develop it. Most likely you want to create a Maven job, similar as in MP1 problem 2(d). You should run this job for every commit you make in SVN. We expect you to have at least three commits/builds. Copy-paste the Jenkins Console Output from your last (and successful) build into a file called mp2_console.txt in your mp2 directory.

(b) [10 points]: Even before you write the code to compute prime paths, write at least five tests for the code and commit them in SVN so that they can be run with “mvn test” (you may want to use the provided PrimePathsTest.java). It is okay that these tests fail initially. You will make them pass as you write the code. (In real development, you would not commit failing tests, even if you write them before the code, but in this exercise we want to ensure that you have several builds of the Jenkins job.) Your tests will be graded based on (1) the style of your test code (including proper naming, use of methods/classes, and proper use of JUnit assertions), (2) not failing for the correct implementation (i.e., your tests being correct), and (3) failing for some incorrect implementations (e.g., how many bugs your tests find in the other students’ code). (We will limit each test-suite run to 1 minute on the department-provided VMs.)
(c) [5 points]: Write the code to compute prime paths. Your code will be graded based on how many tests it passes. Your final SVN commit should pass all your own tests: if the code does not compile or pass your tests, get 0 points! (Hint: If you can’t get the code to work for some input, hard-code the output.)

(d) [10 points]: Measure the statement coverage that your tests achieve on your code using Cobertura to generate an XML coverage report. Commit this report as a file called 1-coverage.xml in your mp2 directory (e.g., with “cp target/site/cobertura/coverage.xml 1-coverage.xml”). You will add some more tests and commit either 2-coverage.xml or 100-coverage.xml. Whenever you add some new tests, remember to commit them to SVN and run your Jenkins job.

If your 1-coverage.xml has less than 100% statement coverage, add some new test(s) to increase statement coverage, measure the new coverage (need not be 100%), and commit 2-coverage.xml.

If your 1-coverage.xml already has 100% statement coverage, congratulations! Still, add some new test(s) to increase some stronger coverage (e.g., branch or path), and commit 100-coverage.xml.

(e**) [5 points]: JGraphT uses JUnit 3 not JUnit 4, so look into this only if you have time and like challenges. While you need not use the JGraphT library for your implementation, you could still add some tests to improve code coverage for the JGraphT core module. Measure coverage for JGraphT, e.g., using jgrapht.sh, you should get 80% “Line Coverage” for “All Packages”. Change some existing test class(es) to improve the line coverage reported by Cobertura by 1% (you would need to cover at least 10 more lines). Add to SVN only the modified test class(es), NOT the entire JGraphT source.

Problem 2 [30 points]: In this problem, you are hired as an expert Jenkins consultant to improve your Jenkins jobs for the current and past self. For questions about Jenkins, feel free to ask on Piazza!

(a) [10 points]: Jenkins derives a lot of power from the large number of plugins it has. Create a Jenkins job called “mp2_cobertura” that connects the Cobertura plugin for Jenkins (https://wiki.jenkins-ci.org/display/JENKINS/Cobertura+Plugin) with your Jenkins job for problem 1. Submit the Jenkins configuration file for this job (i.e., Jenkins/jobs/mp2_cobertura/config.xml) as config_cobertura.xml.

(b**) [5 points]: The Ekstazi plugin (https://github.com/peterlvilim/EkstaziJenkinsPlugin) was developed by Peter Vilim and Aditya Ahuja for the Ekstazi tool presented by Milos Gligoric in the guest lecture. Create a Jenkins job called “mp2_ekstazi” that connects this plugin with your Jenkins job for problem 1. Submit the Jenkins configuration file for this job (i.e., Jenkins/jobs/mp2_ekstazi/config.xml) as config_ekstazi.xml.

(e*) [5 points]: Jenkins is a complex piece of software, and its JUnit tests are rather advanced (https://wiki.jenkins-ci.org/display/JENKINS/Unit+Test). We mentioned on Piazza several tests (from Jenkins/test/src/test/java) that are related to the “hello world” example job from MP1 problem 2(b): org.jvnet.hudson.main.AppTest#test{1,2}, hudson.model.FreeStyleProjectTest#configSubmission, and hudson.LauncherTest#quiet. Write a new JUnit test to automate the steps from MP1 problem 2(c). Submit the new test class to SVN as New2cTest.java.

(d*) [5 points]: As for part (c), write a new JUnit test class to automate the steps from MP1 problem 2(d). Submit the new test class to SVN as New2dTest.java.

Problem 3 [25 points]: Graph I Exercise 1 after Section 2.2.3 (page 51, Chapter 2). Consider this graph and test paths: N = {0, 1, 2, 3, 4, 5, 6, 7}, N0 = {0}, N1 = {7}, E={(0, 1), (1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)}, def(0) = def(3) = use(5) = use(7) = {x}, t1 = [0, 1, 7], t2 = [0, 1, 2, 4, 6, 1, 7], t3 = [0, 1, 2, 4, 5, 6, 1, 7], t4 = [0, 1, 2, 3, 2, 4, 6, 1, 7], t5 = [0, 1, 2, 3, 2, 3, 2, 4, 5, 6, 1, 7], t6 = [0, 1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]. Write answers in a file called prob3.txt in your mp2 directory.

(a) [0 points]: Draw the graph. This makes it easier to answer the rest, but you need not turn anything in.
(b) [5 points]: List all du-paths with respect to \(x\). (Note: include all du-paths, even those that are subpaths of some other du-path.)

(c) [5 points]: For each test path, determine which du-paths that test path tours. For this part of the exercise, you should consider both direct touring and sidetrips. Show your results as a table in the format given in SVN, with both rows and columns ordered lexicographically.

(d) [5 points]: List a minimal test set, choosing from the given test paths, that satisfies all-defs coverage with respect to \(x\). (Direct tours only.) If the given test paths do not suffice, add a minimal set of paths.

(e) [5 points]: List a minimal test set, choosing from the given test paths, that satisfies all-uses coverage with respect to \(x\). (Direct tours only.) If the given test paths do not suffice, add a minimal set of paths.

(f) [5 points]: List a minimal test set, choosing from the given test paths, that satisfies all-du-paths coverage with respect to \(x\). (Direct tours only.) If the given test paths do not suffice, add a minimal set.

Problem 4 [35 points]: Based on Exercise 5 after Section 2.3 (pages 61-62, Chapter 2). (This is related to Problem 4 in MP1.) Consider the pattern matching code (Figure 2.21, page 56 in the book) and the following three test cases:

- subject = “brown owl”, pattern = “wl”, expected output = 7
- subject = “brown fox”, pattern = “dog”, expected output = -1
- subject = “fox”, pattern = “brown”, expected output = -1

Write answers in a file called prob4.txt in your mp2 directory.

(a) [5 points]: Find the actual test path followed by each test case. (Hint: You can run your MP1 code.)

(b) [5 points]: For each test path, list the du-paths from the sets du(10, isSub), du(2, isPat), du(5, isPat), and du(8, isPat) that the test path tours. Show again your results as a table in the format given in SVN, with both rows and columns ordered lexicographically.

(c) [5 points]: Explain why the du-path [5, 6, 10, 3, 4] cannot be toured by any test path. This helps to check your answer for part (b) and to find answers for the next parts.

(d) [5 points]: Add tests to complete coverage of the (feasible) du-paths that are uncovered in part (b). Select tests from the table at the end of Section 2.3 (pages 59-60).

(e) [5 points]: From the tests in part (d), find a minimal set of tests that achieves all-defs coverage with respect to the variable isPat. (Note: do not confuse the iPat variable with isPat.)

(f) [5 points]: From the tests in part (d), find a minimal set of tests that achieves all-uses coverage with respect to the variable isPat.

(g) [5 points]: Is there any difference between all-uses coverage and all-du-paths coverage with respect to the isPat variable in the pat method?