Spring 2012 Course: CS 498, Section DM

Software Testing

Problem Set 2
Assigned: March 2, 2012
Due: Thursday, March 8, 2012 (at 12:30pm, beginning of lecture)

This problem set covers the material from Chapter 2 of the textbook. There are five 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 subproblems labeled **[MP]**, commit your code into SVN. Since you need to draw graphs for a few problems, please bring your (hand-written or printed) solutions to the non-MP problems to the lecture. **Important:** do not collaborate on any of these problems, especially the MP parts! If you have questions, please feel free to schedule a meeting with Darko or email him the questions.

Problem 1 [4 points]: Based on Exercise **1** after Section **2.2.1** (page 42, Chapter 2). There are several ways to define node and edge coverage (see slides 11-13 from Ch2-1-2 or Section 2.2.1 in the book). While for node coverage there were both a longer, more precise definition and an abbreviated definition, for edge coverage there was only an abbreviated definition. Write a longer, more precise definition for edge coverage, paying attention to use proper terms for path coverage.

Problem 2 [32 points]: Based on Exercise **5** after Section **2.2.1** (page 43, Chapter 2), with an additional part (h). Consider the graph defined by the following sets: N = {1, 2, 3, 4, 5, 6, 7}, N_0 = {1}, N_f = {7}, E = {(1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)}. Also consider the following (candidate) test paths: t_0 = [1, 2, 4, 5, 6, 1, 7] and t_1 = [1, 2, 3, 2, 4, 6, 1, 7].
(a) [4 points]: Draw the graph.
(b) [4 points]: List the test requirements for edge-pair coverage.
(c) [4 points]: Does the given set of test paths satisfy edge-pair coverage? If not, identify what is missing.
(d) [4 points]: Consider the simple path [3, 2, 4, 5, 6] and test path [1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]. Does the test path tour the simple path directly or with a sidetrip? If with a sidetrip, identify it.
(e) [4 points]: List the test requirements for node, edge, and prime path coverages on the graph.
(f) [4 points]: List test paths that achieve node coverage but not edge coverage on the graph.
(g) [4 points]: List test paths that achieve edge coverage but not prime path coverage on the graph.
(h) [4 points]: Reconsider the simple path [3, 2, 4, 5, 6] and test path [1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]. Does the test path tour the simple path with a detour? Explain why or why not.

Problem 3 [24 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}, N_0 = {0}, N_f = {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}, t_1 = [0, 1, 7], t_2= [0, 1, 2, 4, 6, 1, 7], t_3 = [0, 1, 2, 4, 5, 6, 1, 7], t_4 = [0, 1, 2, 3, 2, 4, 6, 1, 7], t_5 = [0, 1, 2, 3, 2, 3, 2, 4, 5, 6, 1, 7], t_6 = [0, 1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7].
(a) [4 points]: Draw the graph. (Consider if you can combine it with the previous graph.)
(b) [4 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) [4 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 where rows are test paths and columns are du-paths, with both rows and columns ordered lexicographically.
(d) [4 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 mimimal set of paths.

(e) [4 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 mimimal set of paths.
(f) [4 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 mimimal set.

Problem 4 [40 points]: Based on Exercise **5** after Section **2.3** (pages 61-62, Chapter 2). Consider the pattern matching code (Figure 2.21, page 56 in the book), committed into the **problem4** directory in SVN so that you can modify it there, and consider 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

(x) [4 points]: [MP] Write these three test cases as JUnit tests in **TestPatTest**. You should minimize your copy-paste by having a helper method that takes two input strings and the expected output, calls the **pat** method, and checks the actual output. Then, each test should be a simple call to this helper method. Style will count for grading this part. If you have questions or want feedback, please ask Darko in advance.
(y) [4 points]: [MP] Instrument the pattern matching code so that it computes and prints the executed test path when run. (The node ids in the code match the graph in Figure 2.13, page 47.) Hint: it can be better *not* to print the path throughout the run but only to compute it during the run and print at the end.
(z*) [4 points]: [MP] Implement in **TestPatCheck** a method that can check if the program run with given inputs tours a given path. (* Hard question.)
(a) [4 points]: Find the actual test path followed by each test case.
(b) [4 points]: For each test path, list the du-paths from the sets du(10, iSub), du(2, isPat), du(5, isPat), and du(8, isPat) that the test path tours. Show again your results as a table where rows are test paths and columns are du-paths, with both rows and columns ordered lexicographically.
(c) [4 points]: Explain why the du-path [5, 6, 10, 3, 4] cannot be toured by any test path.
(d) [4 points]: Add tests to complete coverage of the (feasible) du-paths that are uncovered in part (b). You can select tests from the table at the end of Section 2.3 (pages 59-60) or write your own tests.
(e) [4 points]: From the tests in part (e), 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) [4 points]: From the tests in part (e), find a minimal set of tests that achieves **all-uses** coverage with respect to the variable **isPat**.
(g) [4 points]: Is there any difference between all-uses coverage and all-du-paths coverage with respect to the **isPat** variable in the **pat** method?

Problem 5 [20 points]: Consider the Java code in the **problem5** directory in SVN.
(a) [4 points]: Draw the inheritance graph for this code. Explain your nodes and edges.
(b) [4 points]: Draw the call graph for this code. Explain your rationale for nodes and edges.
(c) [4 points]: [MP] Write two JUnit tests for **main**. What level of method and call coverage do your tests achieve? Could they achieve 100%? Why yes or why not?
(d) [4 points]: Describe how you interpret the notion of def-use coverage for object-oriented language such as Java. What are "defs" for an assignment "**o.f = null**": is only the field "**o.f**" being modified or the entire object "**o**"? What if we are writing to some object variable "**o**" but reading from another object variable "**p**": could those two accesses constitute a def-use pair? What about accesses to method local variables of primitive type, say writing to "**x**" and reading from "**y**": could those two accesses constitute a def-use pair? What if we are invoking methods on some object, say "**o.m()**": should we consider it to be a def and/or use of "**o**"? Based on your notions of defs and uses, write at least 4 test cases for one of your refactorings to check whether it properly preserves defs and uses from the input program.
(e) [4 points]: Consider testing, say, 1000 lines of code. How much would you charge for achieving 80% statement coverage? (You can give an absolute amount in dollars or relative to something else.) How much (more) for 90%? How much (more) for 100% (all feasible statements, showing that others are infeasible)? How much (more) for 100% prime path coverage? How much (more) for 100% all-du-paths?