Software Testing, Quality Assurance & Maintenance (ECE453/CS447/SE465): Midterm

February 17, 2017

This open-book midterm has 5 questions and 80 points. Answer the questions in your answer book. You may consult any printed material (books, notes, etc).

Question 1: Test Design (10 points)

I've included below some test code from bukkit's `BukkitMirrorTest`. Split this test into well-designed independent subtests. I recommend 5 tests; use your judgment to decide what the subtests should be. Write out the valid JUnit test cases after splitting. You may write e.g. "// L4" to indicate that you are including line 4 from the provided code. Write a sentence explaining the benefits of splitting the test.

```
1  @Test
2  public void bunchOfTests() throws Throwable {
3      assertThat(Modifier.isStatic(bukkit.getModifiers()), is(true));
4      assertThat(bukkit.isAnnotationPresent(Deprecated.class), is(server.isAnnotationPresent(Deprecated.class)));
5      assertThat(bukkit.getReturnType(), is((Object) server.getReturnType()));
6      assertThat(bukkit.getGenericReturnType(), is(server.getGenericReturnType()));
7      assertThat(bukkit.getGenericParameterTypes(), is(server.getGenericParameterTypes()));
8      assertThat(bukkit.getGenericExceptionTypes(), is(server.getGenericExceptionTypes()));
9  }
```

Question 2: Fuzzing (10 points)

```
1  float computeAngle(int x, int y, int z) {
2    if (x * x + y * y != z * z) {
3      throw new IllegalArgumentException("bad triangle");
4    }
5    return Math.atan2(y, x);
6  }
```

- (5 points) Write a Java function `naiveComputeAngleTest()` that generates a completely-pseudorandom input triple (x, y, z) for this method and calls it. Would `naiveComputeAngleTest()` give you good insight into what `computeAngle()` does? Explain your answer.
- (5 points) Write a Java function `smartComputeAngleTest()` that randomly generates an input that never triggers an `IllegalArgumentException`. You may use `Math.sqrt()`, which returns a double.

(To get a pseudorandom number: `Random r = new Random(); i = r.nextInt();`)

Question 3: Short-circuit evaluation (15 points)

Here is a simplified version of code from Assignment 1.

```
1   String buildCommand(String formatString) {
2     // returns the index at which '$' appears in formatString or -1 if not present
3     int index = formatString.indexOf("$");
4     while (index != -1) {
5       if (index > 0 && formatString.charAt(index - 1) == '!') {
6         // removes the '$' at index from formatString
7         formatString = formatString.substring(0, index - 1) + formatString.substring(
              index);
8         index = formatString.indexOf("$", index);
9         continue;
10      } else {
11        ;
12      }
13      index = formatString.indexOf("$", index + 1);
14    }
15  }
```

- (5 points) Draw a control-flow graph for this method. Nodes as basic blocks, labelled with line numbers. Explicitly write boolean conditions. (Be careful with the short-circuit evaluation.) I had 7 nodes.
- (10 points) Test suite `"$"`, `"!$"` achieves 100% statement coverage but jacoco says "1 branches out of 4 missed" at the `if` statement. Describe the missing branch and explain why it is missing (for 5 points) and write down a test case that achieves 100% branch coverage (another 5 points). (Warning: because I removed code, this behaves slightly differently than on the assignment).

Question 4: Statement and Branch Coverage (20 points)

```
1  public static List<String> wrap(String input, int line_length) {
2    List<String> rv = new ArrayList<String>();
3    int last_break = -1, last_space = 0;
4
5    for (int i = 0;
6         i < input.length();
7         i++) {
8      if (input.charAt(i) == ' ') {
9        last_space = i;
10     }
11
12     if (i - last_break > line_length) {
13       rv.add(input.substring(last_break + 1, last_space));
14       last_break = i;
15     }
16   }
17   if (last_space >= last_break + 1) {
18     rv.add(input.substring(last_break + 1, last_space));
19   }
20   return rv;
21 }
```

(a) (5 points) Draw a minimal-node (9) control-flow graph for this method, with nodes as basic blocks. Label the nodes with the line numbers on the left.

(b) (10 points) Consider `input = "one three two ", line_length = 9`. Argue that this input achieves statement coverage. You may use the fact that the output is:

```
$ java Justify 9 "one three two "
one three
two
```

(c) (5 points) Does my input also achieve branch coverage? Why or why not? (If so, say why; if not, say specifically which branch is not covered).

Question 5: Understanding Mutation (25 points)

(a) (10 points) Write down a non-trivial mutant of `wrap()` that is killed by **my** test case (from (1b)). (Write the line number that you are changing and what you want it to say, e.g.:)

```
Line 5: for (int i = -1;
```

Explain why my test case kills the mutant (show outputs). Also show why the mutant is non-trivial; that is, find a non-empty input on which the mutant and the original do not differ.

(b) (5 points) Write down a test case for `wrap()` (from Question 4) that reveals a bug in the original program. Include actual and expected output. (Contract: the concatenation of the strings returned from `wrap()` must equal `input`.) You do not have to identify or fix the bug. Clearly, this test case does not improve statement coverage beyond the input from (4b).

(c) (10 points) Write down a different non-trivial mutant of `wrap()` that is killed by **your** test case (from part b) and explain why your test case kills the mutant (show outputs). Again explain why the mutant is non-trivial; that is, find a non-trivial input on which the mutant and the original do not differ.

Observe that you've used mutation to improve your test suite from what it was in (4b). You generated a mutant, and the test case from (4b) killed the mutant from (4c). On the other hand, both statement and branch coverage don't get you anything more.