Name: _____

Student Number: _____

SFWRENG 3S03

Software Testing – Sample Solutions

Midterm Test, Winter 2019-20, 50 minutes

Instructions: this test consists of three questions on four pages. Answer all three questions in the space provided. You may use one 8.5x11 cheat sheet (one-sided only) and no other aids.

Question 1 [15 marks]

a) **[3 marks]** Give three reasons (using no more than one sentence each) why the later a fault is found in the software development process, the harder it may be to fix.

Lots of reasonable answers here, including:

- *New work may build on a flawed foundation, so correcting original problems may require significant rework.*
- *If software has been distributed widely, distributing a fix may be expensive.*
- *Developers may have forgotten original context.*
- *More code generally is produced as the project progresses implying that it will be harder to track down the fault.*

b) **[4 marks]** It is cheaper and faster to fix known bugs before you write new code. Why? Give three reasons (using no more than one sentence each). Give reasons that are as different from one another as possible.

Plenty of reasonable answers here too, including:

- *You are familiar with the code now and it will be harder to find the bug later as your familiarity with the code surrounding the bug decreases.*
- *Later code may depend on this code.*
- *It's hard to predict how long overall bug fixing is going to take; waiting until later may promote greater schedule slippage.*
- *An overfull bug database is demoralizing.*
- *Avoiding feature creep.*

c) **[3 marks]** After you find a bug but before fixing it, you should create a test case for it. Give three reasons (using no more than one sentence each) why this is a good idea. Give reasons that are as distinct as possible.

- *You ensure that your fix solves the problem.*
- *It helps you understand the bug and defines desired behaviour.*
- *Tells you when you are done.*
- *Helps produce a test suite with good tests.*
- *Protects against reversions that reintroduce the bug.*

Note: just saying "This lets us use TDD" is not a good answer, unless you actually explain why using TDD is beneficial.

(d) **[3 marks]** Your test suite achieves 55% statement coverage and the tests all pass. Without any further information, what is one fact that you can conclude about the statements that are reported as covered?

55% of statements were executed by test cases. These statements did not throw an uncaught exception. (Saying "statements are reachable" is true but unremarkable/uninteresting.

(e) **[2 marks]** Consider a reachable fault F that infects the program state, propagating to output. Say you delete the line of code containing F. Would you still expect a failure? Where is the fault now?

I'd still expect a fault. Many people said it would be at the same line, or immediately afterwards, but you can't really justify that. What you can argue is that there will probably be a failure either at the previous write to the same state, or at the next access to the same state.

2. **[20 marks]** You have been provided with a class called Square written in Java. Your job now is to test the methods contained in it using JUnit. Answer the following questions.

Most people did really well on this question, as I was liberal in interpreting answers.

(a) **[3 marks]** What three steps do you follow to create tests that run with JUnit ?

- *Create a test class*
- *Import JUnit headers*
- *Implement tests and annotate using appropriate JUnit annotations*

(I was pretty generous in marking this question, anything close that showed you understood the basic functionality of JUnit got marks.)

(b) **[3 marks]** How do you validate conditions in your tests?

Use an assert method.

(c) **[3 marks]** What feedback does JUnit provide when tests fail?

- *Name of failed test and its class.*
- *A message of the form "expected <value1> but was <value2>" or similar.*
- *Summary of how many tests were run with the number of fails.*

(d) **[3 marks]** What feedback does JUnit provide when tests succeed?

- *Time to execute tests*
- *"OK" message*
- *Number of tests run.*

(e) **[3 marks]** Give one advantage of using JUnit for testing.

Tests are easily integrated and can be run automatically for regression testing (I accepted lots of other answers here).

(f) **[3 marks]** Give one disadvantage of using JUnit for testing.

It's only suitable for unit testing (not integration testing etc). I did not accept "you have to use another tool" or "you have to learn something new" as these don't answer the question directly.

(g) **[2 marks]** State one mechanism in JUnit that can be used to "bundle together" multiple test cases.

Test suite (I also accepted test set).

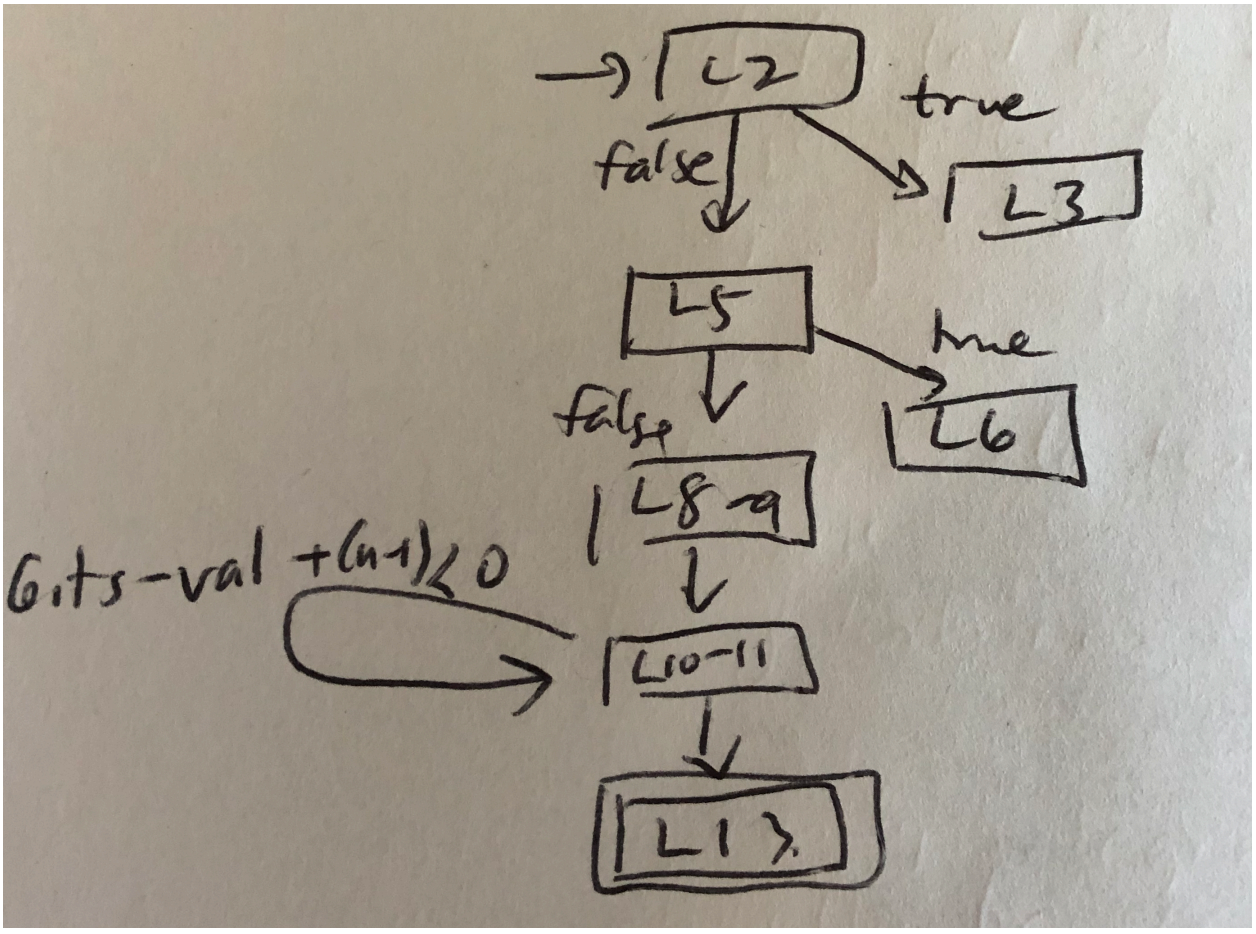
3. **[15 marks]** Here's the implementation of Java's `java.util.random.Random.nextInt` function.

```
public int nextInt(int n) {
  if (n <=0)
     throw new IllegalArgumentException("n must be positive");

  if((n&-n)==n) //i.e.,nisapowerof2
     return (int)((n * (long)next(31)) >> 31);

  int bits, val;
  do {
     bits = next(31);
     val = bits % n;
  } while(bits - val + (n-1) < 0);
  return val;
}
```

(a) **[5 Marks]** Draw the control flow graph for `nextInt`.



Almost everyone got full marks for this, unless you drew an arrow from my L3 to L5 (i.e., after the exception normal control flow resumed).

(b) **[5 Marks]** Since the utility function `next()` returns pseudorandom bits, discuss any difficulties that may arise in ensuring 100% statement coverage for `nextInt()`**.** How can you ensure 100% statement coverage?

This was tricky and I was generous in marks that showed indications of being on the right track. The reachability of lines 1-5 depend on n. It's a do-loop so the body is always executed as soon as line 6 is reached. Line 10 is unconditionally executed as soon as line 6 is. So, you actually have no problems ensuring 100% statement coverage as long as you pick the right n. (I was looking for an analysis like this.) You also need to assume that nextInt() terminates. So you could choose an n strictly positive to reach line 3, an n strictly negative to reach line 2, then an n that's a power of 2, then an n that is not a power of 2. That should cover things.

(c) **[5 Marks]** What about branch coverage? What are the difficulties and how can you ensure 100% branch coverage? Assume that you can change any part of Random's state and call nextInt how you'd like; you may not change `nextInt` itself.

This is also tricky – possibly a bit harder than (b). I found that people who clearly understood what was happening in (b) had little difficulty with (c). Again, I was generous with marking.

The key in (c) is to ensure you go around the loop once, as well as execute outside of the loop once. So how do you guarantee execution goes around the loop when you don't know what next() will return? The solution I came up with was to force next(31) to be smaller than val-(n-1). I accepted a number of other suggestions as well, but this seemed to be the simplest one.