SE 465 Midterm Examination
University of Waterloo
Term: Winter Year: 2019

Date: Thursday, February 28, 2019
Time: 18:30 – 20:00 (90 minutes)
Instructors: Patrick Lam
Lecture Section: 001
Exam Type: Open book, open notes, calculators with no communications capabilities
Number of exam pages (includes cover page): 8

Please Print
Last Name ____________________
First Name ____________________
UWaterloo ID # ____________________
Username ____________________

Question	1	2	3	4	5	Total
Points	15	10	15	10	10	60

Instructions

1. Turn off all communication devices. Communication devices must be stored with your personal items for the duration of the exam. Taking a communication device to a washroom break during this examination is not allowed and will be considered an academic offence.
2. I shuffled the order of the questions from what I said in class for better page breaks.
3. The exam lasts **90** minutes and there are 60 marks.
4. Verify that your name and student ID number is on the cover page.
5. If you feel like you need to ask a question, know that the most likely answer is "Read the Question". No questions are permitted. If you find that a question requires clarification, proceed by clearly stating any reasonable assumptions necessary to complete the question. If your assumptions are reasonable, they will be taken into account during grading.
6. Answer the questions in the spaces provided. If you require additional space to answer a question, please use the second last page and refer to this page in your solutions. You may tear off the last page to use for rough work.
7. Do not write on the Crowdmark QR code at the top of each page.
8. Use a dark pencil or pen for your work.

1 Mutation Testing [10 Marks]

Consider this function `Location.add()`. Write down a (non-trivial, non-stillborn, non-equivalent) mutant to the `Location.add()` function (indicating your change) as well as a test case that kills the mutant. (Use JUnit-like syntax; we don't care about the details). Provide the expected and actual output for your testcase with the original function as well as for the mutant.

Assume that you can create a `Location` object with an expression like `new Location(0.0, 4.65, 3.50, W);`. Describe a `Location` with a tuple like $\langle 0.0, 4.65, 3.50, W \rangle$.

```
public Location(double x, double y, double z, Object world) {
    this.x = x; this.y = y; this.z = z; this.world = world;
}

// credit: org.bukkit.Location.java
/**
 * Adds the location by [sic] another.
 *
 * @param vec The other location
 * @return the same location
 * @throws IllegalArgumentException for differing worlds
 */
public Location add(Location vec) {
    if (vec == null || vec.getWorld() != getWorld()) {
        throw new IllegalArgumentException("Cannot␣add␣Locations␣of␣differing␣
            worlds");
    }

    x += vec.x;
    y += vec.y;
    z += vec.z;
    return this;
}
```

(Extra space for Q1 answers)

2 Branch and Statement Coverage [15 Marks]

Here's the implementation of Java's `java.util.random.Random.nextInt` function. (5 Marks) Draw the control-flow graph for `nextInt`. (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? (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.

```
// credit: java.util.Random

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

    if ((n & -n) == n)  // i.e., n is a power of 2
        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;
}
```

(Extra space for Q2 answers)

3 Input Generation [10 Marks]

You have a test suite which contains the following set of calls to a REST API (`https://reqres.in`, specifically). I've put the POST and PUT payloads in braces after the URL.

```
GET /api/users
GET /api/users/2
POST /api/users { "name": "plam", "job": "SE director" }
PUT /api/users/2 { "name": "plam", "commute": "bicycle" }
DELETE /api/users/2
POST /api/register { "email": "patrick.lam@uwaterloo.ca", "password": "password1" }
POST /api/login { "email": "patrick.lam@uwaterloo.ca", "password": "password2" }
POST /api/logout { "token": "QpwL5tke4Pnpja7X" }
```

(2 Marks) Provide an additional input that looks correct and an input that looks incorrect. (6 Marks) Provide pseudocode that programmatically generates correct inputs, including at least 4 of the calls above. Your pseudocode may call primitives that randomly generate an integer or string. (2 Marks) Describe how to programmatically generate incorrect inputs. (A good way of describing is by modifying the pseudocode).

4 Finite State Machines [10 Marks]

(4 Marks) Propose a Finite State Machine for the REST API in the previous question. The FSM should abstract away from the details in the example requests. It should also include an appropriate cycle. (There are multiple possible correct answers). (6 Marks) Write down the test requirements for Simple Round Trip Coverage and a test suite which satisfies these test requirements. Are there additional test requirements for Complete Round Trip Coverage in your FSM?

5 Short Answer [15 Marks total]

Answer these questions using at most three sentences. Each question is worth 3 marks.

(a) Give an example of a bug that is best detected by exploratory testing.

(b) 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?

(c) For the same test suite as above, what can you conclude about the 45% of statements that aren't covered?

(d) You are developing a test suite for a web page that is to be translated into multiple languages. How do you introduce a level of abstraction into your tests?

(e) 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?