COMP 587
Spring 2020

Midterm Review with Answers

The following review, **in conjunction with your assignments and the in-class handouts**, are intended to be comprehensive for the midterm exam. The questions below are representative of the sort of questions you'll see on the exam.

Test Planning

1.) Name one advantage of Attributes, Components, Capabilities (ACC) over traditional test planning.

Possible answers:
- Takes less time to develop a test plan
- Generally resistant to requirement changes (focuses on high-level software aspects as opposed to nitty-gritty details)
- Easier to keep up to date

2.) Name one advantage of traditional test planning over ACC.

Possible answers:
- Much more detailed
- Requires you to formulate much of the system upfront (potentially a disadvantage)
- More common, and therefore may be perceived as more trustworthy

3.) A startup developing a mobile app for rapid frozen yogurt delivery is interested in formulating a test plan for their software. Should traditional test planning or ACC-based planning be preferred? Why or why not?

Arguably, ACC-based test planning should be preferred. The application is not safety-critical, and requirements are likely to change as the company evolves.

4.) A space agency is developing a real-time operating system for a lunar rover. Should traditional test planning or ACC-based planning be preferred? Why or why not?

Traditional test planning likely should be preferred. The intended usage will require requirements to generally be placed upfront. While this is not safety-critical, the associated costs are expected to be very high, so emphasis should be placed on correctness over agility.

5.) Consider the following partial ACC plan for an app to find someone to walk a dog immediately:

Attributes:
Intuitive: the interface is easy to use
Fast: walkers can be found quickly
Robust: the app does not crash and isn't prone to glitches

Components:
Map: users can see a map of available dog walkers nearby
Reservation Creator: users can reserve a walker
Reservation Manager: users can see past, present, and future walker reservations
Configuration: users can configure payment information and basic app behaviors

Capabilities:
Reservation Creator is Fast: walkers can be found within 20 seconds
Reservation Creator is Intuitive: users can reserve a walker with one press
Map is Intuitive: the map's color scheme is easy to interpret

There are some potential issues with this test plan. Provide some feedback regarding how realistic this plan is.

Some potential issues:

- "Fast" seems to depend on the surrounding market of available dog walkers. This isn't really about the software itself.
- "Robust" is a sort of catch-all for problems. Software generally should be robust anyway.
- "Intuitive" is good, though testing related capabilities may be difficult, as "ease of use" is somewhat subjective.
- "Reservation Creator is Fast": absolutely depends on the market, unless finding walkers can be non-immediate
- "Reservation Creator is Intuitive": it makes sense to test reservation creation, but it either is one press or it isn't
- "Map is Intuitive": the color scheme being easy to interpret is subjective. At the very least, this seems to require a user study; it's not possible to automatically test this as written

Writing Unit Tests

The questions below use Python. You may not be familiar with Python; that's ok. For our purposes, I will not take off for syntactic things. You can even write pseudocode, as long as it's detailed enough.

6.) Consider the following Python code:

```
def cap(value):
  if value < 0:
    return -1
  elif value == 0:
    return 0
  else:
    assert value > 0
    return 1
```

Using `assert`, write some tests which will collectively get 100% line coverage of `cap`.

```
assert cap(-5) == -1
assert cap(0) == 0
assert cap(5) == 1
```

7.) Consider the following Python code and corresponding tests:

```
def decToZero(value, limit):
  while value > 0 and limit > 0:
    value -= 1
    limit -= 1
  return value

assert decToZero(10, 100) == 0
assert decToZero(0, 10) == 0
```

7.a.) While the above tests get 100% line coverage (and maybe even 100% branch coverage depending on what we consider a branch), they arguably don't cover everything. What's missing?

There is never a case where `limit` is not greater than 0. If `A = value > 0` and `B = limit > 0`, we cover `A && B` and `!A && <anything>`, but not `A && !B`.

7.b.) Write additional tests to cover the missing behavior identified above.

```
assert decToZero(100, 10) == 90
```

8.) How can tests be used as a specification of a system?

Tests collectively say what the system should output given various input, without actually implementing the system itself. Specifications serve this same purpose.

9.) It's possible to get 100% coverage without ever doing any meaningful testing. Explain a scenario under which this can happen.

Your tests cover all your code, but no assertions are ever made in the test suite.

10.) You have a 100 line program with 1,000 unique tests. All these tests pass with 100% coverage on every metric. Additionally, you use mutation testing/analysis, and are able to show that all mutants are killed. Is the program correct? Explain.

The program might be correct, and with numbers like these it probably is correct, but it's not guaranteed without more information. In theory, if the program has an infinite number of inputs, tests can only show a bug exists, not prove a program correct.

11.) Mutation testing/analysis is generally considered very expensive. Why?

Possible answers:

- There are a potentially infinite number of mutants to try
- Each mutant may require recompilation of the whole system
- The whole test suite may (and often will) need to be run on each mutant

Mocking and Testability

12.) Consider the following Java code, taken from a larger system:

```
public class Something {
  public int x = 0;

  public void doOperation(int input) {
    x += input;
    System.out.println("Output: " + input);
    OtherThing.value = x;
  }
}
```

12.a.) Is `doOperation` testable? Why or why not?

Not very. This code performs output, and there isn't an easy way to automatically check this output for correctness. Additionally, this appears to be modifying some `static` state in the `OtherThing` class, so it's difficult to test doOperation independently of everything else.

12.b.) What might you do to make `doOperation` more testable?

Instead of using side-effects and mutable state to do its job, `doOperation` could be refactored to return something instead. The output destination could be mocked instead of hard-coding `System.out`. The state change involving `OtherThing.value` should be encapsulated within `Something`, or at least should not involve changing a `static` variable somewhere.

Automated Testing

13.) You are developing a system which is to undergo verification later. That is, it will later be *proven* correct. With this in mind, is it worthwhile to test the system while under development? Why or why not?

Testing is still worthwhile. Verification is time-consuming and expensive. Counterintuitively, you want to be reasonably sure the system is correct before spending the effort to verify it. Correcting mistakes is generally much easier in the development/testing phase than it is to correct them in the verification phase.

14.) What's the difference between random and bounded-exhaustive testing?

Random testing will randomly select test inputs within some range, whereas bounded-exhaustive testing will run all test inputs within a range.

15.) What is the small scope hypothesis? Why is it only a hypothesis?

That most bugs can be found with relatively small inputs. Exactly what "most" and "relatively small" mean are system-dependent. There is some evidence that it's true, but this is very dependent on the particular system.

16.) When is random testing most appropriate?

When the system has a prohibitively large input space, and the scope size in the small scope hypothesis is still too large for bounded-exhaustive testing.

17.) When is bounded-exhaustive testing most appropriate?

The input space for the system is relatively small, or at least the scope size in the small scope hypothesis is expected to be relatively small.

18.) You're developing a system which is to be tested using automated testing techniques. Should you bother with unit tests? Why or why not?

Unit tests are still valuable. Unit tests are generally much easier to write, less expensive to run, easier to interpret, and still effective at finding many common issues.

19.) You have a system which is composed of 1,000 lines of code. Using automated testing techniques, you've subjected the system to 1,000,000 tests, and it has passed all of them. How confident are you that the system is correct? Is there any additional information you'd need to assess this?

It's tempting to say very confident, but it depends on the specifics. For example, if you're testing a calculator, it's easy to generate millions of tests like:

1 + 1
1 + 2
1 + 3...

...which clearly isn't testing much. Test case generators can have blind spots. Coverage and mutation testing/analysis would tell you more.

20.) Consider the Java code below:

```
public class Node {
  public static Random rand = new Random();
  public final int value;
  public final Node rest;

  public Node(final int value, final Node rest) {
    this.value = value;
    this.rest = rest;
  }

  public static Node genNode() {
    if (rand.nextBoolean()) {
      return null;
    } else {
      return new Node(rand.nextInt(), genNode());
    }
  }
}
```

Node represents a single node in a singly-linked list. null is intended to represent a leaf node. genNode is used for randomly generating nodes for testing purposes.

20.a.) Will `genNode` generate diverse tests? Why or why not?

No. Half the tests are expected to be single leaf nodes. There is only low probability of generating a long list.

20.b) What changes could you make to `genNode` to make it generate more diverse tests?

Possible answers (among others):

- Augment it with minimum and maximum list lengths, and generate a number of nodes equal to some randomly-chosen length.
- Use a bounded-exhaustive approach instead of a random one in order to generate all lists up to a given length.
- Start with a high probability of generating a non-leaf node, and decrease this probability as the length gets longer.