Software Testing, Quality Assurance & Maintenance (ECE453/CS447/CS647/SE465): Final Solutions

Question 1 (10 points): Logic Coverage

First part: this was on the project, so it's a bit of a giveaway.

```
void foo(boolean a, boolean b) {
  if (a && b) { ... }
}
```

Now $p = a \wedge b$; PC imposes two test requirements, namely p : true and p : false. We can meet PC by setting a to true and false respectively. CACC imposes three test requirements: a determines p, then we need p to be both true and false; and b determines p, with p both true and false. We can meet CACC with the three tests TT, FT and TF.

For the second part, make active clause coverage impossible, for instance $p = a \wedge \neg a \wedge b$.

Question 2 (10 points): Concurrency

The access to `n` is unprotected by locks, so that the two concurrent calls to `add()` may conspire to put `n` in an inconsistent state. You can find this problem by inspecting the code, or by using race detection tools. The fix is to wrap the calls to `add()` in `synchronized()` blocks on the same lock.

Question 3 (10 points): Mutation

Here is an example of a DTC mutation.

```
public void foo() {
  List l = new List(); // mutate to: Object l = new List();
}
```

The behaviour of Java code does not change depending on the declared types. The declared types are only used to check types at compile time. Only the actual types (`List`, in this case) affect the run-time behaviour of the Java code. So either the mutant is equivalent (for instance if you change `List` to `LinkedList`, a more specific type) or stillborn (if you change to a more general type and rely on the more specific type later on.)

Question 4 (10 points): Mutation

The test harness has to create a `Room` and modify the underlying array that it passed to the `Room` in its constructor, like this:

```
public static void main(String[] argv) {
  float[][] pts = new float[][] {{ 3, 5 }};
  Room r = new Room(pts);

  pts[0][0] = 5;
  System.out.println(r.getPoints()[0][0]);
}
```

If you replace `Room` by `RoomMutant` in this test harness, then the behaviour changes, so the harness strongly kills the mutant.

To make the harness more generic, you should change `Room` to an interface, say `IRoom`, and make the harness take an `IRoom` as a parameter. You could also pass a factory for `IRoom`s to the hasness.

Question 5 (10 points): Input Space Coverage

The two errors in `isPrime` are: 1) the loop condition should be ¡=, not ¡; and 2) even after 1 is fixed, it returns the negation of the value that it should return (false swapped with true).

Given the method name, one would expect `isPrime` to carry out primality testing. So a good partitioning is 1, primes, composites. (Technically, 1 is neither prime nor composite, but it's fine if you combine 1 and composites into one partition). Such a partitioning earns full marks. Better partitionings would refine primes into even primes and odd primes, and composites into square composites and non-square composites.

Question 6 (10 points): Input Space Coverage

Suppose that test set T exhaustively covers the input space of a method `foo()`. Assume that `foo()` does not read any state (fields, files). Further assume that `foo()` has no unreachable code. (5) Explain why or why not T ensures prime path coverage of `foo()`. (5) Explain why or why not T ensures complete path coverage of `foo()`. (It suffices to give an example to explain why not.)

Note that if `foo()` had unreachable code, then T would not achieve prime path coverage and hence not complete path coverage either, since there would be no way to reach the unreachable code.

Assuming that `foo()` has no unreachable code, it is still not true that all prime paths are feasible. Here's one example:

```
void foo(int n) {
  if (n == 5)
    System.out.println("5"); // 1
  else
    System.out.println("not 5"); // 2

  if (n != 5)
    System.out.println("NOT FIVE"); // 3
  else
    System.out.println("FIVE"); // 4
}
```

The prime path including both statements 1 and 3 is infeasible, since the conditions for reaching 1 and 3 contradict each other. It follows directly that complete path coverage is also not guaranteed.

Question 7 (10 points): Graph Coverage

For the `if` statements, you have to include a case where `q` is 1 and a case where `q` is 2. The prime paths end at the node `k < i` created by the if statement. There are infeasible prime paths like the ones in Question 6; it's best to drop them.

Prime path coverage also includes the cycle covering the `for` loop as a prime path.

Finally, we need to visit both implementations of `m`.