Ciências ULisboa | Informática

VVS 2018/2019
Exam 1 – June 6, 2019
Duration: 3 hours

Consider the following SUT, a class implementing graph operations.

```
class AdjacencyListGraph<E extends Comparable<E>> {

    ArrayList<Vertex> verticies;

    private class Vertex {

      // the data on this vertex
      private E data;
      // its adjacencies
      private List<Vertex> adjacentVerticies;

      public boolean addAdjacentVertex(Vertex to) {
          for (Vertex v: adjacentVerticies)
              if (v.data.compareTo(to.data) == 0)
                  return false;  // the edge already exists
          return adjacentVerticies.add(to); // returns true
      }
    }

    public boolean addEdge(E from, E to) {
        Vertex fromV = null, toV = null;
        for (Vertex v: verticies) {
            if (from.compareTo(v.data)==0)
                fromV = v; // from vertex exists
            else if (to.compareTo(v.data)==0)
                toV = v;    // to vertex exists
            // if both nodes exist, stop searching
            if (fromV != null && toV != null) break;
        }
        if (fromV == null) {
            fromV = new Vertex(from);
            verticies.add(fromV);
        }
        if (toV == null) {
            toV = new Vertex(to);
            verticies.add(toV);
        }
        return fromV.addAdjacentVertex(toV);
    }

} // end class
```

Group 1

1. Draw the control flow graph (CFG) of method `addEdge()`. Assume method calls as simple commands.
2. List the requirements produced by edge coverage for `addEdge()`.

3. Write a set of JUnit tests that satisfy the previous requirements.
4. Identify the prime paths of the CFG.

Group 2

1. In the context of Input Space Partitioning (ISP) identify the SUT's input parameters.
2. Model the input domain by identifying two relevant characteristics and present their respective blocks. Choose the most appropriate approach (either interface-based or function-based) and justify your decision.
3. Present the requirements generated by using Pairwise Coverage Coverage. Underline those that are infeasible.

Group 3

Consider the following code:

```
public boolean removeEdge(E from, E to) {
    boolean found = false;
    Vertex fromV = null;

    for (Vertex v: verticies)
        if (from.compareTo(v.data) == 0 && !found) {
            fromV = v;
            found = true;
        }

    if (fromV == null)
       return false;
    return fromV.removeAdjacentVertex(to);
}
```

1. Using *Predicate Coverage*, list its requirements.
2. Build a table with the reachability logic formulas for each predicate.
3. Write JUnit tests to cover these requirements. If there are infeasible requirements, justify why is it so.
4. Compute the determination predicates for each predicate of method `removeEdge()`.

Group 4

Consider that you are working with two mutation operators: a math mutator that switches equality operators, and an assign mutator switching assigned expressions into nulls.

1. Present two mutations for each mutator for method `removeEdge`.
2. Write JUnit tests to kills all previous mutations.
3. Give an example of one of these mutations not being killed because some test was missing.
4. Explain what is a *weak kill*. Is it possible to have a weak kill for these mutations? If yes, give an example; if no, explain why.
5. What is the fundamental premise of mutation testing?