Software Testing: Preparatory work for tutorial on group task 3

Mutation Testing

The code below is part of a method in the `ConvexHull` class in the VMAP system. The following is a small fragment of a method in the `ConvexHull` class. For the purposes of this exercise you do not need to know the intended function of the method. The parameter p is a Vector of Point objects, `p.size()` is the size of the vector p, `(p.get(i)).x` is the x component of the i^{th} point appearing in p, similarly for `(p.get(i)).y`. This exercise continues the work you did on structural testing of code in the previous tutorial in order to measure the adequacy of the test sets you generated in the previous tutorial preparatory exercise.

```
Vector doGraham(Vector p) {
        int i,j,min,M;

        Point t;
        min = 0;

        // search for minimum:
        for(i=1; i < p.size(); ++i) {
            if( ((Point) p.get(i)).y <
                        ((Point) p.get(min)).y )
            {
                min = i;
            }
        }

        // continue along the values with same y component
        for(i=0; i < p.size(); ++i) {
            if(( ((Point) p.get(i)).y ==
                        ((Point) p.get(min)).y ) &&
                 (((Point) p.get(i)).x >
                        ((Point) p.get(min)).x ))
            {
                 min = i;
            }
        }
```

- **Prerequisites:** Review the lecture on Mutation Testing.
- **Preparation:** Review the code fragment drawn from the `doGraham` method above. If need be, check the documentation on the Vector class and any other Java documentation you might require.

Activity

Having considered this code fragment you should do the following preparatory activity, writing brief notes as described and submit them to the designated tutor. The tutorial session will review this work and make suggestions on how to tackle the third group task.

1. You will already have constructed test suites for statement and branch coverage for this code as part of preparatory exercise 2.

2. Construct three or four mutants of the above program. Each mutant should be derived from the original program using one application of one of the following rules (i.e. each mutant contains only one mutation):

   (a) **Changing constants:** a constant c in the program may be replaced by $c + 1$ or $c - 1$.

   (b) **One-off in relations:** $<$ may be replaced by $<=$ and vice versa; $>$ may be replaced by $>=$ and vice versa.

   (c) **Inverted relations:** $<$ or $<=$ may be replaced by $>$ or $>=$ and vice versa.

   (d) **Duplicate statements:** any statement s; can be replaced by s;s;

   (e) **Deleted statements:** you may delete any statement.

3. **Write down your mutants to send to the tutor.**

4. Check and see how many of your mutants are killed by each of your test suites.

5. If there is at least one mutant which is not killed in at least one of the test suites, try to strengthen the test suite to kill all the mutants.

6. If all your mutants are killed, try to generate a mutant which is not killed by the test suite using any of the above mutations.

7. **Write down any strengthened test suites, the mutants they kill and any new mutants you devise to avoid death by your test suites.**

8. **Submit all of your mutants and test suites to the tutor.**

9. The first exercise in the tutorial should be to test the other group's test suites with your mutants.