CS 520 In-class exercise 2

By Patrick Walsh, Matthew Lips

Due **October 26th, 11:59pm**

Questions

Question 1

> *What are the 2 best practices satisfied by the triangle project that make it easier to write unit tests and run them?*

The two best practices satisfied by the triangle project that make it easier to write unit tests is **Composite** and **white box testing**.

- Composite allows develpoers to plug and play with the function, rather than having to build an entire object for the triangle just to get the same outputs for testing
- white box testing allows developers to see inside the code and make testing like decision and statement testing better, but it makes the largest difference with mutation testing. Being able to see the source code like white box testing allows is required for mutation testing, and mutation testing allows for more rigorous tests of the software, ensuring no undefined behaviors sneak through.

Question 2

> *For the isTriangle class with the initial test suite, what is the statement (a.k.a. line) coverage percentage? The decision (a.k.a. branch) coverage percentage? The mutant detection rate?*

In the initial test suite for the isTriangle class has 65% initial line coverage, 53% initial decision coverage, and 0% mutant detection rate.

Question 3

> *Did your approach to writing unit tests differ between developing a coverage-adequate test suite and developing a mutation-adequate test suite? Briefly explain why or why not.*

Our approaches to writing unit tests between coverage-adequate suite and mutation-adequate suite had some inherent differences as each of us programmed on these files separately. One difference is in the amount of tests, since in the code coverage tests there are a small and finite amount of lines and branches, it does not take many tests to get to 100% code coverage. However, there are significantly more possiblilities of mutations in a program and so a higher number of test cases are needed to reach a high mutation

detection rate. Aside from this difference, there are also similarities in the types of tests being made because we are of course testing the same program. So in both test suites we test for inalid lengths, isosceles, equilateral, and equal a_b, b_c, a_c side lengths.

Question 4

> *Consider your mutation-adequate test suite and the triangle program. For any given program, why are some mutants not detectable?*

Some mutants were not detectable as they were solutions that would not make a difference inside of the program.

For example, one of the mutations is shown below.

```
   - [#  36] DDL isTriangle:
--------------------------------------------------------------------------------
  11:         SCALENE = 1
  12:         EQUILATERAL = 2
  13:         ISOSCELES = 3
  14:
- 15:     @staticmethod
- 16:     def classify(a, b, c):
+ 15:     def classify(\
+ 16:         a, b, c):
  17:         '''
  18:         This static method does the actual classification of a triangle, given the lengths
  19:         of its three sides.
  20:         '''
--------------------------------------------------------------------------------
[0.00303 s] survived
```

In this example, the \ character simply moves the classify parameters to the next line, with no changes to the flow of the program. Therefore, there are no detectable changes through mutation tests.

Question 5

> *What changes in the code coverage percentages and mutant detection rate did you observe when deleting (or commenting out) all assertions?*

When removing all assertions, there was a 100% survival rate of mutants.

Question 6

> *Create a definition of “test case redundancy” based on code coverage or mutation analysis. Given your definition of test case redundancy, are some of the test cases in your test suites redundant? Given your definition of test case redundancy, would you remove redundant test cases? Briefly explain why or why not.*

Test case redundancy is a test case that would increase the coverage or mutation detection rate if there was not already a similar test case hitting the same lines of code.

Although redundant test cases are inherently redundant, I would not remove the majority of them. This is under the possibility that developers update code in the program, and the possibility that some of these redundant test cases are not redundant.

Question 7

> *How many decision points did you find for the Control flow graph for normative cases (scalene triangle, equilateral triangle, and isosceles triangle) and exception cases (invalid sides and triangle inequality)? Did these findings help you to create a better test suite?*

Looking through the control flow graph, there were 10 normative cases for control flow, and 7 separate cases for exceptions, counting all possible routes using the decision points. These findings did in fact help create a better test suite as they helped create a top-level overview of the options the user had to flow through the program.

With the top-level control flow view, it is more clear to see the path of the data and how the outputs mesh together.