CS520

Anshumaan Chauhan & Hsin-Yu Wen

2 Best practices that make it easier to test

1. The use of enum in `isTriangle.py`

   In `isTriangle.py`, the enum `Type` allows us to write assertion lines as `self.assertEqual(actual_type, Triangle.Type.EQUILATERAL)`, which is readable than declaring it as a string type.

2. Modularity

   In the project, we have separate files for the class `isTriangle.py` and corresponding test class `test_triangle.py` makes it easier to interpret and debug the project.

Coverage rates

After running `bash test.sh`, we get the coverage rate for isTriangle.py file

- Initial Statement Coverage - 65%
- Initial Decision Converage - 53%
- Intital Mutant detection rate: 23.1%

  Detectable mutant / Total number of Mutants = 12 / 52 (12 mutant was killed. 40 mutants survived.)

Difference between developing Coverage versus Mutation Adequate Test Suites

The objective while developing a Test Suite for coverage is to design test cases, that cover each and every line of code (Statement) or covers all the decision points - possible outcomes of each decision point (Decision). Contrary to this, the objective in Mutation adequate Test Suite is to detect any mutations in the test suite.

Test writing approach for coverage is to design cases that goes through different decision paths and are able to execute all the possible code statements, whereas in mutation, we design test cases to find faults in the code logic that is introduced deliberately either in the initialization of the variables, or conditional checks, or returned values (mutations based on `isTriangle.py`).

Lastly, the main difference is the expectation, for coverage we want the output to have a high percentage of statement/decision coverage whereas for the mutation detection we want to kill as many mutants as possible - they both have nothing in common (although test cases for statement coverage can detect mutants and vice-versa with assertion statements).

Why are some mutants not detectable?

Some of the possible reasons why mutants are not detectable are as follows:

- The change was made in a statement that is never reachable during the execution of the test cases.
- The change made had no impact in the logic - for example simply adding an empty statement between two code segments.
- Inadequate test cases, which do not cover all the possible changes in the decision making steps and code segments will definitely lead to non detection of some mutations.

Impact of removing assertions from Test Suites

For mutation detection rate, the rate dropped to 0 when all assertions are commented out. The mutants aren't killed since the program detects mutants by comparing the expected result and the actual result. For statement coverage and decision coverage, normative and exception cases are covered even if we delete all the assertions, thus the rates stay the same.

Test Case Redundancy

Test Code Redundancy is the situation where a (set of) test case is present performing the job of coverage as well as mutation detection, but we have created multiple instances of this test case in our test suite. In simple words, there are multiple test cases that follow the same flow path, and detect the same mutations - have same ojective in terms of coverage as well as mutation detection.

Yes there are some test cases in our test suites that are redundant. Just for example, `test_invalid_side_a` in Decision Coverage and `test_incorrect_return_valid_value_a` in Mutation are essentially performing the same task. There are many more such examples that fall in the category of Test Code Redundancy.

We would not remove those test cases, because we are running different files for the coverage and mutation detection. Therefore, removal of one would definitely impact that respective test suite metric. However, if we had a singular file consisting of all test cases, then it is totally fine to remove the redundant test case.

Decision Points in the CFGs

In here we calculated the maximum number of decision points.

1. Scalene Triangle - 6
2. Isosceles Triangle - 9
3. Equilateral Triangle - 6
4. Invalid Sides - 1
5. Triangle Inequality - 9

Information about decision points definitely helps in creating a more effective test suite. It enables a better understanding of the program's control flow, which can significantly enhance test case design. By considering decision points, you can:

- Improve Statement and Decision Coverage: You can identify all possible control flow paths and the corresponding executed code statements. This knowledge aids in designing test cases that cover a broader range of scenarios, improving both statement and decision coverage.
- Enhance Mutation Testing: Decision points are critical for mutation testing, where even slight changes can yield diverse outcomes. Understanding decision points allows you to create an adequate number of test cases to detect mutations effectively.
- Address Edge Cases: Decision boundaries often define edge cases. Creating test cases that focus on these boundaries helps ensure the program functions correctly in critical, real-world scenarios.

In summary, information about decision points guides test case design, leading to better coverage and more robust testing, especially in terms of handling mutations and edge cases.