Assignment 3: Mutation Testing

Ziyan Gong / 94478161, Zhen Wang /98552169

3.1. Assess each of these two mutation testing tools based on their (1) ease of use, (2) set of mutation operators supported, and (3) mutation testing strategy and effectiveness in mutation testing. If you were to select one, which tool would you choose? Motivate your selection. Clearly describe how you conducted the assessment and argue for the pros and cons of each tool.

Answer:

The process of Major has several steps. To specify the mutation process, you need to know about Major's domain specific language (Mml). And MML scripts would be compiled by its compiler mmlc. Then generate mutants based on the compiled Mml script. Mutation reports are stored in log files or csv files. Major divides the mutation analysis process into two steps: generate and embed mutants during compilation, run the actual mutation analysis. For the first step, Major provides a lightweight mutator, which is integrated in the openjdk Java compiler. There are 9 kinds of mutation operators (Arithmetic/ Logical/ Conditional/ Relational/ Shift Operator Replacement, Operator Replacement Unary, Expression Value Replacement, Literal Value Replacement, Statement Deletion) in compiler. For the second step, Major provides a default analysis back-end that extends Apache Ant's JUnit task. It allows programmers to have a systematic way provided by the JUnit framework to efficiently develop a software test suite. Moreover, many existing test cases based on JUnit could be directly re-used for software debugging without any modification. Major also has a DSL support for specifying and adapting mutation operators. This makes Major extensible for mutation operators.

PIT just requires the Maven environment, and we can modify the plugins in pom.xml to limit which code is mutated and which tests are run using. The mutation operators support activated by default (conditionals boundary, increments, invert negatives, math, negate conditionals, return values and void method calls) and deactivated by default (constructor calls, inline constant, non void method calls, remove conditionals, experimental member variable and experimental switch). PIT supports command line execution, Ant and Maven build integration, as well as IDE and reporting integration by third-party offerings. The results are stored in HTML files. For mutation strategy, PIT performs a traditional line coverage analysis for the tests before running the tests, then it uses this data along with the timings of the tests to pick a set of test cases that are targeted at the mutated code. This approach makes PIT much faster than previous mutation testing systems, and enables PIT to test entire code bases rather than single classes at a time.

For me, I would choose PIT to use. Firstly, PIT supports IDE. That would save your life. Secondly, it's faster to perform mutation testing in PIT. Thirdly, by experiments, it shows that PIT is better than Major in detecting errors.

3.2. How many mutations were generated by PIT in total? How many were killed by the test suite? Include the mutation coverage scores, per class, in your document.

Answer:

There are 33 mutations in total. And 30 mutations are killed.

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
5	92%	175/190	91%	30/33

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.jpacman.framework.factory	1	100%	22/22	100%	7/7
org.jpacman.framework.model	4	91%	153/168	88%	23/26

Report generated by PIT 0.28

Pit Test Coverage Report

Package Summary

org.jpacman.framework.factory

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	22/22	100%	7/7

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
DefaultGameFactory.java	100%	22/22	100%	7/7

Report generated by PIT 0.28

Package Summary

org.jpacman.framework.model

Number of Classes	Line Coverage		Mutation Coverage	
4	91%	153/168	88%	23/26

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Board.java	92%	55/60	100%	7/7
Game.java	90%	54/60	86%	6/7
PointManager.java	100%	20/20	100%	7/7
Sprite.java	86%	24/28	60%	3/5

Report generated by PIT 0.28

3.3. List and describe the mutations not killed by the test suite. Are they equivalent mutants or are they a weakness of the test suite? Elaborate on the findings. What is the mutation score of the test suite?

Answer:

In Game.java, there was one mutation that is not killed.

```
replaced return of integer sized value with (x == 0 ? 1 : 0) : NO_COVERAGE
```

In Sprite.java, there were two mutations that are not killed.

```
mutated return of Object value for org/jpacman/framework/model/Sprite::getSpriteType to ( if (x != null) null else throw new RuntimeException ) : NO_COVERAGE
mutated return of Object value for org/jpacman/framework/model/Sprite::toString to ( if (x != null) null else throw new RuntimeException ) : NO_COVERAGE
```

NO COVERAGE means there were no tests that exercised the line of code where the mutation was created. Actually, even without mutation, these lines in the original program code are not covered by executing test suites. So this is a weakness of test suite.
The mutation score of this test suite is 30/33.

3.4 Extend the test suite in such a way that it kills the non-equivalent live mutants PIT generates. What is your new mutation score?

Answer:

For all three mutations that are not covered, they are never covered by any test cases for both program code and mutation code, so mutants cannot be found and killed. So extra test cases are needed for them.
For the game.java, we added one test case for won method.

```java
/**
 * Test when PointsOnBoard changes,
 * the player wining condition should still meet.
 *
 * @throws FactoryException Never.
 */
@Test
public void testPointsOnBoardChange() throws FactoryException {
    Game g = makePlay("P# ");
    Player p = g.getPlayer();
    PointManager pm = g.getPointManager();
    pm.addPointsToBoard(1);
    pm.consumePointsOnBoard(p, 1);
    assertTrue(g.won());
}
```

For the Sprite.java, we added two test cases for getSpriteType method and toString method separately.

```java
/**
 * Test that a sprite get the sprite type
 */
@Test
public void testGetSpriteType() {
    Sprite s = new Sprite() {
    };
    assertEquals(SpriteType.OTHER, s.getSpriteType());

}

/**
 * Test that a sprite get the sprite type to string
 */
@Test
public void testGetSpriteTypeToString() {
    Sprite s = new Sprite() {
    };
    assertTrue(s.toString() instanceof String);
    assertTrue(s.toString().toLowerCase().contains(s.getSpriteType().toString().toLowerCase())

}
```

By adding the above test cases, we get the score of 33/33 and 100% coverage.

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
5	94%	178/190	100%	33/33

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.jpacman.framework.factory	1	100%	22/22	100%	7/7
org.jpacman.framework.model	4	93%	156/168	100%	26/26

4.5 Now expand PIT's mutation operators (called mutators in PIT) beyond what is enabled by default. Enable at least three (3) extra mutators and run PIT on the test suite. Extend the

test suite in such as a way that it kills any non-equivalent live mutants PIT generates now. Explain all of this in your report.

Answer:

After adding three more mutators, there are 10 mutators in total which are shown in the figure below:

```
<mutators>
    <mutator>CONDITIONALS_BOUNDARY</mutator>
    <mutator>INCREMENTS</mutator>
    <mutator>INVERT_NEGS</mutator>
    <mutator>MATH</mutator>
    <mutator>NEGATE_CONDITIONALS</mutator>
    <mutator>RETURN_VALS</mutator>
    <mutator>VOID_METHOD_CALLS</mutator>
    <mutator>CONSTRUCTOR_CALLS</mutator>
    <mutator>INLINE_CONSTS</mutator>
    <mutator>NON_VOID_METHOD_CALLS</mutator>
</mutators>
```

There are 2 more mutation not covered in model module.

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
5	96%	182/190	94%	33/35

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.jpacman.framework.factory	1	100%	22/22	100%	7/7
org.jpacman.framework.model	4	95%	160/168	93%	26/28

Report generated by PIT 0.28

The first one is in Game.java and about the method died. It is no_coverage mutant which means the related test case is needed for it. The detail of it is as follows:

199 negated conditional : NO_COVERAGE

By adding related test case, it is covered and there is no survived mutant.

```
/**
 * Test when player died
 *
 * @throws FactoryException Never.
 */
@Test
public void testPlayerDied() throws FactoryException {
    Game g = makePlay("P# ");
    Player p = g.getPlayer();
    p.die();
    assertTrue(g.died());
}
```

The second one is in the board.java and for the method onBoardMessage. The mutant survived thus we have to create more effective test case to kill it. The detail of the surviving mutant is as follows.

mutated return of Object value for org/jpacman/framework/model/Board::onBoardMessage to (if (x != null) null else throw new RuntimeException) : SURVIVED

We catched the AssertionError and checked the message when a tile is not on board. It successfully killed the mutant.

```
@Test
public void testOnBoardMessage() {
    int x = -1;
    int y = -1;
    try{
        board.tileAt(x, y);
    }catch(AssertionError e){
        assertEquals(e.getMessage(),"PRE: "+
                "(" + x + ", " + y + ")"
                + "not on board of size "
                + board.getWidth() + " * " + board.getHeight()
                );
    }
}
```

Finally, we achieved the test score of 35/35, 100% coverage.

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
5	96%	183/190	100%	35/35

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.jpacman.framework.factory	1	100%	22/22	100%	7/7
org.jpacman.framework.model	4	96%	161/168	100%	28/28

Report generated by PIT 0.28