CPEN522: Assignment3

Jiahui Tang 77383255
Srikanthan Thivaharan 41497900

2019/03/12

1 Exercise 3.1

For this question, we assessed PIT and Major based on their (1) ease of use, (2) set of mutation operators supported, and (3) mutation testing strategy and effectiveness in mutation testing. Here are the investigation results:

PIT mutation analysis tool:

(1) Ease of use:

- PIT is easy to run, the only knowledge the user needs to know is how to add Maven Plugin and Configure Maven build. In addition, PIT also support command lines and it is well supported by Eclipse;
- PIT can analyze a project in minutes while other mutation tools would take days to get the results;
- PIT only supports Junit4 and above for version 0.28;
- Additional plugins are needed to get the specific mutations for version below 1.3.0.

(2) Set of mutation operators supported:

- By default, PIT supports 7 mutation operators: Conditionals Boundary, Increments, Invert Negatives, Math, Negate Conditions, Return Values, and Void Method Calls. Moreover, it also provides other 22 operators such as Arithmetic Operator Replacement (AOR) in the "ALL" mode.

(3) Mutation testing strategy and effectiveness:

- PIT performs a **bytecode level manipulation**: PIT works by manipulating the resulting bytecode of the program under test and employs mutation operators that affect primitive programming language features. Additionally, PIT uses a traditional line coverage analysis for the tests before running the mutation test, then it takes advantage of the data to choose the corresponding test cases that are targeted at the mutated code. This way makes PIT have better performance than previous tools;
- Since PIT's changes are performed at the bytecode level, they cannot always be mapped onto source code ones.

Major mutation analysis tool:

(1) Ease of use:

- Major is easy to use for a standalone JAVA class or any project using Apache Ant. The analysis is fast. However, it seems that other build tools such as Maven and gradle are not supported yet;

- Major furnishes its own domain specific language (MML) to let users add new mutation operators and configure useful plugins;
- The website of Major is too simple and is lack of maintenance. There are no issue-tracking sections;
- Major requires JAVA 1.7.0 otherwise there are some warnings and we are not sure whether the warnings will affect the mutation analysis results;
- The analysis results are shown in separate files (csv or map). It is better to generate a report for all the interesting information;
- The specific mutations can be easily found in mutant.log.

(2) Set of mutation operators supported:

- From the documents, we can find that Major supports 9 mutation operators: Arithmetic Operator Replacement (AOR), Logical Operator Replacement (LOR), conditional Operator Replacement (COR), Rational Operator Replacement (ROR), Shift Operator Replacement (SOR), Operator Replacement Unary (ORU), Expression Value Replacement (EVR), Literal Value Replacement (LVR), and Statement Deletion (STD);
- Major employs specialized versions of specific operators that are supposed to be as effective as PIT. The implemented mutation operators are based on selective mutation, i.e. it implements a set of operators whose mutants subsume the mutants generated by other mutation operators not included in the set;
- Compare to PIT, most operators of Major impose a superset of changes with respect to the corresponding ones of PIT and there are operators of PIT that are completely absent from Major.

(3) Mutation testing strategy and effectiveness:

- Major extends the OpenJDK Java compiler and implements conditional mutation as an optional transformation of the abstract syntax tree (AST). It provides **sourcecode-based mutants**.;
- For the first step, Major provides a lightweight mutator, which is integrated in the OpenJDK Java compiler. For the second step, Major provides a default analysis back-end that extends Apache Ant's JUnit task. By leveraging back-end, major is able to get better analysis result and have preformation optimization.

Based on the investigation, I would like to choose PIT. First of all, most of the project now are using maven or gradle and PIT supports both of them. Second, PIT has well-developed website and User forum. If I come across some questions, I could get answered quickly by searching on those websites or using the mailing list. Last but not list, PIT's analysis result is in detail and easy to understand. It provides both the line coverage and mutation analysis result, which is really helpful for the users.

Here we also want to introduce how we conducted the assessment. We first read all the documents on both tool's website, then we found and read some papers[1][2] that were trying to compare different mutation tools, so we can get the basic ideas of those two tools. Later, we tried to run Major on its own examples by executing *runAll.sh* in the example folder, and we are able to get the analysis results. In addition, we also created a single JAVA class called *Factorial.java* as shown inFigure 1, and ran it by using *javac -XMutator:ALL Factorial.java*, and we got 15 mutants in total. For PIT, we wanted to run the same JAVA file of the example of Major, but we found that PIT does not support Junit3 while the test part of Major example is written in Junit3, so we just conducted mutation analysis directly on Jpacman and tried to understand the results.

[1]Kintis, Marinos, et al. "Analysing and comparing the effectiveness of mutation testing tools: A manual study." 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, 2016.

[2]Gopinath, Rahul, et al. "Does choice of mutation tool matter?." Software Quality Journal 25.3 (2017): 871-920.

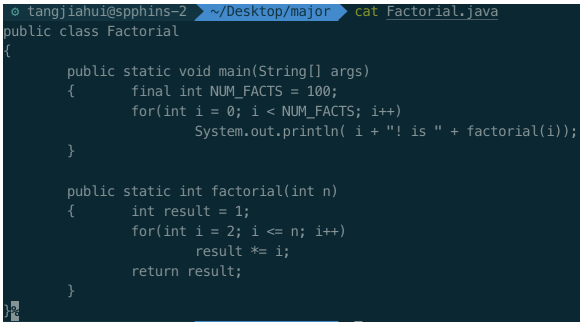


Figure 1: Test Example: Factorial.java

To sum up, for better comparison, as shown in Figure, we listed the pros and cons of each tool.

	PIT	***Major***
Pros	Easy to use. Works well with ant, maven, gradle and others. It also supports command line.	Analysis is fast. Major supports Apache Ant project and standalone JAVA file.
	The mutation analysis process is fast.	Well-defined MML language to add new mutant operators and configure useful plugins.
	The analysis result is presented as HTML files including all the information the user needs.	Compiler-integrated solution and Major uses manipulation of the AST, providing ***source-based mutants*** .
	The documentation of the tool is sufficient, also it provides the source cod and User Group.	The mutation can be easily found in mutants.log
	7 mutation operators in default, and it supports 29 operators for "ALL" group.	Major supports 9 mutation operators.
Cons	Only supports Junit4 or above for version 0.28	Requires JAVA 1.7.0 otherwise there's some warnings
	Bytecode level changes, cannot always be mapped onto source code ones.	Not support Maven and Gradle.
	Needs plugins to get the mutants for verison below 1.3.0	No enough reference documentation; lack of maintenance.
		The results are in separate files (cvs or map).

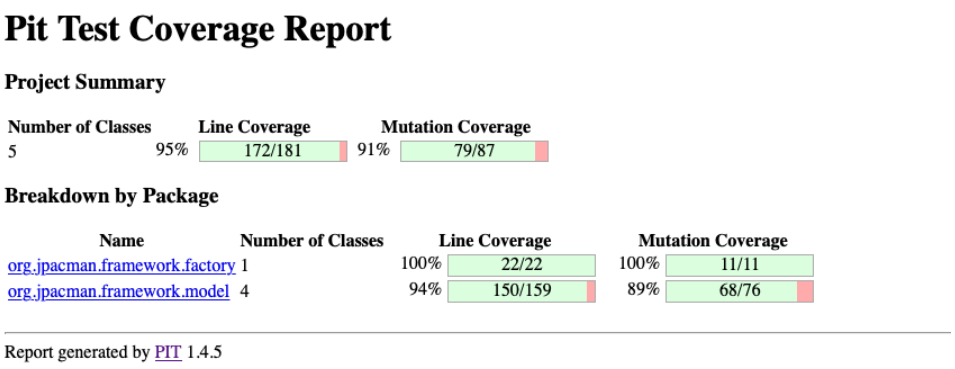
Figure 2: Comparison between PIT and Major

2 Exercise 3.2

In this Exercise, we choose to use PIT mutation analysis tool. The original Jpacman project is configured to use PIT version 0.28. Since we also want to get to know the specific mutations, we update it into the latest version which is 1.4.5. Moreover, in the original *pom.xml*, the target classes are *Factory.DefaultGameFactory, Model.Sprite, Model.Board, Model.PointManager, and Model.Game*. The targetTests are just the previous version of test suites. Here, to get the accurate analysis result of current

test suite, we also added **BoardTest** class that we wrote in Assignment 2.

The mutation analysis result is shown in Figure 3, 87 mutations were generated by PIT in total. 79 (11+68) mutations were killed by the test suite. Overall, the mutation coverage score is 79/87. For package Factory and Model, the value is 11/11 and 68/76 respectively.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
5	95%	172/181	91%	79/87

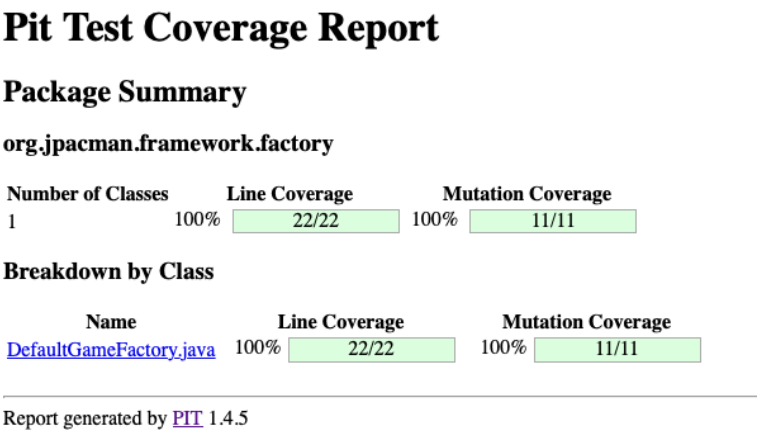
Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.jpacman.framework.factory	1	100%	22/22	100%	11/11
org.jpacman.framework.model	4	94%	150/159	89%	68/76

Report generated by PIT 1.4.5

Figure 3: Overall Mutation Analysis Result

In the package *Factory*, there is only one JAVA class which is *DefaultGameFactory*, so the mutation coverage is 11/11. Figure 4 displays the result.



Pit Test Coverage Report

Package Summary

org.jpacman.framework.factory

Number of Classes	Line Coverage		Mutation Coverage	
1	100%	22/22	100%	11/11

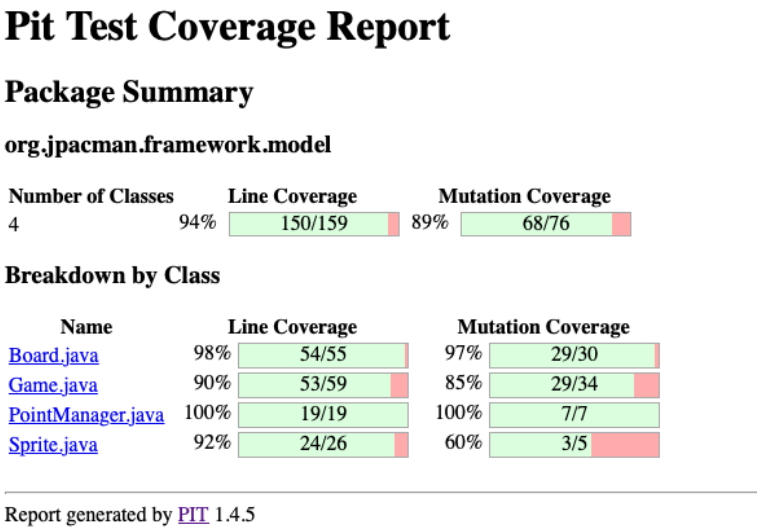
Breakdown by Class

Name	Line Coverage		Mutation Coverage	
DefaultGameFactory.java	100%	22/22	100%	11/11

Report generated by PIT 1.4.5

Figure 4: Mutation Analysis Result of Factory

In the package *Model*, we only considered four classes: *Board, Game, PointManager and Sprite*. The mutation coverage value is 29/30, 29/43, 7/7 and 3/5 respectively. The result is shown in Figure 5.



Pit Test Coverage Report

Package Summary

org.jpacman.framework.model

Number of Classes	Line Coverage		Mutation Coverage	
4	94%	150/159	89%	68/76

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
Board.java	98%	54/55	97%	29/30
Game.java	90%	53/59	85%	29/34
PointManager.java	100%	19/19	100%	7/7
Sprite.java	92%	24/26	60%	3/5

Report generated by PIT 1.4.5

Figure 5: Mutation Analysis Result of Model

3 Exercise 3.3

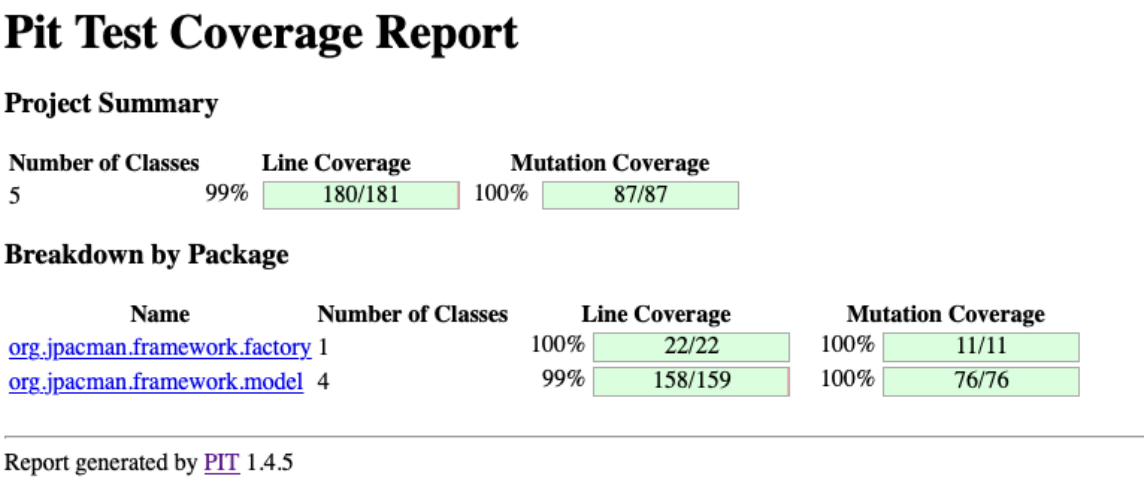
As Figure 6 shows, according to the mutation result, most of mutations (7 cases) that not killed by the test suite we choose are because of no coverage. This indicates that there are some weaknesses in the test suite since those lines were not covered during the mutation analysis. For the one case that survived, it is a mutation generated by RETURN_VALS_MUTATOR. This means that the test suite does not test so well about the return value, which also indicates that there are weaknesses in the test suite. Overall. The mutation score of the test suite we choose is 79/87=90.8%.

Package Name	*Class Name*	*# of unkilled mutations*	*Description of unkilled mutations*
org.jpacman.framework.factory	DefaultGameFactory.java	0	/
org.jpacman.framework.model	Board.java	1 Survived	Line 183: mutated return of Object value for org/jpacman/framework/model/Board::onBoardMessage to (if (x != null) null else throw new RuntimeException) → SURVIVED
	Game.java	5 No coverage	Line 164: removed call to org/jpacman/framework/model/Game::addObserver → NO_COVERAGE
			Line 194: mutated return of Object value for org/jpacman/framework/model/Game::getBoardInspector to (if (x != null) null else throw new RuntimeException) → NO_COVERAGE
			Line 199: negated conditional → NO_COVERAGE
			Line 199: replaced return of integer sized value with (x == 0 ? 1 : 0) → NO_COVERAGE
			Line 204: replaced return of integer sized value with (x == 0 ? 1 : 0) → NO_COVERAGE
	PointManager.java	0	/
	Sprite.java	2 No coverage	Line 82: mutated return of Object value for org/jpacman/framework/model/Sprite::getSpriteType to (if (x != null) null else throw new RuntimeException) → NO_COVERAGE
			Line 87: mutated return of Object value for org/jpacman/framework/model/Sprite::toString to (if (x != null) null else throw new RuntimeException) → NO_COVERAGE

Figure 6: The details of Mutation Result

4 Exercise 3.4

As we can see in Figure 6 of Exercise 3.3, there are 1 survived and 7 no coverage cases. For the one survived case, we added a test case to indirectly test onBoardMessage() method: we tried to produce AssertionError so that we can compare the error message. For the other 7 cases that have no coverage, we added several test cases that call the related methods so that those lines would be covered in the final result. Since there is no equivalent mutant, we are finally able to reach an 87/87=100% mutation score, as shown in Figure 7. All the test cases we added can be found in Appendix A, B and C.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
5	99%	180/181	100%	87/87

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.jpacman.framework.factory	1	100%	22/22	100%	11/11
org.jpacman.framework.model	4	99%	158/159	100%	76/76

Report generated by PIT 1.4.5

Figure 7: Mutation analysis result after extending the test suite.

5 Exercise 3.5

By default, PIT supports 7 mutation operators: Conditionals Boundary, Increments, Invert Negatives, Math, Negate Conditions, Return Values, and Void Method Calls. For this exercise, we also added CONSTRUCTOR_CALLS, EMPTY_RETURNS, FALSE_RETURNS, AOD (2 mutators: AOD_1_MUTATOR, AOD_2_MUTATOR) and AOR (4 mutators: AOR_1_MUTATOR, AOR_2_MUTATOR, AOR_3_MUTATOR, AOR_4_MUTATOR) as shown below in Figure 8 and Figure.

```xml
<plugin>
    <!-- invoke with
         mvn org.pitest:pitest-maven:mutationCoverage
    -->
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.4.5</version>
    <configuration>
        <targetClasses>
            <param>org.jpacman.framework.factory.DefaultGameFactory</param>
            <param>org.jpacman.framework.model.Sprite</param>
            <param>org.jpacman.framework.model.Board</param>
            <param>org.jpacman.framework.model.PointManager</param>
            <param>org.jpacman.framework.model.Game</param>
        </targetClasses>
        <targetTests>
            <param>org.jpacman.test.framework.factory.FactoryIntegrationTest</param>
            <param>org.jpacman.test.framework.model.SpriteTest</param>
            <param>org.jpacman.test.framework.model.PointManagerTest</param>
            <param>org.jpacman.test.framework.model.BoardTileAtTest</param>
            <param>org.jpacman.test.framework.model.BoardTest</param>
            <param>org.jpacman.test.framework.model.GameTest</param>
        </targetTests>
        <excludedMethods>
            <!-- methods called invariant are only used for assertion checking. -->
            <param>*nvariant*</param>
        </excludedMethods>
        <maxMutationsPerClass>
          7
        </maxMutationsPerClass>
        <verbose>
          true
        </verbose>
        <jvmArgs>
            <value>-enableassertions</value>
        </jvmArgs>
        <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>EMPTY_RETURNS</mutator>
           <mutator>FALSE_RETURNS</mutator>
           <mutator>AOD</mutator>
           <mutator>AOR</mutator>
         </mutators>
    </configuration>
</plugin>
```

Figure 8: Pom.xml for Exercise 3.5

Active mutators

- AOD_1_MUTATOR
- AOD_2_MUTATOR
- AOR_1_MUTATOR
- AOR_2_MUTATOR
- AOR_3_MUTATOR
- AOR_4_MUTATOR
- BOOLEAN_FALSE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- CONSTRUCTOR_CALL_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

Figure 9: New Mutator Suite we used for Exercise 3.5

Then we reran mutation analysis, we got a new mutation score: 136/139=97.8% as shown in Figure 10. The details of the unkilled mutation are shown in Figure 11.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
5	99%	180/181	98%	136/139

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.jpacman.framework.factory	1	100%	22/22	100%	17/17
org.jpacman.framework.model	4	99%	158/159	98%	119/122

Report generated by PIT 1.4.5

Figure 10: Mutation Result after we added new mutators.

Package Name	*Class Name*	*# of unkilled mutations*	*Description of unkilled mutations*
org.jpacman.framework.factory	DefaultGameFactory.java	0	/
org.jpacman.framework.model	Board.java	2 Survived	Line 157: 6. Replaced integer operation with first member → SURVIVED
			Line 157: 22. Replaced integer modulus with addition → SURVIVED
	Game.java	0	/
	PointManager.java	1 Survived	Line 37: 3. Replaced integer operation by second member → SURVIVED
	Sprite.java	0	/

Figure 11: Details of unkilled mutations after we added new mutators

As we can see from Figure 11, there are 3 new survived mutations: 2 for Board.java and 1for PointManager.java. Then we looked those mutations and we found that 1 mutation for Board.java is an equivalent live mutant: *((current + delta) % max + max) % max* would be the same as *((current + delta) + max + max) % max* since there is always a modulus operation outside the bracket, so we cannot kill this mutation. For the other two cases, we wrote two test cases for them. The code can be found in Appendix D and E.

Finally, we can reach a mutation score of 138/139=99.2% as shwon in Figure 12. In fact, if we exclude the equivalent case, the mutation score should be 100%.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
5	99%	180/181	99%	138/139

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.jpacman.framework.factory	1	100%	22/22	100%	17/17
org.jpacman.framework.model	4	99%	158/159	99%	121/122

Report generated by PIT 1.4.5

Figure 12: Final Mutation Result after we added new test cases.

Appendix A Exercise 3.4: New testcase for BoardTest.java

```
/**
 * Test method for onBoardMessage() according to
 * mutation analysis result.
 */
@Test
public void testOnBoardMessage() {
        try {
        board.tileAt(-1, -1);
        fail("Test Failed");
        }catch(AssertionError e) {
                String e_message = e.getMessage();
                String message = "PRE: (-1, -1)not on board of
                    size " + WIDTH + " * " + HEIGHT;
                assertThat(message, equalTo(e_message));
        }
}
```

Appendix B Exercise 3.4: New testcases for GameTest.java

```
/**
 * Test that the observers are informed after a ghost move
 * by using attach() method and getBoardInspector() method
 * according to mutation analysis result.
 *
 * addObserver() -> attach()
 * getBoard() -> getBoardInspector()
 *
 * @throws FactoryException Never.
 */
@Test
public void testObserverAttachAndGetBoardInspector() throws
    FactoryException {
        // given
        Observer anObserver = mock(Observer.class);
        Game game2 = makePlay("G #");
        game2.attach(anObserver);
        Ghost ghost = (Ghost)
            game2.getBoardInspector().spriteAt(0, 0);
        // when
        game2.moveGhost(ghost, Direction.RIGHT);
        // then
        verify(anObserver).update(any(Observable.class),
            anyObject());
}

/**
 * Test that player dies if he bumps into a ghost by
 * getting the return value of died() according to
 * mutation analysis result.
 *
 * "Move kills player" -> died()
 *
 * @throws FactoryException Never.
```

```
     */
    @Test
    public void testPlayerMovesToGhostAndDied() throws
        FactoryException {
            Game g = makePlay("PG#");
            Player p = g.getPlayer();

            g.movePlayer(Direction.RIGHT);

            assertThat(g.died(), equalTo(!p.isAlive()));
            assertThat(tileAt(g, 1, 0), equalTo(p.getTile()));
    }

    /**
     * Test that player wins if they ate all the food.
     *
     *  -> won()
     *
     * @throws FactoryException Never.
     */
    @Test
    public void testPlayerMovesToFoodAndWon() throws FactoryException {
            Game g = makePlay("P..#");
            Player p = g.getPlayer();
            PointManager pm = g.getPointManager();

            g.movePlayer(Direction.RIGHT);
            g.movePlayer(Direction.RIGHT);

            assertThat(g.won(), equalTo(pm.allEaten()));
            assertThat(tileAt(g, 2, 0), equalTo(p.getTile()));
    }
```

Appendix C Exercise 3.4: New testcase for SpriteTest.java

```
/**
     * Test toString() and getSpriteType() according
     * to mutation analysis result.
     */
    @Test
    public void testGetSpriteTypeAndToString() {
            assertThat(john.getSpriteType(),
                equalTo(SpriteType.OTHER));
            String str = john.getSpriteType().toString() + " occupying
                " + john.getTile();
            assertThat(john.toString(), equalTo(str));
    }
```

Appendix D Exercise 3.5: New testcase for Board.java

```
    /**
     * Exercise 3.5
     * New Test cases based on the new Mutation analysis
     * Result. Kill the mutation that replaced integer
```

```
     * operation with first member
     */
    @Test
    public void testTileAtOffsetMore() {
            assertThat(board.tileAtOffset(tile_1_1, -(WIDTH * 2), 2),
                equalTo(tile_1_3));
    }
```

Appendix E Exercise 3.5: New testcase for PointManager.java

```
/**
     * Exercise 3.5
     * New test cases based on the result
     * of new Mutation analysis. Kill the
     * mutation that replaced integer
     * operation by second member.
     *
     */
    @Test public void testEatSomeFood_2() {
            final int pointsToEat = totalPoints / 2;
            final int pointsToEat_2 = 1;
            pm.consumePointsOnBoard(player, pointsToEat);
            assertFalse(pm.allEaten());
            assertEquals(pointsToEat, pm.getFoodEaten());
            pm.consumePointsOnBoard(player, pointsToEat_2);
            assertThat(pointsToEat, not(pm.getFoodEaten()));
            assertEquals(pointsToEat + pointsToEat_2,
                pm.getFoodEaten());
    }
```