CPEN522: Assignment2

Jiahui Tang 77383255
Srikanthan Thivaharan 41497900

2019/02/13

1 Exercise 2.1

For this exercise, we decided to choose scenario 1 in our Assignment 1, which is about exploring the basic function of starting, stopping and exiting a JPacman Game. We replayed it by performing the following steps: 1) The user launches JPacman game; 2) The user clicks the start button at the bottom of the window; 3) The user clicks the stop button at the bottom of the window; 4) The user presses the exit button at the bottom of the window. The resulting test coverage in the coverage view is depicted in Figure 1, then we looked deeper into some classes.

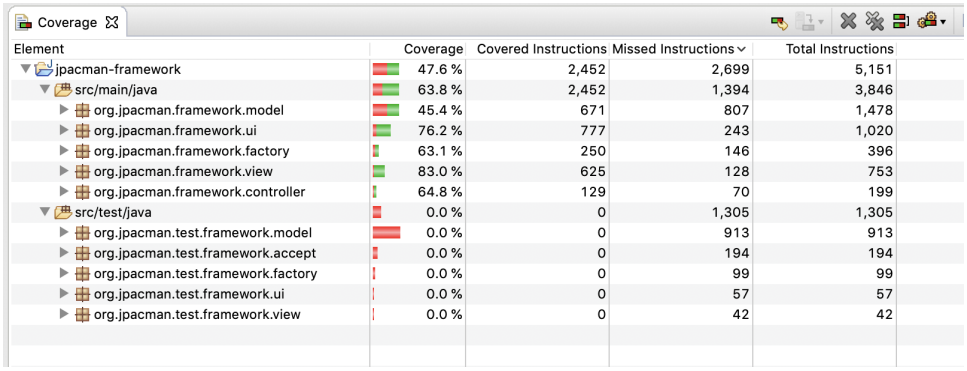


Figure 1: The resulting test coverage for scenario 1.

From the figure, we can find that the overall coverage rate is 47.6%. For the application code, the coverage is 63.8%. Because we did not run any code from Junit, the code coverage for that part is 0.0%. Since our scenario is about UI, so we first looked into *org.jpacman.framework.ui* and *org.jpacman.framework.view*, then looked through other classes. Here are our interesting findings:

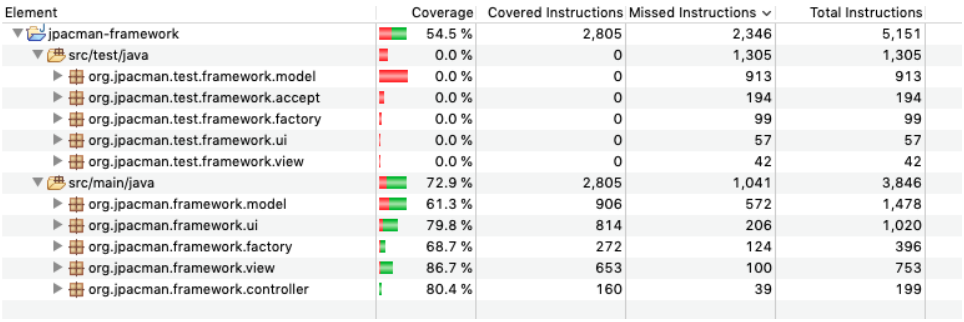
a) For the java classes for UI, almost all of them reach a coverage rate greater than 75%, except *PackmanKeyListener*. Since in our scenario, JPacman game is loaded and the UI is shown to us, so it is reasonable that most of the UI code except for the asserts are covered. Besides, in *Pacman-Interaction*, because we did not move Pacman, the *movePlayer()* and some relevant methods are not covered. More obvious, we also did not press any key, so the coverage for *PacmanKeyListener* is just 14.6%;

b) Similarly, for the java classes for View, our actions covered 83% of the code. The one with the least coverage is *ImageLoader* (70%), and after we doublechecked it, we found the uncovered part is still the exceptions.

c) For the other code, since we did not play the game, eat any foods or meet any ghost, so the coverage rate for *Game*, *Player*, and *PointManager* are all below 60%.

2 Exercise 2.2

After enabling assertion checking, as shown in Figure 2, it is easy to find that the overall code coverage is increased to 54.5%. For the testing code, same to the previous exercise, the coverage rate still stands at 0. However, the overall coverage for main code is enhanced from 63.8% to 72.9%. When we looked deeper, all the class with assertions got a better code coverage (from 3 of 4 branches missed to only 2 of 4 missed). For instance, the coverage for code for controller is increased from 64.8% to 80.4%.

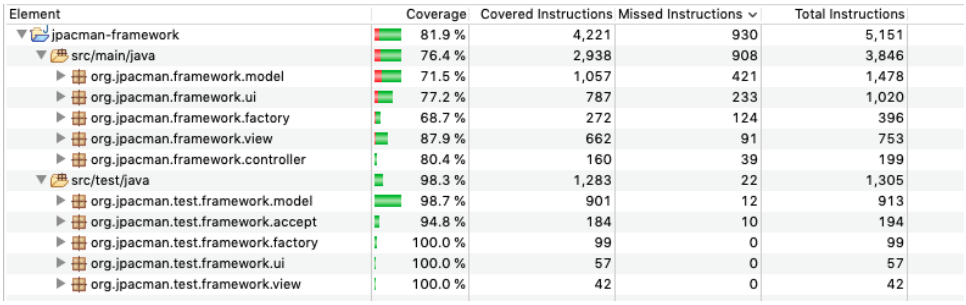


Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
jpacman-framework	54.5 %	2,805	2,346	5,151
src/test/java	0.0 %	0	1,305	1,305
org.jpacman.test.framework.model	0.0 %	0	913	913
org.jpacman.test.framework.accept	0.0 %	0	194	194
org.jpacman.test.framework.factory	0.0 %	0	99	99
org.jpacman.test.framework.ui	0.0 %	0	57	57
org.jpacman.test.framework.view	0.0 %	0	42	42
src/main/java	72.9 %	2,805	1,041	3,846
org.jpacman.framework.model	61.3 %	906	572	1,478
org.jpacman.framework.ui	79.8 %	814	206	1,020
org.jpacman.framework.factory	68.7 %	272	124	396
org.jpacman.framework.view	86.7 %	653	100	753
org.jpacman.framework.controller	80.4 %	160	39	199

Figure 2: The resulting test coverage with assertion checking enabled for scenario 1.

3 Exercise 2.3

The pictures below show the total coverage of the whole test suite in JPacman (Figure 3 has assertion checking enabled, Figure 4 does not). The coverage percentages as a whole, for the application code, and for test cases are 81.9%, 76.4%, and 98.3% respectively in Figure 3. Without assertions checking, those coverage rates are 75.0%, 67.2%, and 98.2% respectively. For both situations, the best representative of the test suite coverage is the one for the application code. Because the test suite is to test the main code, we do not need to care about the coverage percentages for those test cases. Since Junit test cases are running automatically by Eclipse, the coverage rate for test cases is always high, so the coverage as a whole is also affected.



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
jpacman-framework	81.9 %	4,221	930	5,151
src/main/java	76.4 %	2,938	908	3,846
org.jpacman.framework.model	71.5 %	1,057	421	1,478
org.jpacman.framework.ui	77.2 %	787	233	1,020
org.jpacman.framework.factory	68.7 %	272	124	396
org.jpacman.framework.view	87.9 %	662	91	753
org.jpacman.framework.controller	80.4 %	160	39	199
src/test/java	98.3 %	1,283	22	1,305
org.jpacman.test.framework.model	98.7 %	901	12	913
org.jpacman.test.framework.accept	94.8 %	184	10	194
org.jpacman.test.framework.factory	100.0 %	99	0	99
org.jpacman.test.framework.ui	100.0 %	57	0	57
org.jpacman.test.framework.view	100.0 %	42	0	42

Figure 3: The resulting test coverage with assertion checking enabled.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
jpacman-framework	75.0 %	3,864	1,287	5,151
src/main/java	67.2 %	2,583	1,263	3,846
org.jpacman.framework.model	54.1 %	799	679	1,478
org.jpacman.framework.ui	75.6 %	771	249	1,020
org.jpacman.framework.factory	63.1 %	250	146	396
org.jpacman.framework.view	84.2 %	634	119	753
org.jpacman.framework.controller	64.8 %	129	70	199
src/test/java	98.2 %	1,281	24	1,305
org.jpacman.test.framework.accept	93.8 %	182	12	194
org.jpacman.test.framework.model	98.7 %	901	12	913
org.jpacman.test.framework.factory	100.0 %	99	0	99
org.jpacman.test.framework.ui	100.0 %	57	0	57
org.jpacman.test.framework.view	100.0 %	42	0	42

Figure 4: The resulting test coverage without assertion checking enabled.

However, we still felt that the coverage for the application code does not clearly explain the test suite coverage at the fullest. It is just the best between above three coverage rates. Test suite has executed some parts of the application code, which has not executed earlier. Some application code could have executed due to method calls from inside the application code itself.

For example:

1. In *tileInvariant()*, *tileInvariant()* method is not called directly by any test case but the Test Coverage indicates that some part of the method is indicated as covered. This is because, the *BoardTileAtTest* calls the *Board* class which in-turn calls *tileInvariant()*.
2. In *put()*, *put()* method is not directly called by any test case but the Test Coverage indicates that some part of the method is indicated as covered. This is because the *addSprite()* method internally calls it.

```
44    protected final boolean tileInvariant() {
45        boolean result = true;
46        for (int x = 0; x < width; x++) {
47            for (int y = 0; y < height; y++) {
48                result = result
49                && (tileAt(x, y).getX() == x)
50                && (tileAt(x, y).getY() == y);
51            }
52        }
53        return result;
54    }
```

Figure 5: Example - tileInvariant().

```
public void put(Sprite s, int x, int y) {
    assert withinBorders(x, y) : "PRE1: " + onBoardMessage(x, y);
    assert s != null : "PRE2: Sprite not null";
    assert s.getTile() == null : "PRE3: Sprite should not occupy" + s.getTile();
    s.occupy(tileAt(x, y));
}
```

Figure 6: Example - put().

4 Exercise 2.4

As shown in Appendix A, we added 20 test methods in our BoardTest class to test *model.Board*. For some methods like *withBorders()*, *getWidth()*, *getheight()*, and *tileAtDirection()*, we only need to test

the basic functions of those methods. For the others, we also care about the assertion error, so we used **@Test(expected = AssertionError.class)** to distinguish those error. For example, we tested *tileAtOffset()* with the start is null (expected an assertion error), with the start is not on the board (expected an assertion error), and with the normal functions. More details can be found in the comments part of our code.

5 Exercise 2.5

The domain matrix table we used in our unit test class is shown below in Figure 7. Here we set the width of board is 5 and the height of board is also 5, so according to the matrix table, we only need to create 8 test cases to get essentially independent combinations of domains. The code of our unit test class can be found in Appendix B.

Boundary conditions for " x >= 0 && x < width (5) && y >= 0 && y < height (5)"										
Boundary			Test Cases							
Variable	Condition	Type	t1	t2	t3	t4	t5	t6	t7	t8
x	>=0	on	0							
		off		-1						
	< width (5)	on			5					
		off				4				
	typical	in					1	2	3	4
y	>=0	on					0			
		off						-1		
	< height (5)	on							5	
		off								4
	typical	in	1	2	3	4				
Expected Result			TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE

Figure 7: Domain matrix table.

6 Exercise 2.6

For **BoardTest** class: Since we tested most of the assertions, so it is necessary to run the code coverage with assertion checking enabled (-ea). As shown in Figure 8, without our test class, the code coverage for Board class is only 57.6%. After we added our test class, we get our single test class coverage as shown in Figure 9. The code coverage rate is increased into 83.9%. The resulting coverage is around the acceptable value (85%). We then checked the missed instructions, and found that most of them are because of the assert statements as shown in Figure 10 (We only covered 3 of 4 branches of those assert). Also, as presented in Figure 11, in *tielAtOffset()*, since we cannot create a null result, so we only covered 2 of 4 branches in this case. All in all, the resulting coverage percentage is reasonable and acceptable.

jpacman-framework (1) (12-Feb-2019 3:00:52 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
jpacman-framework	81.9 %	4,221	930	5,151
src/main/java	76.4 %	2,938	908	3,846
org.jpacman.framework.model	71.5 %	1,057	421	1,478
Board.java	57.6 %	276	203	479
Sprite.java	59.8 %	98	66	164
Tile.java	68.1 %	113	53	166
Level.java	65.9 %	56	29	85

Figure 8: Board class coverage with -ea option using existing test cases.

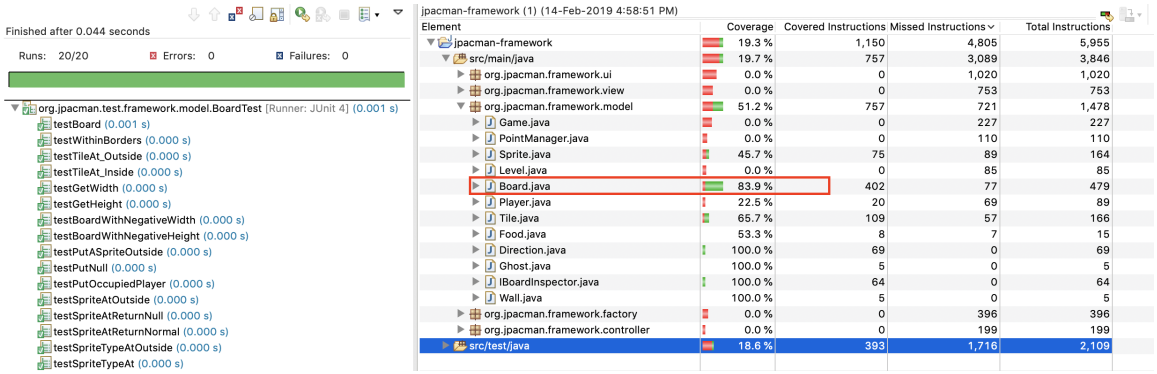


Figure 9: Board.java single class coverage using our test class (BoardTest).

```
93      public Sprite spriteAt(int x, int y) {
94          assert withinBorders(x, y) : "PRE: " + onBoardMessage(x, y);
95          return tileAt(x, y).topSprite();
96      }
97
98      @Override
99      public SpriteType spriteTypeAt(int x, int y) {
100 1 of 4 branches missed.nBorders(x, y) : "PRE: " + onBoardMessage(x, y);
101         Sprite s = spriteAt(x, y);
102         SpriteType result;
103         if (s == null) {
104             result = SpriteType.EMPTY;
105         } else {
106             result = s.getSpriteType();
107         }
108         return result;
```

Figure 10: Covered 3 of 4 branches for most assert statements.

```
129     public Tile tileAtOffset(Tile start, int dx, int dy) {
130         assert start != null : "PRE1: start tile should not be null.";
131 2 of 4 branches missed.nBorders(start.getX(), start.getY())
132                 : "PRE2" +  onBoardMessage(start.getX(), start.getY());
133
134         int newx = tunnelledCoordinate(start.getX(), getWidth(), dx);
135         int newy = tunnelledCoordinate(start.getY(), getHeight(), dy);
136         assert withinBorders(newx, newy);
137
138         Tile result = tileAt(newx, newy);
139
140         assert result != null : "POST: result should not be null.";
141         return result;
142     }
```

Figure 11: Only 2 of 4 branches covered in the assert statements of tileAtOffset().

For **withinBordersTest** class: Since this class only tests one method, so the code coverage for Borad class is only 26.9%. However, if we dive into the test method: withinBorders(int x, int y) as shown in Figure, actually, we covered all the part of this method (all green means that there is no any branch missed), so for this single method, we reached a 100% coverage.

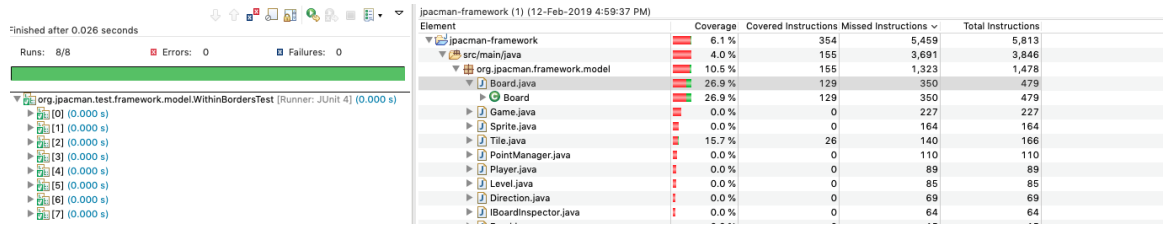


Figure 12: Board class coverage with -ea option using our test class (withinBordersTest).

```
85      */
86     public boolean withinBorders(int x, int y) {
87         return
88             x >= 0 && x < width
89             && y >= 0 && y < height;
90     }
```

Figure 13: The statement in withinBorders() is all green (all covered).

7 Exercise 2.7

Figure 14 shows the new coverage of the whole test suite, including our two test classes. We can easily find that the coverage percentage increases from 81.9% to 85.1%. The difference is 3.2%. Since we enhanced the code coverage for Board class from only 57.6% to 83.9%, as a result, there is no wonder that the overall coverage would go up.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
jpacman-framework	85.1 %	4,945	868	5,813
src/main/java	80.1 %	3,080	766	3,846
org.jpacman.framework.model	81.1 %	1,199	279	1,478
Board.java	83.9 %	402	77	479
Sprite.java	59.8 %	98	66	164
Tile.java	77.7 %	129	37	166
Level.java	65.9 %	56	29	85
PointManager.java	78.2 %	86	24	110
Game.java	90.3 %	205	22	227
Player.java	77.5 %	69	20	89
Food.java	73.3 %	11	4	15
Direction.java	100.0 %	69	0	69
Ghost.java	100.0 %	5	0	5
IBoardInspector.java	100.0 %	64	0	64
Wall.java	100.0 %	5	0	5
org.jpacman.framework.ui	77.2 %	787	233	1,020
org.jpacman.framework.factory	68.7 %	272	124	396
org.jpacman.framework.view	87.9 %	662	91	753
org.jpacman.framework.controller	80.4 %	160	39	199

Figure 14: New code coverage view after including BoardTest and WithinBordersTest class.

8 Exercise 2.8

The two least covered classes in JPacman is *FactoryException* (0.0%) and *PacmanKeyListener* (14.6%). Since during the Junit test, there is no factory exception, also no user interactions, so the code coverage for those two classes is minimal. We created two test classes for the two cases: **FactoryExceptionTest** and **PacmanKeyListenerTest**.

In **FactoryExceptionTest**, we created two factory exceptions and tried to catch them[1], after we got the exceptions, we tried to check the content of messages and causes. As we can see from Figure 15, the code coverage for FactoryException class changes into 100%. The code of our test class can be found in Appendix C.

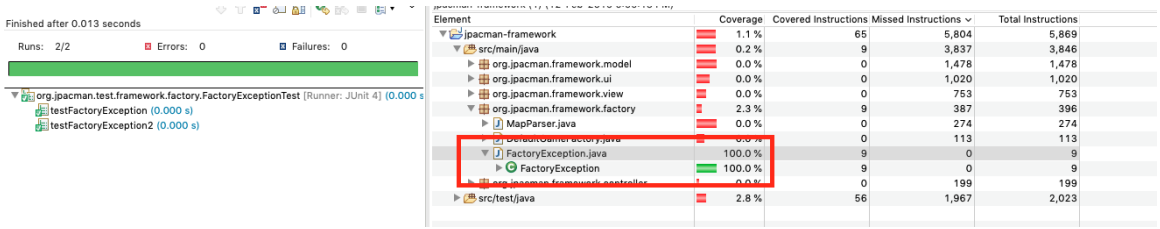


Figure 15: FactoryException.java single class coverage using our test class (FactoryExceptionTest).

In **PacmanKeyListenerTest**, we basically did the same stuff (smoke testing) as *MainUISmokeTesting*. Since *PacmanKeyListener* class constructor is not visible, we cannot initialize it directly, so we used **Robot** package to help us to call it indirectly and to mimic player's actions. After we created a several of player actions by stimulating *KeyEvents*[2][3][4], we can check manually to see if the expected operations are performed or not. For example, in our test case, the user is supposed to go left and then back to the right, so the UI shall show that the food is ate, as displayed in Figure 16. Beside, it worthy mentioning that *Robot* does not work well on MacBook Pro, so we finally ran our code in Windows system. From Figure 17, we can see that the code coverage for *PacmanKeyListener* is improved into 100%. The code of our test class can be found in Appendix D.

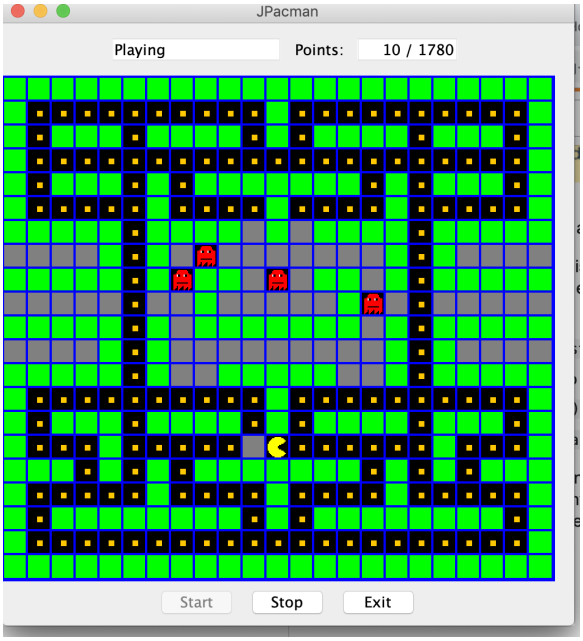


Figure 16: Simulating User's actions to eat a food during our smoke test.

[1]https://stackoverflow.com/questions/2469911/how-do-i-assert-my-exception-message-with-junit-test-annotation
[2]https://stackoverflow.com/questions/14867040/how-to-create-a-keyevent
[3]https://stackoverflow.com/questions/21355201/java-robot-class-keyevent
[4]https://stackoverflow.com/questions/54650020/cant-simulate-enter-key-press-on-macbook

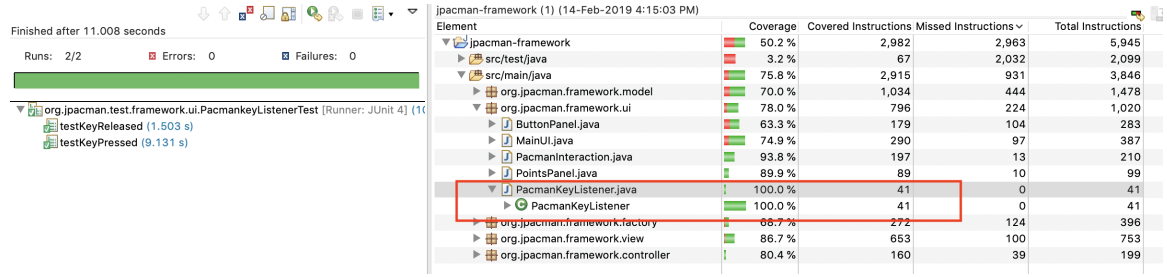


Figure 17: PacmanKeyListener.java single class coverage using our test class (PacmanKeyListenerTest).

In the end, as presented in Figure 18, after we added the two test classes, the final code coverage as a whole is able to reach 86.1%.

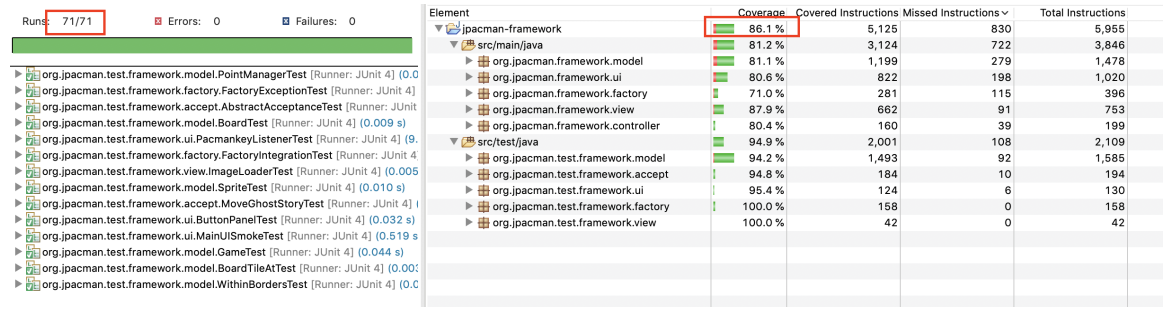


Figure 18: The final code coverage view with 71 test runs in total.

Appendix A BoardTest.java

```
/**
 * Exercise 2.4 for CPEN522
 * Create a new test class for testing model.Board.
 */
package org.jpacman.test.framework.model;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;
import static org.hamcrest.CoreMatchers.equalTo;

import org.jpacman.framework.model.Board;
import org.jpacman.framework.model.Direction;
import org.jpacman.framework.model.Tile;
import org.jpacman.framework.model.IBoardInspector.SpriteType;
import org.jpacman.framework.model.Player;
import org.jpacman.framework.model.Sprite;
import org.jpacman.framework.model.Ghost;
import org.jpacman.framework.model.Food;
import org.jpacman.framework.model.Wall;

/**
 * @author Group8
 *
```

```
 */
public class BoardTest {
        private Board board, board_test1, board_test2;
        private final Sprite sprite = new Sprite() { };
        private final Player player = new Player() { };
        private final Ghost ghost = new Ghost() { };
        private final Food food = new Food() { };
        private final Wall wall = new Wall() { };
        private Tile tile_1_1, tile_1_2, tile_2_1, tile_2_2;
        private static final int WIDTH = 5, HEIGHT = 5;

        /**
         * Set up a board with specified width and height.
         * @throws java.lang.Exception
         */
        @Before
        public void setUp() throws Exception {
                board = new Board(WIDTH, HEIGHT);
                tile_1_1 = board.tileAt(1, 1);
                tile_1_2 = board.tileAt(1, 2);
                tile_2_1 = board.tileAt(2, 1);
                tile_2_2 = board.tileAt(2, 2);
        }

        /**
         * Test method for withinBorders()
         */
        @Test
        public void testWithinBorders() {
                for (int x = 0; x < WIDTH; x++) {
                        for (int y = 0; y < HEIGHT; y++) {
                                assertTrue(board.withinBorders(x, y));
                                }
                        }
                assertFalse(board.withinBorders(-1, HEIGHT - 1));
                assertFalse(board.withinBorders(WIDTH, HEIGHT - 1));
                assertFalse(board.withinBorders(WIDTH - 1, -1));
                assertFalse(board.withinBorders(WIDTH - 1, HEIGHT));
                assertFalse(board.withinBorders(WIDTH, HEIGHT));
        }

        /**
         * Test method for tileAt(). It is suppose to throw
         * an AssertionError if we try to get a tile out of
         * the board.
         */
        @Test(expected = AssertionError.class)
        public void testTileAt_Outside() {
                board.tileAt(WIDTH, HEIGHT);
        }

        /**
         * Test method for tileAt(). tileAt() shall return
         * A tile with correct x, y positions.
         */
        @Test
```

```
public void testTileAt_Inside() {
        for (int x = 0; x < WIDTH; x++) {
                for (int y = 0; y < HEIGHT; y++) {
                        assertThat(board.tileAt(x, y).getX(),
                           equalTo(x));
                        assertThat(board.tileAt(x, y).getY(),
                           equalTo(y));
                }
        }
}

/**
 * Test method for getWidth(). It shall return
 * the correct width as assigned.
 */
@Test
public void testGetWidth() {
        assertThat(board.getWidth(), equalTo(WIDTH));
        }

/**
 * Test method for getHeight(). It shall return
 * the correct height as assigned.
 */
@Test
public void testGetHeight() {
        assertThat(board.getHeight(), equalTo(HEIGHT));
}

/**
 * Test method for Board(). Test if an assertion
 * error is thrown when a negative width is
 * assigned.
 */
@Test(expected = AssertionError.class)
public void testBoardWithNegativeWidth() {
        board_test1 = new Board(-1, HEIGHT);
}

/**
 * Test method for Board(). Test if an assertion
 * error is thrown when a negative height is
 * assigned.
 */
@Test(expected = AssertionError.class)
public void testBoardWithNegativeHeight() {
        board_test2 = new Board(WIDTH, -1);
}

/**
 * Test method for put(). Test if an assertion
 * error is thrown when putting a sprite at
 * the outside of the board.
 */
@Test(expected = AssertionError.class)
public void testPutASpriteOutside() {
```

```
        board.put(sprite, -1, -1);
}

/**
 * Test method for put(). Test if an assertion
 * error is thrown when putting a null.
 */
@Test(expected = AssertionError.class)
public void testPutNull() {
        board.put(null, 0, 0);
}

/**
 * Test method for put(). Test if an assertion
 * error is thrown when putting an occupied
 * sprite.
 */
@Test(expected = AssertionError.class)
public void testPutOccupiedPlayer() {
        player.occupy(tile_1_1);
        board.put(player, 1, 1);
}

/**
 * Test method for spriteAt(). Test if
 * an assertion error is thrown when giving
 * a sprite that is out the board.
 */
@Test(expected = AssertionError.class)
public void testSpriteAtOutside() {
        board.spriteAt(-1, -1);
}

/**
 * Test method for spriteAt(). Test if the method
 * returns null when it should be.
 */
@Test
public void testSpriteAtReturnNull() {
        assertNull(board.spriteAt(2, 2));
}

/**
 * Test method for spriteAt(). Test the generic
 * function of the method.
 */
@Test
public void testSpriteAtReturnNormal() {
        board.put(sprite, 2, 2);
        assertThat(board.spriteAt(2, 2), equalTo(sprite));
}

/**
 * Test method for spriteTypeAt(). Test if
 * an assertion error is thrown when trying
 * to find a sprite type which is not on the
```

```
 * board.
 */
@Test(expected = AssertionError.class)
public void testSpriteTypeAtOutside() {
        board.spriteTypeAt(-1, -1);
}

/**
 * Test method for spriteTypeAt(). Test the
 * generic functions.
 */
@Test
public void testSpriteTypeAt() {
        assertThat(board.spriteTypeAt(2, 2),
            equalTo(SpriteType.EMPTY));
        board.put(player, 3, 1);
        assertThat(board.spriteTypeAt(3, 1),
            equalTo(SpriteType.PLAYER));
        board.put(ghost, 3, 2);
        assertThat(board.spriteTypeAt(3, 2),
            equalTo(SpriteType.GHOST));
        board.put(food, 3, 3);
        assertThat(board.spriteTypeAt(3, 3),
            equalTo(SpriteType.FOOD));
        board.put(wall, 3, 4);
        assertThat(board.spriteTypeAt(3, 4),
            equalTo(SpriteType.WALL));
}

/**
 * Test method for tileAtOffset(). Test if it
 * throws an assertion error if the start is
 * null.
 */
@Test(expected = AssertionError.class)
public void testTileAtOffsetNull() {
        board.tileAtOffset(null, 1, 1);
}

/**
 * Test method for tileAtOffset(). Test if it
 * throws an assertion error if the start is not
 * on the board.
 */
@Test(expected = AssertionError.class)
public void testTileAtOffsetStartFromOutside() {
        board.tileAtOffset(board.tileAt(-1, -1), 1, 1);
}

/**
 * Test method for tileAtDirection(). Test the
 * basic functions.
 */
@Test
public void testTileAtDirection() {
        assertThat(board.tileAtDirection(tile_1_1,
```

```
                Direction.DOWN), equalTo(tile_1_2));
        assertThat(board.tileAtDirection(tile_1_1,
                Direction.RIGHT), equalTo(tile_2_1));
        assertThat(board.tileAtDirection(tile_2_2, Direction.UP),
                equalTo(tile_2_1));
        assertThat(board.tileAtDirection(tile_2_2,
                Direction.LEFT), equalTo(tile_1_2));
    }

    /**
     * Test method for tileAtOffset(). Test the
     * basic functions.
     */
    @Test
    public void testTileAtOffset() {
        assertThat(board.tileAtOffset(tile_1_1, 1, 1),
                equalTo(tile_2_2));
        assertThat(board.tileAtOffset(tile_1_1, WIDTH, 1),
                equalTo(tile_1_2));
        assertThat(board.tileAtOffset(tile_1_1, 1, HEIGHT),
                equalTo(tile_2_1));
        assertThat(board.tileAtOffset(tile_1_1, WIDTH, HEIGHT),
                equalTo(tile_1_1));
    }

}
```

Appendix B WithinBordersTest.java

```
/**
 * Exercise 2.5 for CPEN522
 * Add a unit test class for the model.Board.withinBorders method.
 */
package org.jpacman.test.framework.model;

import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.equalTo;

import java.util.Arrays;
import java.util.Collection;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

import org.jpacman.framework.model.Board;

/**
 * @author Group8
 *
 */
@RunWith(Parameterized.class)
public class WithinBordersTest {
    private Board board;
```

```
        private int x, y;
        private boolean inside;

        // we use a 5x5 board here
        private static final int WIDTH = 5;
        private static final int HEIGHT = 5;

        /**
         * Create a new test case to test whether
         * if the given position is within the
         * board.
         * @param x
         * @param y
         * @param inside
         */
        public WithinBordersTest(int x, int y, boolean inside) {
                this.x = x;
                this.y = y;
                this.inside = inside;
                board = new Board(WIDTH, HEIGHT);
        }

        /**
         * The actual test case.
         */
        @Test
        public void testWithinBorders() {
                assertThat(board.withinBorders(x, y), equalTo(inside));
        }

        /**
         * Use the 1x1 domain testing strategy to
         * create a serials of test cases with
         * parameterized data.
         */
        @Parameters
        public static Collection<Object[]> data() {
                Object[][] values = new Object[][] {
                        {0, 1, true},
                        {-1, 2, false},
                        {5, 3, false},
                        {4, 4, true},
                        {1, 0, true},
                        {2, -1, false},
                        {3, 5, false},
                        {4, 4, true}
                };
                return Arrays.asList(values);
        }

}
```

Appendix C FactoryExceptionTest.java

```
/**
```

```
 * Test class for FactoryException
 */
package org.jpacman.test.framework.factory;

import static org.junit.Assert.*;

import org.jpacman.framework.factory.FactoryException;
import org.junit.Test;

/**
 * @author group8
 *
 */
public class FactoryExceptionTest {
        private FactoryException factory_test1;
        private FactoryException factory_test2;
        String message1 = "Testing Factory Exception1";
        String message2 = "Testing Factory Exception2";
        Throwable error = new Throwable("Error");

        @Test
        public void testFactoryException() {
                factory_test1 = new FactoryException(message1);
                try {
                        throw factory_test1;
                } catch (FactoryException e) {
                        assertEquals(message1, factory_test1.getMessage());
                }
        }

        @Test
        public void testFactoryException2() {
                factory_test2 = new FactoryException(message2, error);
                try {
                        throw factory_test2;
                } catch (FactoryException e) {
                        assertEquals(message2, factory_test2.getMessage());
                        assertEquals(error, factory_test2.getCause());
                }
        }

}
```

Appendix D PacmankeyListenerTest.java

```
/**
 * Test class for PacmanKeyListener
 */
package org.jpacman.test.framework.ui;

import static org.junit.Assert.*;

import org.junit.Test;

import java.awt.AWTException;
```

```java
import java.awt.Robot;
import java.awt.event.KeyEvent;


import org.jpacman.framework.factory.FactoryException;
import org.jpacman.framework.ui.MainUI;

/**
 * @author group8
 *
 */
public class PacmankeyListenerTest {
        MainUI ui = new MainUI();

        /**
         * Test method for KeyReleased. Since there is
         * no any context in this method, so no any
         * exception is expected.
         * @throws FactoryException
         */
        @Test
        public void testKeyReleased() throws AWTException,
           FactoryException{
                Robot robot = new Robot();
                ui.initialize();
                ui.start();
                try {
                robot.keyRelease(KeyEvent.VK_UP);
                } catch (Exception e) {
                fail("keyReleased failed");
                }
        }

        /**
         * Test method for KeyPressed(). We used
         * UI smoke testing to inspect that all the
         * operations are performed.
         *
         * The actions are:
         * Start the game -> sleep for 3 seconds ->
         * go up -> go down -> stop the game ->
         * sleep for 3 seconds -> start the game ->
         * go left -> go right -> sleep for 3 seconds
         * -> exit the game
         * @throws FactoryException
         */
        @Test
        public void testKeyPressed() throws AWTException, FactoryException
           {
                Robot robot = new Robot();
                ui.initialize();
                ui.start();
                try {
                robot.keyPress(KeyEvent.VK_S);
                Thread.sleep(3000);
                robot.keyPress(KeyEvent.VK_UP);
```

```
                robot.keyPress(KeyEvent.VK_DOWN);
                robot.keyPress(KeyEvent.VK_Q);
                Thread.sleep(3000);
                robot.keyPress(KeyEvent.VK_S);
                robot.keyPress(KeyEvent.VK_LEFT);
                robot.keyPress(KeyEvent.VK_RIGHT);
                Thread.sleep(3000);
                robot.keyPress(KeyEvent.VK_X);
                } catch (Exception e) {
                fail("KeyPressed failed");
                }
        }

}
```