Assignment 2: Coverage

Files Modified

Board Tests: (Exercises 2.4 and 2.5)

- https://github.com/UBC-TestingCourse/group4/blob/master/src/test/java/org/jpacman/test/framework/model/BoardTest.java
- https://github.com/UBC-TestingCourse/group4/blob/master/src/test/java/org/jpacman/test/framework/model/BoardWithinBordersTest.java

FactoryException Tests: (Exercise 2.8)

- https://github.com/UBC-TestingCourse/group4/blob/master/src/test/java/org/jpacman/test/framework/factory/FactoryExceptionTest.java

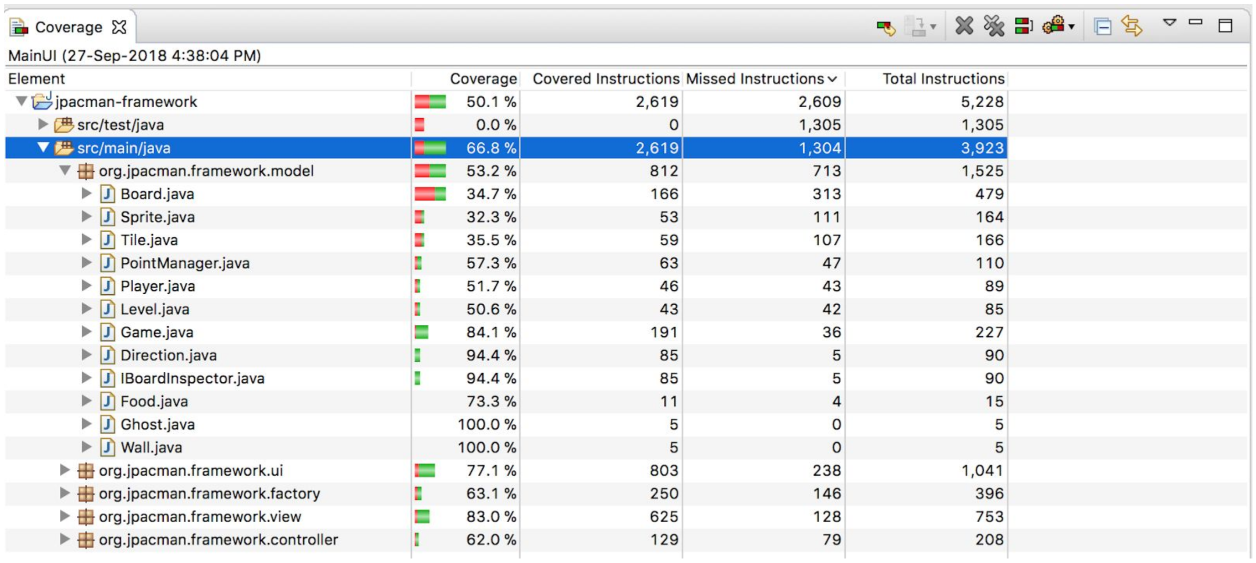
Sprite Tests: (Excercise 2.8)

- https://github.com/UBC-TestingCourse/group4/blob/master/src/test/java/org/jpacman/test/framework/model/SpriteTest.java

JUnit and Coverage

Exercise 2.1.

To determine coverage, we picked one method of ending the game: winning the game (collecting all the food), successfully completing the game at 1780/1780 points. We got an overall overage of 50.1%, which translates to about 66.8% of application coverage.



Coverage

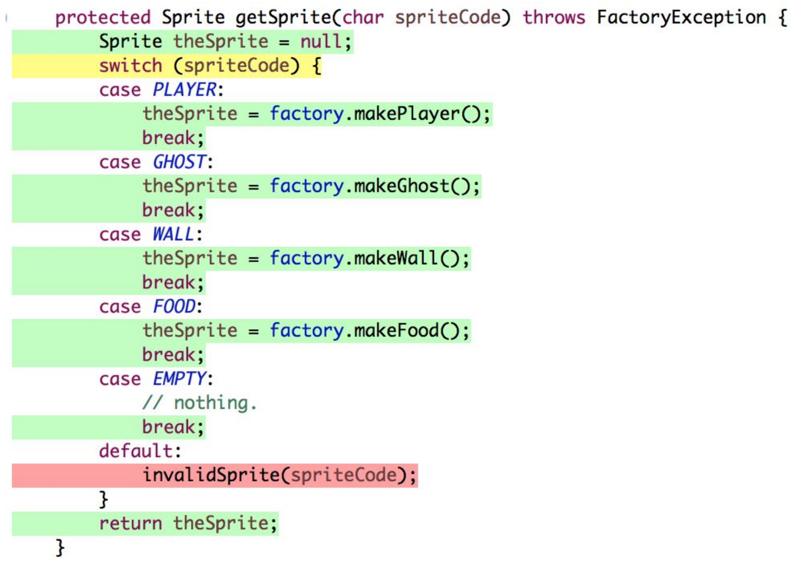
MainUI (27-Sep-2018 4:38:04 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
jpacman-framework	50.1 %	2,619	2,609	5,228
src/test/java	0.0 %	0	1,305	1,305
src/main/java	66.8 %	2,619	1,304	3,923
org.jpacman.framework.model	53.2 %	812	713	1,525
Board.java	34.7 %	166	313	479
Sprite.java	32.3 %	53	111	164
Tile.java	35.5 %	59	107	166
PointManager.java	57.3 %	63	47	110
Player.java	51.7 %	46	43	89
Level.java	50.6 %	43	42	85
Game.java	84.1 %	191	36	227
Direction.java	94.4 %	85	5	90
IBoardInspector.java	94.4 %	85	5	90
Food.java	73.3 %	11	4	15
Ghost.java	100.0 %	5	0	5
Wall.java	100.0 %	5	0	5
org.jpacman.framework.ui	77.1 %	803	238	1,041
org.jpacman.framework.factory	63.1 %	250	146	396
org.jpacman.framework.view	83.0 %	625	128	753
org.jpacman.framework.controller	62.0 %	129	79	208

Figure 2.1.1. EclEmma coverage from manual testing

Some interesting observations from EclEmma:

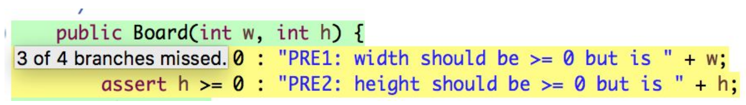
1. FactoryException.java was not run at all. Using Call Hierarchy, we found that the methods that instantiate FactoryException, such as InvalidSprite in MapParser.java, were never called. For example, the getSprite method calls InvalidSprite, but because a char with a default sprite code was never inputted, invalidSprite is never called. EclEmma reflects this in figure 2.1.2.



```
    protected Sprite getSprite(char spriteCode) throws FactoryException {
        Sprite theSprite = null;
        switch (spriteCode) {
        case PLAYER:
            theSprite = factory.makePlayer();
            break;
        case GHOST:
            theSprite = factory.makeGhost();
            break;
        case WALL:
            theSprite = factory.makeWall();
            break;
        case FOOD:
            theSprite = factory.makeFood();
            break;
        case EMPTY:
            // nothing.
            break;
        default:
            invalidSprite(spriteCode);
        }
        return theSprite;
    }
```

Figure 2.1.2. Coverage of getSprite function in MapParser.java shown by EclEmma

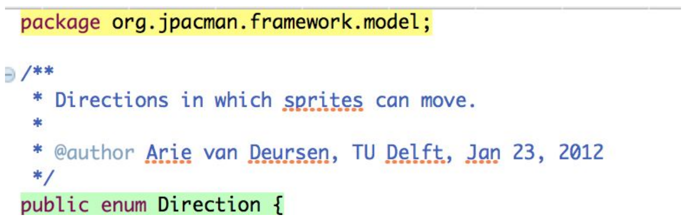
2. In Board.java, not all branches of the assert statements in lines 20 and 21 are executed. The assert statements are not actually run (since assertions are not enabled), so we only have a partial coverage of these lines. Assert statements, at runtime, have an additional if statement enclosing them to determine if asserts should be checked or not. Because the -ea parameter is not enabled here, the if condition returns false, never checking the assert.



```
    public Board(int w, int h) {
3 of 4 branches missed. 0 : "PRE1: width should be >= 0 but is " + w;
        assert h >= 0 : "PRE2: height should be >= 0 but is " + h;
```

Figure 2.1.3. Lines 19-21 from Board.java as covered by EclEmma

3. Coverage is at 94.4% in Direction.java, but all lines of code are fully covered except the package, as shown in figure 2.1.4.



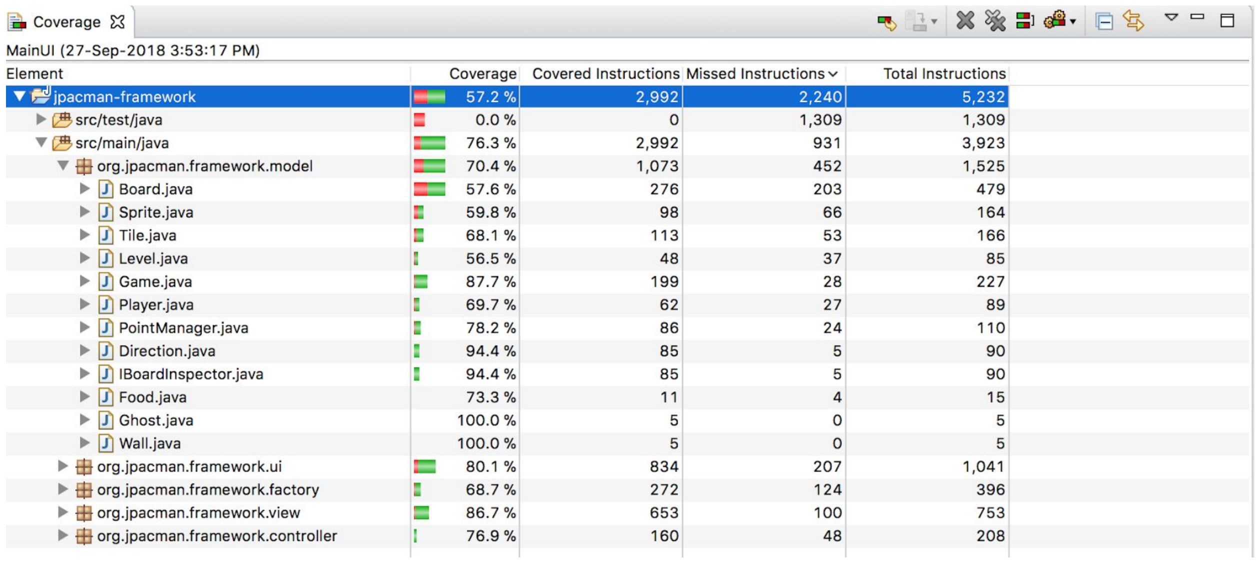
```
package org.jpacman.framework.model;

/**
 * Directions in which sprites can move.
 *
 * @author Arie van Deursen, TU Delft, Jan 23, 2012
 */
public enum Direction {
```

Figure 2.1.4. Package only partially covered in Direction.java

Exercise 2.2.

With the -ea argument, the overall coverage increased to 57.2%, which is a 76.3% application coverage. This is because assert statements are now enabled at runtime.

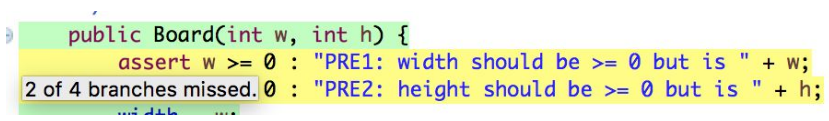


Coverage
MainUI (27-Sep-2018 3:53:17 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
jpacman-framework	57.2 %	2,992	2,240	5,232
src/test/java	0.0 %	0	1,309	1,309
src/main/java	76.3 %	2,992	931	3,923
org.jpacman.framework.model	70.4 %	1,073	452	1,525
Board.java	57.6 %	276	203	479
Sprite.java	59.8 %	98	66	164
Tile.java	68.1 %	113	53	166
Level.java	56.5 %	48	37	85
Game.java	87.7 %	199	28	227
Player.java	69.7 %	62	27	89
PointManager.java	78.2 %	86	24	110
Direction.java	94.4 %	85	5	90
IBoardInspector.java	94.4 %	85	5	90
Food.java	73.3 %	11	4	15
Ghost.java	100.0 %	5	0	5
Wall.java	100.0 %	5	0	5
org.jpacman.framework.ui	80.1 %	834	207	1,041
org.jpacman.framework.factory	68.7 %	272	124	396
org.jpacman.framework.view	86.7 %	653	100	753
org.jpacman.framework.controller	76.9 %	160	48	208

Figure 2.2.1. EclEmma coverage from manual testing with -ea option

If we consider Board.java, which we found interesting because only 1/4th of the instructions were run, the coverage for each assert has now increased. This is because we return true for assertions being enabled and reach the assert condition. However, the assert never fails, due to no invalid inputs being tested, so we still don't get full coverage from the assert statements.

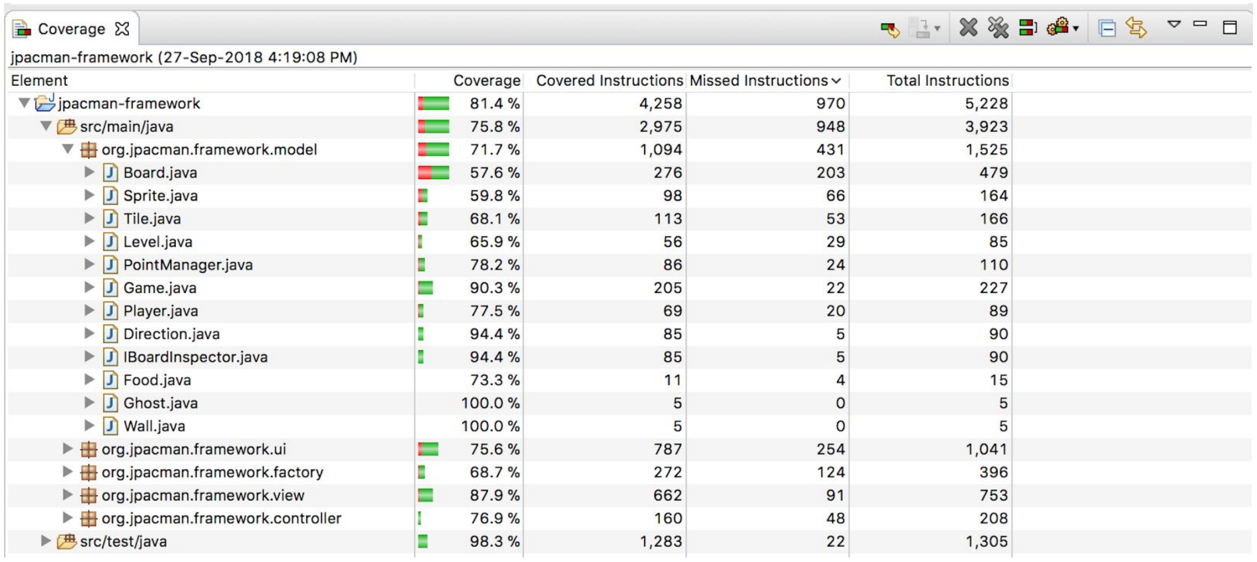


```
public Board(int w, int h) {
    assert w >= 0 : "PRE1: width should be >= 0 but is " + w;
2 of 4 branches missed.0 : "PRE2: height should be >= 0 but is " + h;
```

Figure 2.2.2. Lines 19-21 from Board.java as covered by EclEmma

Exercise 2.3.

When we run the JPacman test suite, we get a overall coverage of 81.4%. Note that this is with the -ea argument enabled in EclEmma. If the -ea option were to be removed, we get a lower coverage.

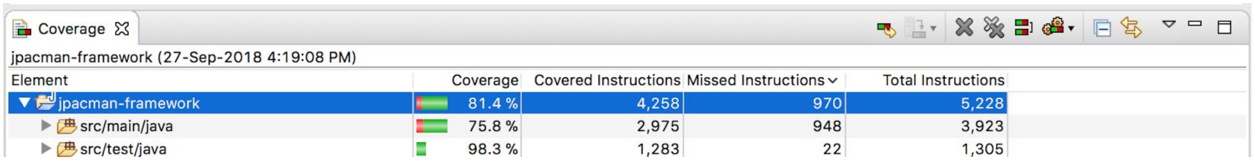


Coverage

jpacman-framework (27-Sep-2018 4:19:08 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
jpacman-framework	81.4 %	4,258	970	5,228
src/main/java	75.8 %	2,975	948	3,923
org.jpacman.framework.model	71.7 %	1,094	431	1,525
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
PointManager.java	78.2 %	86	24	110
Game.java	90.3 %	205	22	227
Player.java	77.5 %	69	20	89
Direction.java	94.4 %	85	5	90
IBoardInspector.java	94.4 %	85	5	90
Food.java	73.3 %	11	4	15
Ghost.java	100.0 %	5	0	5
Wall.java	100.0 %	5	0	5
org.jpacman.framework.ui	75.6 %	787	254	1,041
org.jpacman.framework.factory	68.7 %	272	124	396
org.jpacman.framework.view	87.9 %	662	91	753
org.jpacman.framework.controller	76.9 %	160	48	208
src/test/java	98.3 %	1,283	22	1,305

Figure 2.3.1. EclEmma coverage for test suite (expanded)



Coverage

jpacman-framework (27-Sep-2018 4:19:08 PM)

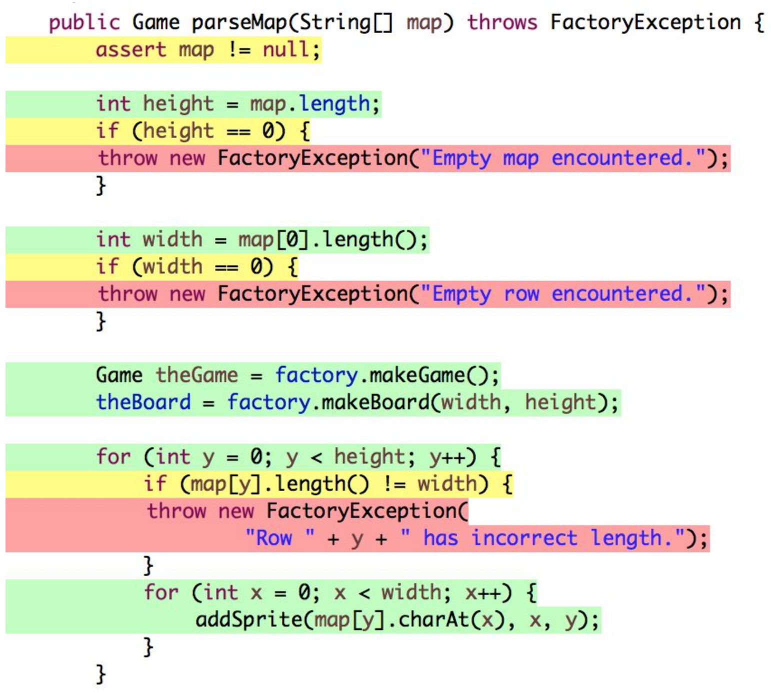
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
jpacman-framework	81.4 %	4,258	970	5,228
src/main/java	75.8 %	2,975	948	3,923
src/test/java	98.3 %	1,283	22	1,305

Figure 2.3.2. EclEmma coverage for test suite (condensed)

The application code covered is around **75.8%**, while we cover **98.3%** of the test code. A large part of the overall increase in code coverage is due to the test suite code being covered.

Of these three parameters, the **application code** is really the best indicator of the test suite's coverage. We are directly running the tests, so it's not surprising that 98.3% of the test code is covered. This number is combined with the 75.8% application code coverage to achieve an overall coverage of 81.4%, so overall coverage is also not a great representation of how much of the application the test cases cover.

An application code coverage of 75.8% indicates that there is about 25% of the code that is not tested by the test suite. For example, if we look in MapParser.java, we see that many factory exceptions are not thrown, so our tests do not try edge cases.



```
    public Game parseMap(String[] map) throws FactoryException {
        assert map != null;

        int height = map.length;
        if (height == 0) {
         throw new FactoryException("Empty map encountered.");
        }

        int width = map[0].length();
        if (width == 0) {
         throw new FactoryException("Empty row encountered.");
        }

        Game theGame = factory.makeGame();
        theBoard = factory.makeBoard(width, height);

        for (int y = 0; y < height; y++) {
            if (map[y].length() != width) {
             throw new FactoryException(
                        "Row " + y + " has incorrect length.");
            }
            for (int x = 0; x < width; x++) {
                addSprite(map[y].charAt(x), x, y);
            }
        }
```

Figure 2.3.3. Coverage of the parseMap function in MapParser.java

<u>JUnit and Coverage</u>

Exercise 2.4.

We added several test cases for each method in Board.java. The test class is available in *org.jpacman.test.framework.BoardTest.java*.

For initialization:

- Any positive value would normally create a board, so we want to also test invalid (negative values).
- An expected case (board with a positive width and height) was added to ensure that the board initialization functions as expected.
- The test cases we added provide combinations of invalid heights and widths and expected output to cover all possible errors.
- Since boards may have a size of 0 * 0, such a board was also created in a test to ensure that a 0 * 0 board can be successfully created.

For put and spriteAt methods:

- We can have null sprites or sprites that are already occupying other tiles, so some tests were written to cover these corner cases.
- We may request for a sprite from a tile with no sprite, so a test was written to ensure that this is correctly detected.

- Since a board tile can be specified for these methods, we also attempted invalid board tiles that were either negative in value or outside of the size of the board (e.g Tile 11 in a 10 * 10 board).

For tileAt methods:

- Since TileAtDirection is tested fairly extensively in another class, we only added a simple test case with one direction to ensure that the method functions as expected.
- The TileAt method allows invalid inputs that are outside of the board's boundaries, so these invalid inputs are tested along with some expected values (a tile within the board's boundaries).
- The tileAtOffset method is tested for positive and negative inputs
  - We created test cases where the offset is highly intuitive. For a given tile at (x,y) and an assigned (dx, dy) to offset the tile by, we get the tile at (x + dx, y + dy).
  - In some cases (x + dx, y + dy) would not actually be on the board. We expect that any additional tiles would be counted from tile (0, 0) of the board. That is, if x + dx and y + dy are both > the max width and height of the board, we would select the tile at (x + dx - max width, y + dy - max height).

Exercise 2.5.

Using the 1x1 domain testing strategy, we tested the inputs outlined in table 2.5.1. The test class is available in *org.jpacman.test.framework.BoardWithinBordersTest.java*.

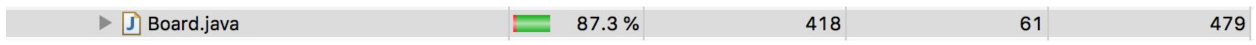
Boundary Conditions: x >= 0 && x < 10 && y >= 0 && y < 20;

Boundary			**Test Cases**							
Variable	**Condition**	**type**	**t1**	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**
x	>= 0	on	0							
		off		-1						
	< 10	on			10					
		off				9				
	typical	in					3	4	7	8
y	>= 0	on					0			
		off						-1		
	< 20	on							20	
		off								19
	typical	in	15	17	11	10				
Result			TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE

Table 2.5.1. 1x1 Matrix for the boundary condition x >= 0 && x < 10 && y >= 0 && y < 20

Exercise 2.6.

Our coverage increased to 87.3%. Originally this was 57.6%, so this is a significant increase.

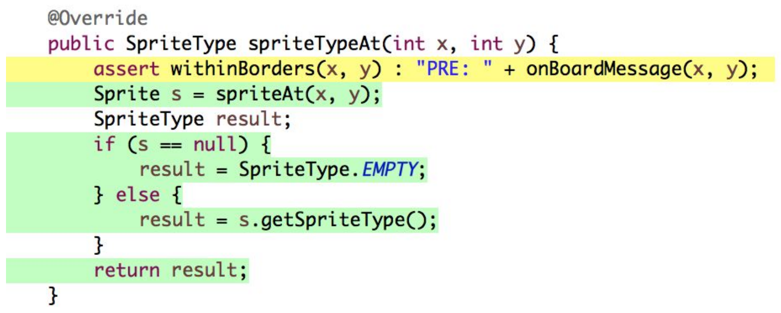


Board.java	87.3 %	418	61	479

Figure 2.6.1. Coverage of Board.java after adding new tests in exercises 2.4 and 2.5

Much of this is because:

- We now have tests that target error cases, so this means that assertions are not always false. Given that Board.java had many assert statements this increased coverage significantly.
- Some corner cases contained if statements. For example, the (s == null) branch of the spriteTypeAt method was originally never tested. Adding tests that covered these corner cases increased coverage.



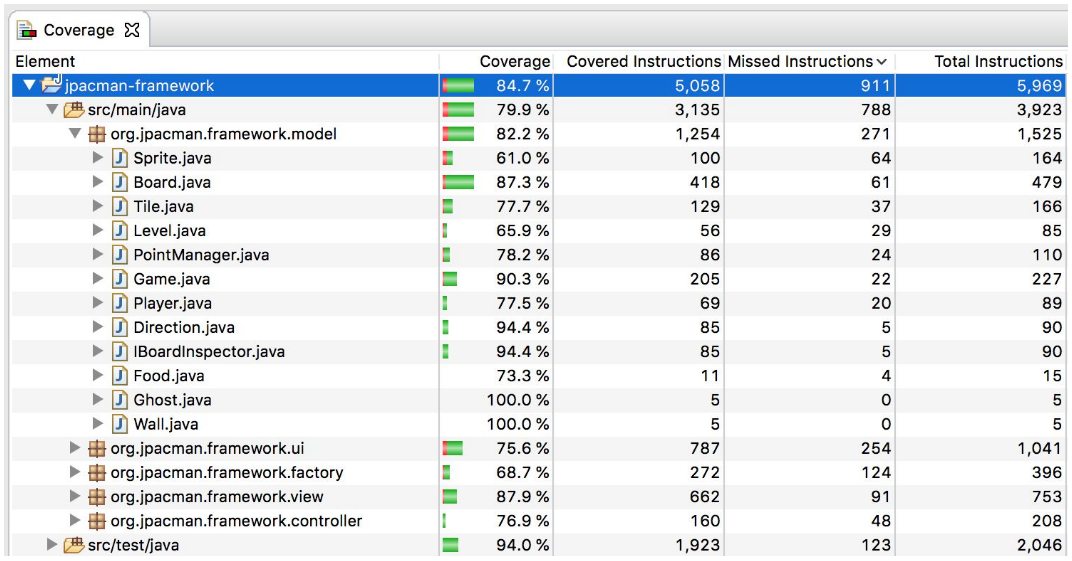
```
@Override
public SpriteType spriteTypeAt(int x, int y) {
    assert withinBorders(x, y) : "PRE: " + onBoardMessage(x, y);
    Sprite s = spriteAt(x, y);
    SpriteType result;
    if (s == null) {
        result = SpriteType.EMPTY;
    } else {
        result = s.getSpriteType();
    }
    return result;
}
```

Figure 2.6.2. Coverage of spriteTypeAt from Board.java with new tests

It is not possible to achieve full coverage for several reasons:

- We have the -ea parameter either enabled or disabled in any given run. This means at any given run of EclEmma, the if conditions for all asserts would either be true or false. At best, we will still miss one branch (where asserts are disabled) if we enabled asserts.
- Private methods are difficult to test. A private method like tunnelledCoordinate from Board.java is never called with invalid values from the source code, so the assert statements will always be true. The method that calls tunnelledCoordinate, tileAtOffset, has two asserts (lines 130-131) that would catch any invalid tiles passed in.
- Protected methods are similar to private methods in difficulty to test. The Board constructor catches all invalid inputs in its asserts before calling the tileInvariant method.
- Some asserts rely on values that are returned from other methods called internally. For example, the assert on line 140 confirms that the tileAt method does not return an invalid tile. Since the tunnelledCoordinate method never returns an invalid value, the assert will never be false.

Exercise 2.7


Coverage

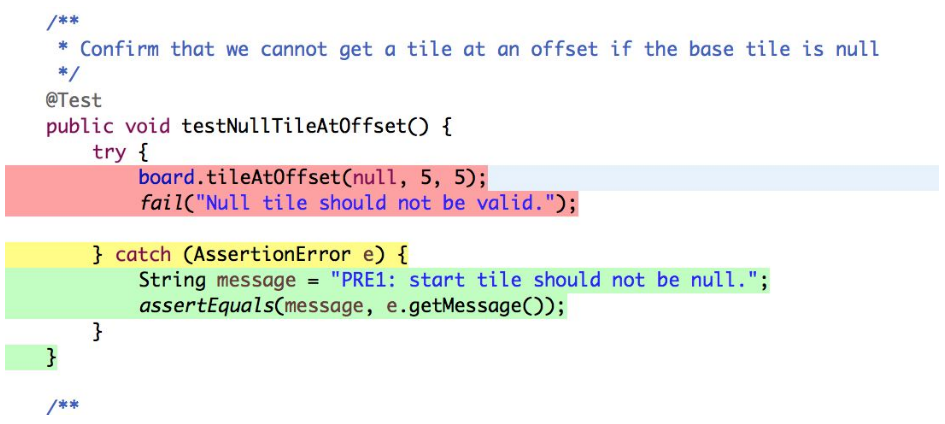
Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
jpacman-framework	84.7 %	5,058	911	5,969
src/main/java	79.9 %	3,135	788	3,923
org.jpacman.framework.model	82.2 %	1,254	271	1,525
Sprite.java	61.0 %	100	64	164
Board.java	87.3 %	418	61	479
Tile.java	77.7 %	129	37	166
Level.java	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
Direction.java	94.4 %	85	5	90
IBoardInspector.java	94.4 %	85	5	90
Food.java	73.3 %	11	4	15
Ghost.java	100.0 %	5	0	5
Wall.java	100.0 %	5	0	5
org.jpacman.framework.ui	75.6 %	787	254	1,041
org.jpacman.framework.factory	68.7 %	272	124	396
org.jpacman.framework.view	87.9 %	662	91	753
org.jpacman.framework.controller	76.9 %	160	48	208
src/test/java	94.0 %	1,923	123	2,046

Figure 2.7.1. Coverage after adding tests for Board.java

The coverage of the entire program has increased slightly at 84.7%. We see a slight increase in coverage of the source code, since Model.java is now covered more thoroughly. There are still many methods not nearly as well covered in the source code, so the impact is not large.

The coverage of the whole test suite is now 94%, which actually decreased from the original coverage of 98.3% in exercise 2.3. The reason for this is because we have several try catch blocks in our test code to confirm that we receive the correct assertion message from any erroneous input. We provide a fail clause to ensure that the try catch will fail if no assertion error was caught. This means that we expect some of the test code to not be run at all.

Figure 2.7.2. shows this phenomenon. Since we pass in a null sprite to tileAtOffset, we expect an assertion error to be thrown. As a result, tileAtOffset is not fully covered and the fail is never reached. This means that the test suite code is not fully run and covered.



```
/**
 * Confirm that we cannot get a tile at an offset if the base tile is null
 */
@Test
public void testNullTileAtOffset() {
    try {
        board.tileAtOffset(null, 5, 5);
        fail("Null tile should not be valid.");

    } catch (AssertionError e) {
        String message = "PRE1: start tile should not be null.";
        assertEquals(message, e.getMessage());
    }
}

/**
```

Figure 2.7.2. Coverage for testNullTileAtOffset for BoardTest.java

Exercise 2.8

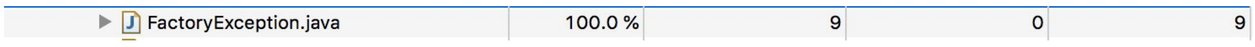
The least covered classes in JPacman with -ea parameter enabled are:

- FactoryException (0% coverage)
- PacmanKeyListener (14.6% coverage).
- Sprite (59.8% coverage)

PacmanKeyListener is particularly difficult to test, because it relies on key input from the user. Because of this, we chose to write tests for the class with the third lowest coverage, Sprite.

For the FactoryException class:

- We confirm that the two constructors properly inherit the super class Exception
- We check that our provided input is reflected in the new exception created (message and throwable type is correct)
- This brings us to 100% coverage

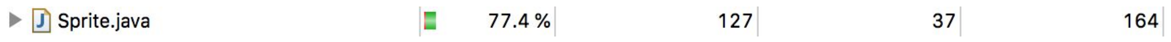


FactoryException.java	100.0 %	9	0	9

Figure 2.8.1. Coverage for FactoryException.java using our tests

For the Sprite class:

- Two methods, getSpriteType and toString were not covered by the original tests in SpriteTest. We added some tests that used these methods.
- The occupy and deoccupy methods did not check invalid inputs that would fail assertions in the implementation. We added tests that included these invalid inputs.
    - Not all invalid inputs could be generated. Some assert statements check for parameters that are generated within the method, and can never be false. This makes it impossible to cover failing branches of the assert statements.
- This resulted in a 77.4% coverage



Sprite.java	77.4 %	127	37	164

Figure 2.8.2. Coverage for Sprite.java using our tests