February 7, 2016 Group 14 members:
APPP514 Assignment 2 - Babatunde Akinniyi - Cyrus Sassani

Exercise 2.1:

Some general observations:
I noticed the hierarchy of test coverage results highlighted in green (100% code coverage), yellow (50% code coverage), red (0% code coverage).

The Project, packages, classes, and methods have different code coverage result percentages depending on how well their respective codes are covered in unit tests.

There are 5 different counters or measures of code coverage could be expressed in terms of their numbers of times covered, missed, and total percentage of coverage. These counters are: instructions, branches, lines, complexity, types, and methods.

Scenario-specific observations:
Playing the game without moving the pacman until he gets eaten by the ghostsmen result in the same code coverage – 53.3% even though length of time taken to lose game is different.

Exercise 2.2:
Assert statements confirm assumptions about the expected behavior of a program and by default they are disabled at run-time. The test coverage (%) increases when I enable assertions for coverage analysis of the JPacman codebase, because there are more lines of code that are compiled to confirm the validity of program. In some way, it's another form of test case execution outside of a test suite and test class. However, most of these assertions only have 50% test coverage, that is, partial coverage as highlighted in Yellow. This is because most test case values assert only the positive (successful) cases or control "branch" that results in "true" while ignoring other cases like the cases that results in an assertion failure – "false". If most of the assertions in the program result to a failure, we would have noticed a significant drop in coverage results, when assertions are enabled. This typically makes the overall test coverage result misleading when asserts are enabled because the tool is expecting some coverage for the negative path, even though they are really not part of the actual programs functionality or may be covered in another test method. On the other hand, when assertions are disabled, we find that right number of missed code branches from code functionality is highlighted and overall code test coverage is more precise.

Exercise 2.3:
The coverage for the test cases suite is very high at a **98.2%** rate because most instructions for all JUnit test classes are executed except for a few constructors and assertions statements which are typically partially covered.

The coverage for the project group as a whole (**74.6%**) is not the most accurate representation of the test suite coverage because it has coverage values from the test cases suite. Given that

Junit test cases typically have high coverage it provides an overall misleading higher coverage value for the whole project group coverage.

The best representative of the test suite coverage is the application code coverage of **66.8%** in which the test suite is run against demonstrating the precise amount of application code instructions executed for each test case in the test suite.

Exercise 2.4

BoardTest.java

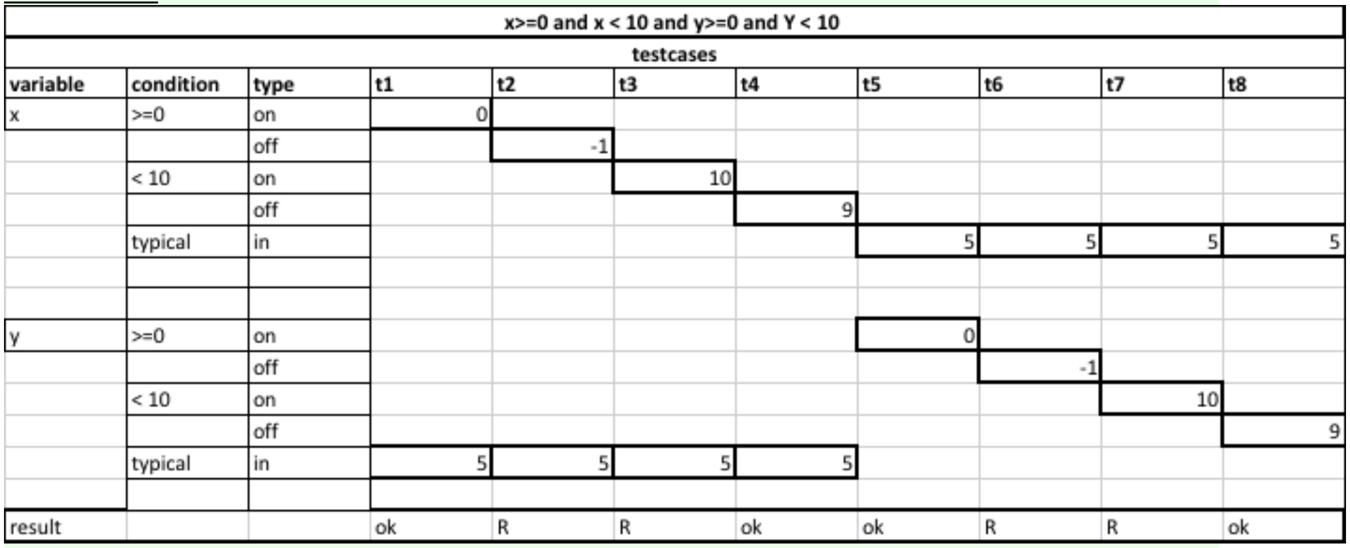
```
package org.jpacman.test.framework.model;


import static org.junit.Assert.*;


import org.jpacman.framework.model.Board;
import org.jpacman.framework.model.Ghost;
import org.jpacman.framework.model.IBoardInspector.SpriteType;
import org.jpacman.framework.model.Sprite;
import org.jpacman.framework.model.Tile;
import org.junit.Before;
import org.junit.Test;
import static org.hamcrest.CoreMatchers.equalTo;


public class BoardTest {


       private static final int width = 10;
       private static final int height = 10;

       private Board board;


       @Before
       public void setUp() throws Exception {
              board = new Board(width, height);

       }


       @Test
       public void testBoardTooWide() {


              try {
                new Board(1, -1);
        } catch (AssertionError e) {
                assertTrue(true);
        }
}

@Test
public void testBoardTooTall() {


        try {
                new Board(-1, -1);
        } catch (AssertionError e) {
                assertTrue(true);;
        }
}


@Test
public void testGetHeight() {
        assertEquals(height, board.getHeight());;
}

@Test
public void testGetWidth() {
        assertEquals(width, board.getWidth());;
}

@Test
public void testPut() {
        int x = 5;
        int y = 5;

        Ghost ghost = new Ghost();
        board.put(ghost, x, y);

        assertTrue(board.tileAt(x, y).containsSprite(ghost));


}

@Test
public void testPutNull() {
        try {
                board.put(null, 0, 0);
        } catch (AssertionError e) {
                assertTrue(true);
        }
}

@Test
public void testPutOutside() {
        Ghost ghost = new Ghost();


        try {
                board.put(ghost, -1, 0);
        } catch (AssertionError e) {
                assertTrue(true);
        }
}


@Test
public void testWithinBorders() {


        assertTrue(board.withinBorders(5, 5));

        assertTrue(board.withinBorders(0, 0));
        assertTrue(board.withinBorders(0, 9));
        assertTrue(board.withinBorders(9, 0));
        assertTrue(board.withinBorders(9, 9));

        assertFalse(board.withinBorders(0, 10));
        assertFalse(board.withinBorders(-1, 0));

}


@Test
public void testSpriteAt() {
        int x = 5;
        int y = 5;

        Ghost ghost = new Ghost();
        board.put(ghost, x, y);

        assertTrue(board.tileAt(x, y).containsSprite(ghost));
}


@Test
public void testSpriteTypeAt() {
        int x = 5;
        int y = 5;

        Ghost ghost = new Ghost();
        board.put(ghost, x, y);

        assertEquals(SpriteType.GHOST, board.spriteTypeAt(x, y));
}

@Test
public void testSpriteTypeAtOutside() {


        try {
                assertEquals(SpriteType.GHOST, board.spriteTypeAt(-1, 0));
        } catch (AssertionError e) {
                assertTrue(true);
        }


}


@Test
public void testPositiveTileAt() {
        int x = 5;
        int y = 5;

        Tile tile = board.tileAt(x, y);
        assertEquals(x, tile.getX());
        assertEquals(y, tile.getY());
}

@Test
public void testTileAtOutside() {


        try {
                board.tileAt(0, -1);
        } catch (AssertionError e) {
                assertTrue(true);
        }
}



@Test
```

```
        public void testOnBoardMessageFail() {


                int x = -1;
                int y = 5;

                Ghost ghost = new Ghost();
                try {
                        board.put(ghost, x, y);
                } catch (AssertionError e) {
                        assertTrue(true);
                }
        }

        @Test
        public void testTileAtOffsetFail() {
                int x = 5;
                int y = 5;

                try {
                        board.tileAtOffset(null, x, y);
                } catch (AssertionError e) {
                        assertTrue(true);
                }
        }

        @Test
        public void testTileAtOffsetOutside() {
                int x = 5;
                int y = 5;
                Tile tile = new Tile(11, 11);

                try {
                        board.tileAtOffset(tile, x, y);
                } catch (AssertionError e) {
                        assertTrue(true);
                }
        }


 }
```

```
testBoardTooWide – testing a failure condition when board width is outside size of board.
testBoardTooTall - testing a failure condition when board height is outside size of board.
testGetHeight – testing the getter function for board height.
testGetWidth - testing the getter function for board width.
testPut – test the success case for put a sprtite on a board.
testPutNull – putting a null sprite on a board should throw an exception
testPutOutside – test putting a sprite on a tile that doesn’t exist results in an error
```

```
testWithinBorders – test that a coordinate is valid for the board size
testSpriteAt – test that the sprite we put is at the right tile and location
testSpriteTypeAt – test that the type of sprite on a tile is valid
testSpriteTypeAtOutside – test that putting a ghost spritetype outside the board fails
testPositiveTileAt – test that retrieving the tile coordinates of a board is valid
testTileAtOutside – test that retrieving the tile coordinates outside of a board is invalid
testOnBoardMessageFail – test that the message is shown when an invalid path occurs on board
testTileAtOffsetFail – testing that a null tile at an offset location fails on board
testTileAtOffsetOutside – testing that a tile outside of board boundaries also fails
```

Exercise 2.5



x>=0 and x < 10 and y>=0 and Y < 10										
testcases										
variable	**condition**	**type**	**t1**	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**
x	>=0	on	0							
		off		-1						
	< 10	on			10					
		off				9				
	typical	in					5	5	5	5
y	>=0	on					0			
		off						-1		
	< 10	on							10	
		off								9
	typical	in	5	5	5	5				
result			ok	R	R	ok	ok	R	R	ok

```
@RunWith(Parameterized.class)
    public static class WithinBordersTest {

        private final int x, y;
        private Board board;

        private static final int width = 10;
        private static final int height = 10;

        public WithinBordersTest(int x, int y) {
            this.x = x;
            this.y = y;

            board = new Board(width, height);
        }

        @Test
        public void testWithinBorder() {
            assertTrue(board.withinBorders(x, y));
        }
        @Parameters
        public static Collection<Object[]> values() {
            Object[][] values = new Object[][] {
                            {0, 5},
                            {-1, 5},
                            {10, 5},
                            {9, 5},
                            {5, 0},
                            {5, -1},
                            {5, 10},
                            {5, 9}
                    };
            return Arrays.asList(values);
        }

    }
```

Exercise 2.6
A 100% test coverage is not necessary as it doesn't necessarily guarantee a 100% code quality. There could still be bugs in the program even with a 100% code coverage. Moreover, the effort it takes to get to a 100% code quality may not provide the value we need in code quality. Our current test coverage of the Board.java class is 81.8% is sufficient because it covers all the main control branch conditions for all instruction sets in the class. The methods are executed against multiple test values. The assert statements are particularly difficult to fully cover in the test class methods even with –ea (java assertions) enabled.

Exercise 2.7
There is a decrease in the test coverage of the entire test suite from 98.2% to 94.9%. This is because we added in the BoardTest.java that has a test coverage of 85.2% with a significant amount 459 instructions out of a total of 1764 test suite instructions resulting in a total impact of a 3.3% decrease. This doesn't mean our actual code coverage has been impacted negatively.

Exercise 2.8
The two classes with least code coverage are FactoryException.java (0%): 0/9 instructions set and PacmanKeyListener.java (6/41): 14.6%.
FactoryException: We added test methods that threw the exceptions in the two different ways exceptions could be thrown. Improved to a 100%!
PacmanKeyListener.java: We added test cases that tested keypress listener events for the up, down, right and left arrow keys as well as the start, stop, and exit buttons. Improved to a 100%.