Assignment 2

Kaibo Ma (32400129) -> kiddo122
David Henderson (19475128) -> D-H
Matthew Piaseczny (32993131) -> mpiasek

Exercise 2.1

Interesting Fact #1) When playing the game and testing the code coverage, there seemed to be a cap to the code coverage of the project. Regardless of if I play the game and only collect a couple pellets or if I complete the whole game the code coverage caps out at ~50%.

Interesting Fact #2) Looking through some of the classes some of the lines are never triggered. Upon closer inspection some methods have no references and could be removed with no effect. An example of this is the method setPoints in Food.java.

Interesting Fact #3) Looking through the class RandomGhostMove and the function doTick().

```
if (theGhost == null) {
        return;
}
```

This if statement is never evaluated to true in either playing the game or during running the test cases. This method might of been put in place at one time to catch some sort of coding error and is not longer needed because that bug has been fixed.

Exercise 2.2

While looking at the PointsManager class, with “-ea” variable there is more coverage than without. It has coverage about 55.5%. However there is still no coverage for the invariant function which looks at if the points are positive. This function is called by other functions in the class but it is always passing as false. Thus it never is able to run into the invariant() function.

Exercise 2.3

74.6% coverage for whole thing
66.8% for application code
98.2% for test cases
- Application code would be the best representative of the test suite coverage because eclemma sees what statements are executed, and the test suites would be the functions executing the source code. Thus, the test suite only covers 66.8% of the application code.

Exercise 2.4

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

import static org.junit.Assert.*;

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

import org.jpacman.framework.model.Board;
import org.jpacman.framework.model.Direction;
import org.jpacman.framework.model.Sprite;
import org.jpacman.framework.model.Tile;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class BoardTest {
	private final Sprite john = new Sprite() { };
	private final int startx, starty;
	private final Direction dir;
	private final int nextx, nexty;
	private final Board board;

	private static final int WIDTH = 10;
	private static final int HEIGHT = 20;
	/**
	 *
	 * @param x
	 * @param y
	 * @param d
	 * @param nx
	 * @param ny
	 */
	public BoardTest(int x, int y, Direction d, int nx, int ny) {
		startx = x;
		starty = y;
		dir = d;
		nextx = nx;
		nexty = ny;
		board = new Board(WIDTH, HEIGHT);
	}


	/**
	 * The actual test case.
	 */
	@Test(expected=AssertionError.class)
	public void testSpritAtBoard() {
		board.put(john, startx, starty);
```

```
            assertEquals(board.spriteAt(startx, starty), john);

            board.put(john, WIDTH+1, HEIGHT);
            assertEquals(board.spriteAt(WIDTH, HEIGHT), john);
            board.put(john, -1, -1);
            assertEquals(board.spriteAt(-1, -1), john);

    }

    @Test
    public void testWithinBoarder() {
            board.put(john, WIDTH-1, HEIGHT-1);
            assertTrue(board.withinBorders(john.getTile().getX(),john.getTile().getY()));

            Tile tile = new Tile(WIDTH+1, HEIGHT+1);
            john.deoccupy();
            john.occupy(tile);
            assertFalse(board.withinBorders(john.getTile().getX(),john.getTile().getY()));

            Tile tile2 = new Tile(WIDTH-1, HEIGHT+1);
            john.deoccupy();
            john.occupy(tile2);
            assertFalse(board.withinBorders(john.getTile().getX(),john.getTile().getY()));

            Tile tile3 = new Tile(WIDTH+1, HEIGHT-1);
            john.deoccupy();
            john.occupy(tile3);
            assertFalse(board.withinBorders(john.getTile().getX(),john.getTile().getY()));

            Tile tile4 = new Tile(0, 0);
            john.deoccupy();
            john.occupy(tile4);
            assertTrue(board.withinBorders(john.getTile().getX(),john.getTile().getY()));

            Tile tile5 = new Tile(-1, -1);
            john.deoccupy();
            john.occupy(tile5);
            assertFalse(board.withinBorders(john.getTile().getX(),john.getTile().getY()));

            Tile tile6 = new Tile(0, -1);
            john.deoccupy();
            john.occupy(tile6);
            assertFalse(board.withinBorders(john.getTile().getX(),john.getTile().getY()));

    }


    /**
     * List of (x,y)/Direction/(newx,newy) data points.
     * @return Test data to be fed to constructor.
     */
    @Parameters
    public static Collection<Object[]> data() {
            Object[][] values = new Object[][] {
```

```
                                        // x-axis boundaries, y random inpoints
                                        // left boundary
                                        { 2, 2, Direction.UP, 2, 1 },
                                        { 2, 2, Direction.DOWN, 2, 3 },
                                        { 2, 2, Direction.LEFT, 1, 2 },
                                        { 2, 2, Direction.RIGHT, 3, 2 },
                                        // worm holes
                                        { 0, 2, Direction.LEFT, WIDTH - 1, 2 },
                                        { WIDTH - 1, 2, Direction.RIGHT, 0, 2 },
                                        { 2, 0, Direction.UP, 2, HEIGHT - 1 },
                                        { 2, HEIGHT - 1, Direction.DOWN, 2, 0 } };
                return Arrays.asList(values);
        }

}
```

We added these test cases to ensure that all branches for withinboarder() and put() was covered. We did not test any of the methods without specifications as we interpreted this exercise as black box testing. There were many branches for put() so there are many boundary conditions we tested. Once withinBoarder() and put() was covered we were able to cover onBoardMessage() as it is called by all the functions if withinBoarder() through an exception.

Exercise 2.5

The board has width (x) = 10 and height (y) = 20

X Value	Y Value	Expected Outcome
5	19	True
5	20	False
5	21	False
5	-1	False
5	0	True
5	aw	True
9	15	True
10	15	False
11	15	False
-1	15	False
0	15	True
1	15	True

CODE BEGINS HERE

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

import static org.junit.Assert.assertEquals;

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

import org.jpacman.framework.model.Board;
import org.jpacman.framework.model.Direction;
import org.jpacman.framework.model.Tile;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import org.mockito.runners.MockitoJUnitRunner;

/**
 * Test the Board.withinBorder method by means of a series of test cases using the one by one
method.
 *
 * @author Group 2
 */
@RunWith(Parameterized.class)
public class BoardWithinBorderTest {

	private Board board = new Board(WIDTH, HEIGHT);

	private static final int WIDTH = 10;
	private static final int HEIGHT = 20;

	private int x;
	private int y;
	private boolean expectedValue;

	public BoardWithinBorderTest(Integer x, Integer y, Boolean expectedValue) {
		this.x = x;
		this.y = y;
		this.expectedValue = expectedValue;
```

```
    }

    @Parameterized.Parameters
      public static Collection xyValues() {
        return Arrays.asList(new Object[][] {
          { 5, 19, true },
          { 5, 20, false },
          { 5, 21, false },
          { 5, -1, false },
          { 5, 0, true },
          { 5, 1, true },
          { 9, 15, true },
          { 10, 15, false },
          { 11, 15, false },
          { -1, 15, false },
          { 0, 15, true },
          { 1, 15, true }
        });
      }

    @Test
  public void testCase() {
            boolean isWithinBorders = board.withinBorders(x, y);
            assertEquals(expectedValue, isWithinBorders);
  }

}
```

Exercise 2.6

After implementing the two new test classes, we were able to increase the coverage by a lot. Overal coverage: 84.0%, Main: 76.9% and Test 97.5%. We were unable to attain 100% because of the assert calls in Board class. We were able to handle them with @Test(expected=AssertionError.class) in the beginning on the test function However this does not increase the coverage by too much.

Exercise 2.7

Yes there was an increase, the application code coverage increased from 66.8% to 75.9%, the whole code coverage increased from 74.6% to 83.8%, and the test class coverage increased from 98.2% to 97.2%. This makes sense, seeing as how the test suite we wrote was aimed at covering code that had previously not been covered in Board.java. With Board.java being only one class, seeing a big increase in coverage from just one class is surprising.

Exercise 2.8

The 2 classes that are the least covered in JPacMan are:

- PacmanKeyListener (14.6%)
- FactoryException (0%)

The classes were improved to

- PacmanKeyListener (95.1%)
- FactoryException (44.4%)

This improved both classes code coverage by

- PacmanKeyListener (80.5%)
- FactoryException (44.4%)

For the PacmanKeyListener class the bulk of the class is in the keyPressed method. To test this method I used the MainUI class to get the PacmanKeyListener and then mocked the various key presses that the keyPressed expects. There is no output from the method so this is only a smoke test.

For the FactoryException class, the class consists of two constructors. To test the use of one of the constructors I had the game load a malformed map. This caused the exception to be thrown which forced the constructor to be called.