SENG275 – Lab 7
Due Wednesday March 17, 11:55 pm

Technical Note: Over the last few weeks, an increasing number of students have been having problems with dependencies in IntelliJ – imported classes are appearing in Red, and tests cannot be built. Student Taimur Khan found a solution that seems to be working for everyone – apparently IntelliJ's internal Java module cache is getting corrupted over time. Choosing **'Invalidate Caches / Restart'** (and then Just Restart) from IntelliJ's main menu (File menu on Windows) appears to be fixing the problem. Please let us know if you continue to have trouble.

Welcome to Lab 7. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked. The questions in **bold** will not be graded. However, you should give these questions some thought, as these are the sorts of questions that could appear on a quiz or midterm. This lab is worth 4% of your final grade.

Quick summary of what you need to do:

1. Using your completed decision table for JPacman Unit collisions, write tests for the PlayerCollision.collide() method in Exercise 1.
2. Choose ONE of the conceptual questions in Exercise 2, and answer it in Labs/Lab_07/readme.txt.
3. Complete the State Machine Model for JPacman in Exercise 3, and put an image of your diagram in Labs/Lab_07/
4. Translate your diagram to a State Transition Table in Exercise 4, and put an image of your table in Labs/Lab_07/

You're done!

Collision Testing

JPacman has three main classes of Unit – Player, Pellet, and Ghost (Ghost is further subclassed, but for the purposes of this lab we'll only be considering these three Unit types). These three types of gameplay object can collide with one another as the game proceeds, and the results of a collision are determined by the types involved. We would like to test this collision system.

The PlayerCollision.collide() method dispatches to various private methods depending on the classes of the colliding Units.

	Variants					
Unit 1	Pellet					
Unit 2	Pellet					
Consequence	None					

Fill out the table with decision table with the Units that could collide, and consult the PlayerCollision class to determine what consequences should result from each collision.

You do not have to hand this table in – use it to guide your testing:

Exercise 1.

For each column in the table, test that the appropriate outcome has occurred, and that no inappropriate methods have been called (refer to Lab 6 for a list of useful verification instructions). Write your tests in a file named test/…/level/PlayerCollisionsTest.java. (1%)

You may make the following simplifying assumptions:

1. The only Unit types you need to consider are Ghost, Pellet and Player.
2. Assume that collisions are commutative – that is, collide(a, b) is the same as collide(b, a).

Hints:

- You should test the outcomes of collisions that have no consequences (like Pellet-Pellet above).
- Using Mockito will make your tests shorter and easier to write.

You may use the following code to get started:

```
package nl.tudelft.jpacman.level;

import nl.tudelft.jpacman.npc.Ghost;
import nl.tudelft.jpacman.points.PointCalculator;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.*;

class PlayerCollisionsTest {

    PlayerCollisions cmap;
    PointCalculator pc;

    @BeforeEach
    void setUp() {
        pc = mock(PointCalculator.class);
        cmap = new PlayerCollisions(pc);
    }

}
```

Pragmatic Testing

Now that we've learned how to test, we need to consider the advantages and disadvantages of different testing methods. A good software testing practitioner should be able to make judgments about testing strategies, and support those opinions with reasons.

Exercise 2:

Choose ONE of the following questions, and answer it in the space provided in Labs/Lab_07/readme.txt (100 words maximum) (1%)

ANSWER ONLY ONE OF THE QUESTIONS. ADDITIONAL ANSWERS WILL NOT BE MARKED.

The other questions, though, would make good midterm questions.

- See the Ghost#randomMove() method. It makes use of Java's Random class to generate Exercise 2 random numbers. How would you test such method, if everytime you execute the method you get a different answer? Explain your idea (max 100 words)

- JPacman contains a test that can become a flaky test: see LauncherSmokeTest.smokeTest. Read the test and find out why this test can be flaky. Next, discuss other reasons why a test can become flaky and what can we do to avoid them.(max 100 words) You can read: https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html

- What is your opinion regarding achieving 100% of code coverage? What are the advantages? What are the disadvantages? How should one deal with such metrics, in your opinion?(max 100 words)

- You made intensive use of mocks in this assignment. So, you definitely know its advantages. But, in your opinion, what are the main disadvantages of such approach? Explain your rea sons. (max 100 words) You can read https://8thlight.com/blog/uncle-bob/2014/05/10/WhenToMock.html and http://www.jmock.org/oopsla2004.pdf

- Our test suite is pretty fast. However, the more a test suite grows the more time it takes to execute. Can you think of scenarios (more than one) that can lead a single test (and eventually the entire test suite) to become slow? What can we do to mitigate the issue?(max 100 words)

- There are occasions in which we should use the class' concrete implementation and not mock it. In what cases should one mock a class? In what cases should one not mock a class? Hint: Think about the test level (unit, integration, system testing). You can also read the following paper, if you are curious about how mock objects evolve over time: https://bit.ly/2HMVHGH

These are subjective questions, and don't have one specific, correct answer. Professional testers may disagree about the answers – expert opinion is valuable only because those opinions are usually supported with Good Reasons. Don't worry about figuring out the 'right' answer to the question you decide to answer – instead, do your best to support your argument with good, persuasive reasons.

State Machines

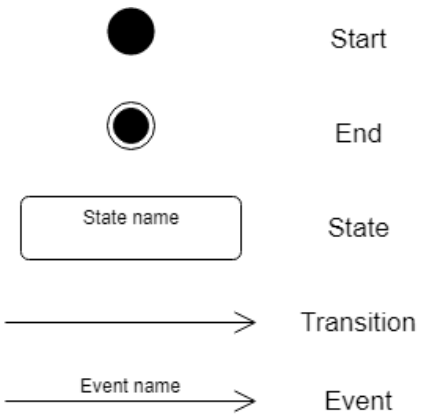
Consider the following prose description of a game of JPacman:

> The game is first created. At that point, only clicking the start button causes anything to happen.
>
> If the start button is clicked, the game is now playing. The players and ghosts begin moving, and continue moving until stop is clicked, the player and a ghost collide, the player eats the last pellet, or the stop button is clicked.
>
> If the stop button is clicked, the game is halted, until the start button is clicked, in which case it goes back to playing.
>
> If the player and a ghost collide, the game is lost.
>
> If the player eats the last pellet, the game is won.

Exercise 3:

Create a State Machine Model diagram that depicts the behaviour of JPacman. Use the above description, and you should play JPacman itself if you want to remind yourself how the state transitions work.

Use the following UML symbols for your state machine diagram:



Use whatever method you wish to create the diagram – you may use a drawing program or an online app such as draw.io, or draw your diagram on paper and scan it or take a picture of it with your phone. Whatever means you use, place the resulting image file in your Labs/Lab_07 folder. Any image format will be acceptable.

Exercise 4:

Convert the state machine model diagram you've created into a State Transition Table.

Remember, in such a diagram, the States that your program can be in are written down the left side, and the Events that cause state transitions appear across the top. Write the states that correspond to each state-event pair in the cells of the table where appropriate, and leave the other cells blank.

You may use the following incomplete table as a guide if you wish:

	Start clicked	Stop clicked			
CREATED					
PLAYING					

Add an image of your table to Labs/Lab_07/ as in Exercise 3 above.