SENG275 – Lab 9/10
Due Sunday July 16, 11:55 PM

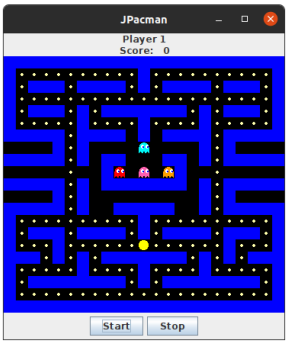
Technical note – you now have a Brightspace folder named lab09_10.

Welcome to Labs 9 and 10, which together are worth 4% of your final grade. In this lab you will inspect a piece of software named Jpacman – a partially-implemented Pacman clone written in Java. Once you feel you understand the structure of the program, you will write tests to confirm its function. You may freely decide both which portions of the program to test, and what testing methodology to employ. Your mark will depend on whether or not you have properly implemented your tests, as well as the number of different methodologies you've chosen, and their applicability to the code being tested.

Jpacman

Jpacman was developed by Delft University of Technology in the Netherlands for their software testing course. If you're unfamiliar with Pacman, the game it is based on, your character (Pacman) moves around a maze filled with dots and ghosts. When Pacman collides with a dot (called a Pellet in the game), he eats it, and gains points. If Pacman collides with a ghost, he dies and the game ends. If Pacman eats all the pellets, he wins.

Try running the game – in src/main/java, you'll find a class named Launcher. Run this class.



Press the Start button and the ghosts will begin chasing you. You may move North, South, East or West using the arrow keys.

Explore

Once you're finished playing, begin exploring the project. There are three source code folders of note – **default-test**, **main**, and **test**. The main folder contains the source code for the game itself; implementations of the board, the game state, the graphics, the ghosts (in the npc folder), etc. The **default-test** folder contains 39 tests that have already been written for you – try running them. To do so, right-click on the **default-test** folder and choose **Run tests in…**

Create Tests

The **test** folder is where you'll write your test code. There are two classes already here for you to use if you wish, to test methods of the Direction and Unit classes. You'll want to create your own test classes – the easiest way to do this is to find some class or method in **main** that you wish to test, and right-click that code. Choose **Generate…** from the drop-down menu, then **Test**. Select a method if you wish, then press OK.

For example, suppose we wished to test the isAlive() function in level/Player. Find the implementation code, right click and choose **Generate,** then **Test**. Select the isAlive() function from the box, and click **OK** (we want git to track our test code, so say yes to that question). Now you have a new file in the test folder, under **test/java/nl.tudelft.jpacman/level**.

By default, IntelliJ will import our junit.api.Test annotation, as well as the basic Junit assertions. Depending on the types of tests we wish to write, though, we may need more.

Testing Methodologies

Recall the various testing methodologies we've tried out throughout the term:

- Unit testing (Junit with @Test and @ParameterizedTest)
- Coverage testing in IntelliJ
- Specification testing
- Mutation testing with PiTest
- Integration testing with Mockito
- Property testing with Jquik
- System testing (smoke tests)
- Manual testing (scripted and exploratory)

Imports

For some of these testing methodologies, you'll need to import libraries into your test class:

Junit:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.params.*;
import org.junit.jupiter.params.provider.*;
import java.util.stream.*;
```

AssertJ:

```
import static org.assertj.core.api.Assertions.*;
```

PiTest:

Mutation testing has been enabled, and as usual may be run through **tasks → verification → pitest** in the Gradle pop-out box on the right edge of the screen.

Mockito:

```
import org.mockito.Mockito;
import static org.mockito.ArgumentMatchers.*;
```

Jqwik:

```
import net.jqwik.api.*;
import net.jqwik.api.arbitraries.*;
import net.jqwik.api.constraints.*;
```

Deliverables:

Test as much as you can. Diversity of testing methods is more important than quantity – a submission with a hundred unit tests will receive a lower grade than one with some unit tests, some integration tests, some coverage testing and a manual test.

Write your code in the **test** folder. If you have documents that you wish to submit (for example, observations during exploratory testing, or a script for scripted testing, or screenshots of mutation or coverage tests), place them in the **test/resources** folder.

REMEMBER TO ADD, COMMIT AND PUSH YOUR FILES TO YOUR REMOTE REPOSITORY!

Ideas:

- board/DirectionTest is incomplete
- board/OccupantTest is incomplete. It looks like we're trying to figure out if a BasicUnit, when created, has no square to occupy. Then we're trying to figure out if unit knows its occupying a square if we make it occupy one. Then we're trying to move the unit out of a square, then back in, and seeing if it knows it's still occupying the square. What do all these words mean? What's a unit? What's a square? What can they do? You'll need to read the implementation to find out!

- Each of the four ghosts has its own separate AI routine, implemented in a function called nextAiMove(). Given the specifications of these ghosts, can you test that they do indeed follow that specification? Can you put the ghost into some situation (perhaps using a custom map built using a GhostMapParser?) If you know the situation the ghost is in, you should be able to ask it for its next move, and check to see if that's what you expect.
- The MapParser class is pretty complex – it has a variety of dependencies. Perhaps you can use Mockito to isolate it from those dependencies, and then test one of its methods. ParseMap() looks like a good place to start – can you get that function to produce a map comprised of just one wall? (mapParser.parseMap(List.of("#"))) If so, does this class interact with its dependencies correctly?
- What about coverage tests? Take a class like Game.start(). It's possible to write tests to achieve 100% branch coverage on this method. Give it a try!
- Can you find in the code where collisions between different game entities are handled? What are the expected results for each type of collision? (ghosts colliding with players are handled differently than ghosts colliding with ghosts, for example). How can this be tested?
- When we wish to validate the program (confirm that it meets the user's specification), we can use an Acceptance Test – a series of tests that confirm that some specification is reached. Try to write code to verify the following user story:

  Title: suspend the game
  As a player,
  I want to be able to suspend the game
  So that I can pause and do something else

  Scenario 1:
  Given the game has started;
  When the player clicks the "Stop" button;
  Then the game is not in progress.