SENG275 – Lab 4
Due Wednesday Feb 24, 11:55 pm

Welcome to Lab 4. 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. Determine the boundary values and partitions necessary to test Clyde's AI. Implement these tests in ClydeTest.java, as shown on page 6. (worth 1.5%)
2. Using the same technique, write tests for Inky's AI in InkyTest.java, as shown on page 7. (worth 1.5%)
3. Determine the boundary values and equivalence partitions for Board.withinBorders(). Write a parameterized test in BoardTest.java that sufficiently explores these partition categories. (worth 1%)
4. Commit your changes to the JPacman repository, and push the commits back to gitlab.

   You're done!

This week we'll be combining boundary and partition analysis with unit tests. We will do this to test the artificial intelligence of two of the Ghosts in JPacman, as well as a method to check the size of a level.

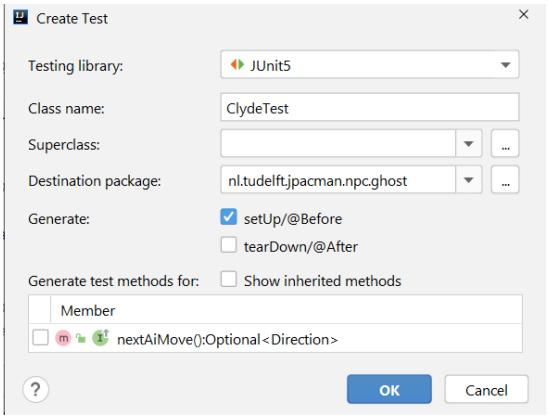
Setting up Clyde's AI

The ghosts are implemented in the npc -> ghost folder, and each is subclassed from the Ghost class. Each ghost has its own unique AI, and the expected behaviour of each is documented in that ghost's class. For example, in Clyde.java, we read:

> Clyde has two basic AIs, one for when he's far from Pac-Man, and one for when he is near to Pac-Man. When Clyde is far away from Pac-Man (beyond eight grid spaces), Clyde behaves very much like Blinky, trying to move to Pac-Man's exact location. However, when Clyde gets within eight grid spaces of Pac-Man, he automatically changes his behavior and runs away.

We would like to test Clyde's behaviour. It is impractical to do this during a running game, since it's difficult to measure distances on the screen, hard to reproduce exact setups, and the game cannot be paused. So we will create an artificial scenario – we will create a very simple level for PacMan and Clyde to occupy, in the exact positions we specify, and we will ask Clyde which direction he wants to move, by calling his nextAiMove() method.

Create a test class for Clyde. You can do this manually, but it's quicker to let IntelliJ do it – go into Clyde.java, and find the class declaration (line 42). Click on line, right click and choose Generate -> Test. We'd like to accept the generated name, and we'd also like a @BeforeEach created:



Click ok, and choose to add the new file to Git in the pop-up box that follows. Find the new file in your test package.

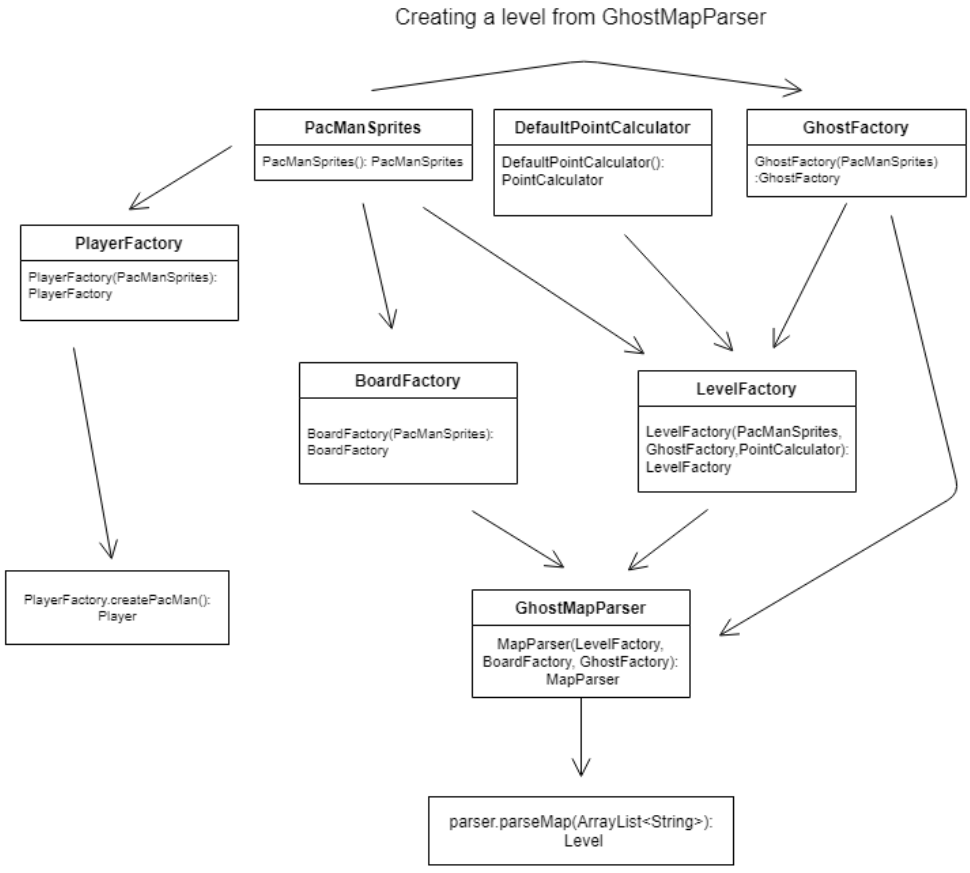
We would like to create the simplest possible level that tests Clyde's behaviour. What we would like is a level that looks something like this:

```
############
P........C.
############
```

That is, we would like a simple, straight corridor with walls on either side. PacMan stands at the west end of the corridor, and Clyde stands at the east end. Clyde is free to move either EAST or WEST, and we want to see what he chooses to do, based on the distance between P and C.

Fortunately, JPacman allows us to created a level in exactly this way. We can specify a level layout by passing an ArrayList of Strings to a GhostMapParser. That will give us a Level object back, which we can then use for our testing.

Unfortunately, creating this GhostMapParser isn't easy – the constructor for GhostMapParser takes other objects as inputs, which in turn require other objects for their instantiation. The following diagram expresses these relationships:



If you recall from CSC225, the above is a DAG (a directed acyclic graph), and so there is a procedure to determine the order in which we should instantiate the various classes:

1. Instantiate any node whose in-degree is zero (no incoming arrows).
2. Erase that node and all its out-edges. Repeat from 1.

That sounds like a hassle, though, so here's the setUp code required (paste into your ClydeTest class):

```
private GhostMapParser parser;
private Player pacman;

@BeforeEach
void setUp() {
    PacManSprites sprites = new PacManSprites();
    PointCalculator pointCalculator = new DefaultPointCalculator();
    GhostFactory ghostFactory = new GhostFactory(sprites);
    BoardFactory boardFactory = new BoardFactory(sprites);
    LevelFactory levelFactory = new LevelFactory(sprites, ghostFactory, pointCalculator);

    parser = new GhostMapParser(levelFactory, boardFactory, ghostFactory);
    PlayerFactory playerFactory = new PlayerFactory(sprites);
    pacman = playerFactory.createPacMan();
}
```

This gives us our GhostMapParser, and creates our PacMan unit (although it doesn't add him to the level yet, which hasn't been created. We'll create a slightly different level for each test, so we'll leave that to the individual tests.

Writing the tests for Clyde

We need to figure out what tests to write. We'll use the category-partition method, and we know from reading the documentation that our boundary points are 8 and 9 – at a distance of 8 or less, Clyde behaves one way, and at distance of 9 or more, he behaves another. We also know from skimming over Clyde's nextAiMove() method that he decides not to move at all (the method returns Optional.empty()) if either there is no player, or if there is no valid path to the player. That gives us three separate inputs, each of which is bivalent – it can be either yes or no. Let's call them "playerExists?", "ValidPath?" and "FarAway?".

PlayerExists?	ValidPath?	FarAway?	Expected action
Y	Y	Y	Move towards
Y	Y	N	
Y	N	Y	
Y	N	N	
N	Y	Y	
N	Y	N	
N	N	Y	
N	N	N	

Complete the chart yourself. As you can see, since there are three bivalent inputs, there are 2^3 rows, and the entries in the rows are just the cartesian product of the possibilities for each input.

Each of these may not correspond to a particular test. From the above chart, you will not need more than 8 tests, but fewer may be ok. For example, are four tests required (or even possible) for the case where the player doesn't exist?

The first test appears below.

```
@Test
void testReachableFar() {
    ArrayList<String> map = new ArrayList<>(List.of(
            "#############",
            "#P........C.",
            "#############"
    ));

    Level level = parser.parseMap(map);
    Clyde clyde = Navigation.findUnitInBoard(Clyde.class, level.getBoard());

    level.registerPlayer(pacman);
    pacman.setDirection(Direction.EAST);

    Optional<Direction> next = clyde.nextAiMove();
    assertTrue(next.isPresent());
    assertSame(Direction.WEST, next.get());

}
```

As you can see, we create our arrray of Strings for the setup we want to test, then we pass it to our parser. We have one Clyde in this scenario, so we find him, and we want one PacMan, so we register him and set him facing EAST.

We then ask Clyde which direction he wants to move, confirm that he answered with an actual direction (and not Optional.empty(), which means "no direction"). Finally we get the direction, and compare it to WEST, which is what we expect, given the distance of 9, which is our boundary value for "far away".

Digression: Optional

It would be simpler if nextAiMove() just returned a DIRECTION object, and if there was a DIRECTION corresponding to "no direction". Instead, the JPacman developers decided to return an object of type Optional<DIRECTION>. Optional is a *wrapper* around a DIRECTION – a variable of type Optional<DIRECTION> might contain one of the DIRECTION enums, or it might contain Optional.empty().

Optional was added to the language in Java 8, and was intended to get around the whole 'null pointer exception' problem in Java. If "no direction" was just null, nextAiMove() would have to

pass a null back to whoever called it, and if the caller misused that null, the result would be a runtime error. By wrapping null inside an object and preventing direct access to it, certain types of runtime errors become more unlikely.

Optional is a powerful and flexible tool, with a lot of interesting uses, especially in Java that uses lambda expressions and a functional style. For our purposes, however, we only need to know:

- nextAiMove() returns an Optional<DIRECTION>
- we can assign it to a variable: Optional<DIRECTION> opt = clyde.nextAiMove();
- If it contains Optional.empty() (ie, "no direction") then opt.isEmpty() will return true, and opt.isPresent() will return false.
- Otherwise, we can find out what it contains with opt.get(). The result will be one of the DIRECTION enums (EAST, WEST, NORTH, SOUTH)

Exercise 1:

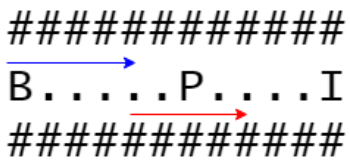
Complete the chart above, and use it to write the remaining tests for Clyde's nextAiMove method. There will be between 1 and 8 valid testable possibilities; consider which ones you need to test and which ones you don't. These tests must appear in ClydeTest.java, in your npc.ghost directory in the test package. You do not need to submit the chart.

Testing Inky

Test Inky in the same way as you did Clyde. The relevant parts of Inky's documentation are:

> Inky has the most complicated AI of all. Inky considers two things: Blinky's location, and the location two grid spaces ahead of Pac-Man. Inky draws a line from Blinky to the spot that is two squares in front of Pac-Man and extends that line twice as far. Therefore, if Inky is alongside Blinky when they are behind Pac-Man, Inky will usually follow Blinky the whole time. But if Inky is in front of Pac-Man when Blinky is far behind him, Inky tends to want to move away from Pac-Man (in reality, to a point very far ahead of Pac-Man).
>
> To actually implement this in jpacman we have the following approximation: first determine the square of Blinky (A) and the square 2 squares away from Pac-Man (B). Then determine the shortest path from A to regardless of terrain and walk that same path from B. This is the destination.

Inky's AI is more complex, and depends on the behaviour of Blinky, a separate ghost. First, if either the player or Blinky don't exist, Inky does nothing. If Blinky can't find a path of any length (regardless of terrain) to the player, Inky also does nothing. Otherwise, draw an arrow from Blinky to the spot two squares in front of PacMan, then extend that arrow the same distance again. The end of that arrow is what Inky wants to move towards.



Example: Suppose PacMan is facing WEST. We draw an arrow (here in Blue) to the square two squares in front of PacMan. We then extend the arrow by the same distance in the same direction (here in Red). This is a square three places to the west of Inky, so he decides to move WEST.

Exercise 2

Write three tests for Inky's nextAiMove(). Write one test for each of the following outputs:

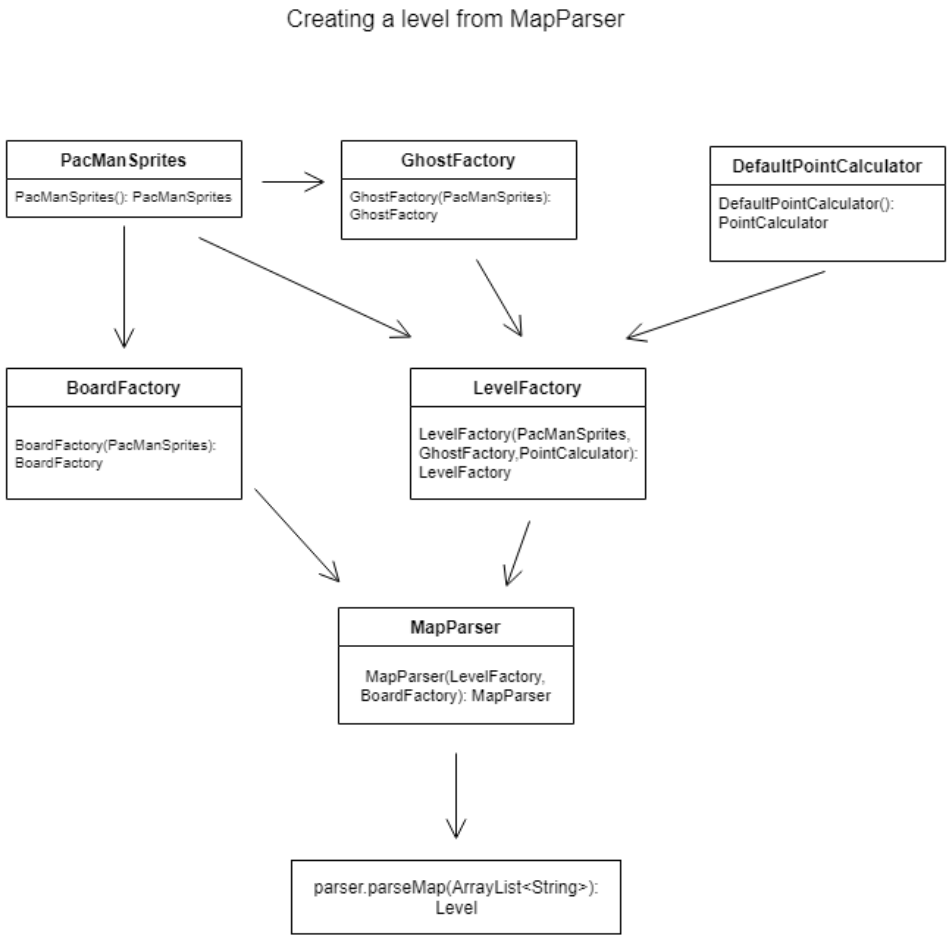
- Inky decides not to move
- Inky decides to move closer to PacMan
- Inky decides to move away from PacMan

Many other tests are possible, but three will be sufficient. In order to parse a map that contains Inky and Blinky, you will need to modify the GhostMapParser class to recognize these other two ghosts. These tests must be written in InkyTest.java, in the npc.ghost folder in your test package.

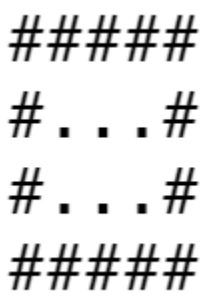
Board.withinBorders()

Board objects have a withinBorders() method that simply returns true if a given x,y coordinate is within the board, and false otherwise. Create a test for this function (again, a short-cut to do so is to right-click on the declaration of the function you want to test, and choose Generate -> Test from the right-click menu). Give the function a better name – the suggested 'BoardTest' is too vague.

An empty board has no need for ghosts or players, so you can use the simpler MapParser class if you prefer:



The board to test is 4 rows high by 5 columns wide, and looks like this:



To create this board, and put it in the class variable 'board', add the following to your test class:

```
private MapParser parser;
private Level level;
private Board board;

@BeforeEach
void setUp() {
    PacManSprites sprites = new PacManSprites();
    PointCalculator pointCalculator = new DefaultPointCalculator();
    GhostFactory ghostFactory = new GhostFactory(sprites);
    BoardFactory boardFactory = new BoardFactory(sprites);
    LevelFactory levelFactory = new LevelFactory(sprites, ghostFactory,
pointCalculator);

    parser = new MapParser(levelFactory, boardFactory);
    ArrayList<String> map = new ArrayList<>(List.of(
            "#####",
            "#...#",
            "#...#",
            "#####"
    ));
    level = parser.parseMap(map);
    board = level.getBoard();
}
```

Exercise 3:

Identify the x,y boundary points for this method (the method considers x=0,y=0 to be the upper left square of the board). Write a parameterized test to test all the boundary points. You will need to pass multiple arguments to the test at a time; read section 2.2 of the textbook (Boundary Testing) – the section "Automating Boundary Testing with Junit" discusses the use of the @CsvSource annotation. For example, if I expect x=-1, y=-1 to produce the output false, I could pass "-1,-1,FALSE" to my test using @CsvSource.