



# SOFT2201/COMP9201

## Assignment 2

**Due: Sunday 22 September 2024, 11:59PM AEST**

*This assignment is worth 15% of your final assessment*

**IMPORTANT:** You are only allowed to implement Assignment 2 Requirements in this assignment. Do not include any other features, otherwise mark deductions will be applied.

**IMPORTANT:** You must use the template we have provided and only the JavaFX library for your GUI. You may not use any other GUI library.

### Task Overview

In assignment 2, you will implement your Pac-Man game and refactor your UML class diagram according to your implemented code. You can find a general description of the Pac-Man game in the Assignment 1 description. You must ensure that your game is configurable with a JSON configuration file. You **must use the following GoF design patterns** in your implementation as requested below:

- **Observer Pattern:** To update the UI of the game window, including the score, the number of lives Pac-Man has, and the GAME OVER, YOU WIN and READY screens.
- **Factory Method Pattern:** All game entities (including walls, pellets, Pac-Man and the ghosts) will be constructed using the Factory Method Pattern
- **Singleton Pattern:** Identify class(es) in your implementation that are naturally singletons and require global access. Implement these classes using the Singleton pattern to ensure that only one instance of each class exists throughout the application.
- **Command Pattern:** To handle keyboard input from the Player and move Pac-Man up, left, right, and down

You can use other design patterns wherever appropriate.

## What we provide to you

In the [code scaffold](#) given to you, you will be able to find:

- **JSON file:** An example configuration file (`config.json`)
- **Map file:** An example map text file (`map.txt`)
- **build.gradle file:** A sample `build.gradle` file
- **Code Scaffold:** A scaffold codebase is provided to help you get started. You can modify this as much or as little as you like.
  - The scaffold is not runnable in its current state. You will need to do some modifications before it is able to be run.
  - Note: The scaffold uses pixel collision rather than grid collision. You may implement collision however you wish.
  - You must use JavaFX for your GUI - no other GUI library is allowed.
- **Sprites:** You will be using these to render the game entities. These can be found in `src/main/resources`.
- **Font:** The font used to display the game messages and score. This can be found in `src/main/resources`.

Note, to download the scaffold, open the File Explorer on the EdStem code submission page, and right-click to Download all files.

## What we expect from you

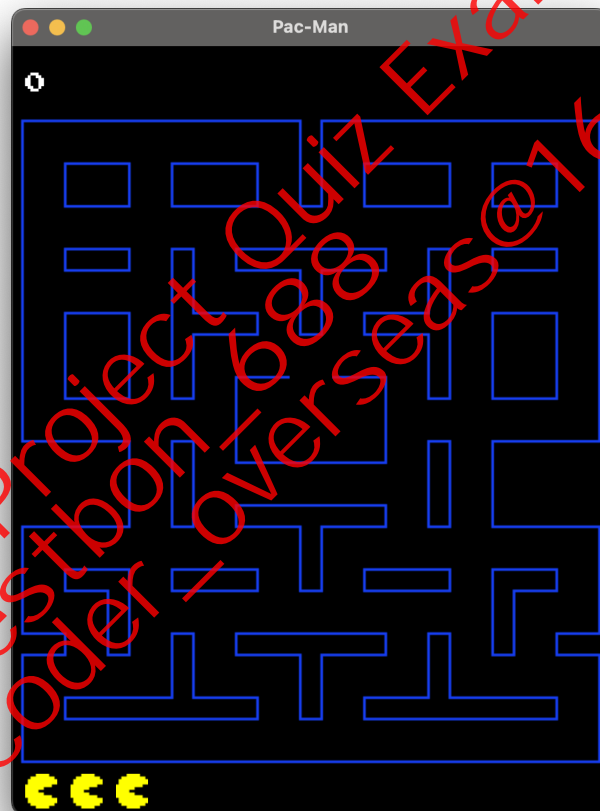
### Implementation Task (9 marks)

You will use the Java Programming Language to implement the Pac-Man game described below. It may be helpful for you to consider your design in Assignment 1 in your implementation.










Your Pac-Man game is expected to include the following features:

#### Map

The map designates where Pac-Man and the ghosts are allowed to move. It is laid out as a grid; entities can move along rows and columns where there is no wall (represented by a blue line).



Map layouts are stored in text files (e.g. `map.txt`), and the name of the map file can be found in the JSON configuration file under the `map` attribute. Map files store maps as multidimensional character arrays, where each character represents what is in that cell.

Character	Renderable	Sprite
0	Empty Cell	N/A
1	Horizontal Wall	
2	Vertical Wall	
3	Corner Wall (up + left)	
4	Corner Wall (up + right)	
5	Corner Wall (down + left)	
6	Corner Wall (down + right)	
7	Pellet	
p	Pac-Man	
g	Ghost	

Notes:

- You are guaranteed that maps are 36 rows tall and 28 columns wide with the first 3 rows and the last 2 rows being empty. Each grid space is 16x16 pixels.
- The screen size of the game is 448x576.
- You are guaranteed that all maps given are valid. Each map represents a maze that allows Pac-Man and the Ghost to move around.
- **All of the Renderables (Pac-Man, Ghosts, Walls, Pellets) included in the map must be constructed using the Factory Method pattern.**

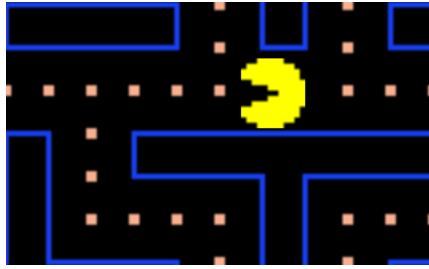
## Pac-Man

Players must be able to control Pac-Man with the arrow keys to move through the maze (up, down, left, right). **The handling of keyboard input from the Player and the corresponding movement of Pac-Man should be done using the Command Pattern.**

Pac-Man has five sprites that can be rendered: one with a closed mouth and one open-mouth sprite for each cardinal direction. Pac-Man's image should alternate between open-mouth and closed-mouth every eight frames, and the open-mouth sprite should relate to the direction in which he is moving. For example, if Pac-Man is moving upwards, Pac-Man's mouth should be pointing upwards.

Upon the start of the game, Pac-Man should move in the leftward direction automatically.

Unless colliding with a wall, Pac-Man is always moving. If colliding with a wall, Pac-Man will stay in the same position, but will continue to alternate his sprites between open-mouth and closed-mouth. The player can queue moves so that Pac-Man will turn accordingly when next possible. For example, if Pac-Man is in the following position, moving leftwards, and the player has pressed the following keys:



- down: Pac-Man would continue left and turn downwards at the next possible intersection.
- up: Pac-Man would continue left and turn upwards at the next possible intersection.
- right: Pac-Man would turn around immediately (Pac-Man can always turn around)
- left: Pac-Man would continue unchanged.

Note that if the player presses multiple keys before Pac-Man turns, the most recent key is used. For example, if the player presses down and then right before Pac-Man actually turns downwards, Pac-Man will turn rightwards only.

Pac-Man moves at a constant speed. His speed is specified for each level in the configuration file under the `pacmanSpeed` attribute.

The `numLives` attribute in the configuration file specifies the number of lives Pac-Man has. The remaining lives are represented on the screen with images of Pac-Mans facing right at the bottom.

**Note: you must use the Observer Pattern to display the number of lives Pac-Man has remaining on the screen.**



When Pac-Man runs out of lives, the game is over.

### Ghosts

Ghosts are the enemies of Pac-Man. Their goal is to prevent Pac-Man from collecting all the pellets on the map by hunting him down and hitting him. In assignment 2, there is only one type of ghost. The number of ghosts in the game is specified on the map, which will show all the starting positions of the ghosts.



Like Pac-Man, Ghosts can move horizontally and vertically on the map. They cannot pass through walls; however, they can pass through each other. Ghosts have two modes they can be in: **SCATTER** and **CHASE**. These modes determine the behaviour of the Ghosts when they reach an intersection.

Your application will alternate the ghosts between these two modes, with the first mode being SCATTER. The durations of these modes (in seconds) are specified in the JSON configuration file under the `modeLengths` attribute.

Ghosts move at a constant speed. Their speeds for each mode are specified for each level in the configuration file in the `ghostSpeed` attribute.

When a Ghost reaches an intersection (a point at which it can make a turn), it determines its target location and then moves in the direction that is closest to the target, based on Euclidean distance.

A Ghost's target location is determined by its current mode:

- If the Ghost is in SCATTER mode, the target location is a pre-assigned corner of the map. On creation, assign each ghost a randomly selected corner of the map.
- If the Ghost is in CHASE mode, the target location is the position of Pac-Man.

Note that, unless trapped, a ghost cannot turn around and move in the opposite direction to which it just moved. For example, if a ghost has moved upwards to an intersection, it cannot turn downwards, instead it must move in whichever of the other three cardinal directions is closest to the target location.

When a ghost collides with Pac-Man, Pac-Man loses a life, and all ghosts and Pac-Man return to their starting positions. When Pac-Man loses all of his lives, the game ends.

### **Pellet**

Pac-Man's goal is to collect every pellet on the map. Pac-Man collects a pellet by colliding with it. When this occurs, the pellet is removed from the map. Pellet entities do not move, nor do ghosts collect them.

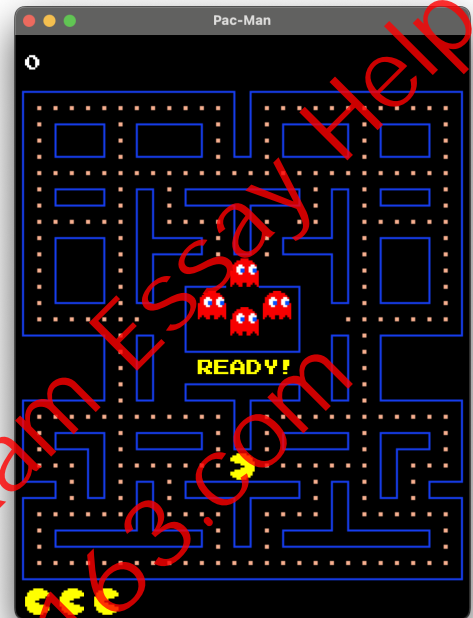
For each pellet Pac-Man consumes, he earns 100 points. His points are displayed on the top of the screen above the maze. **Note, you are required to display the player's score on the screen using the Observer Pattern.**

If Pac-Man is hit by a ghost, pellets are not reset - instead all pellets collected before the collision remain cleared from the map. Once Pac-Man collects all the pellets on the map, the game moves to the next level.

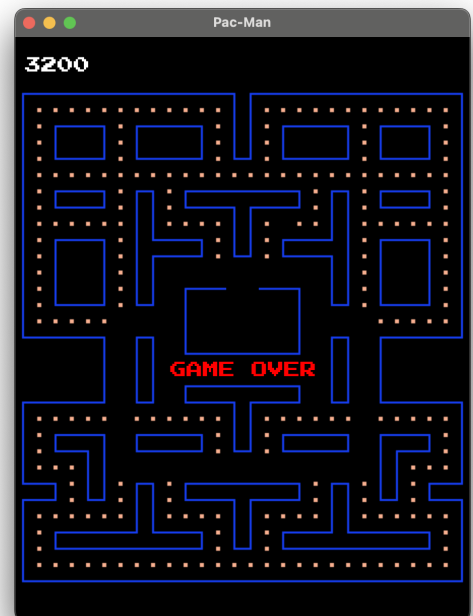
## Additional Requirements

When the game entities are in their starting positions, the screen is shown with the **READY!** text as shown for 100 frames before continuing. This applies to:

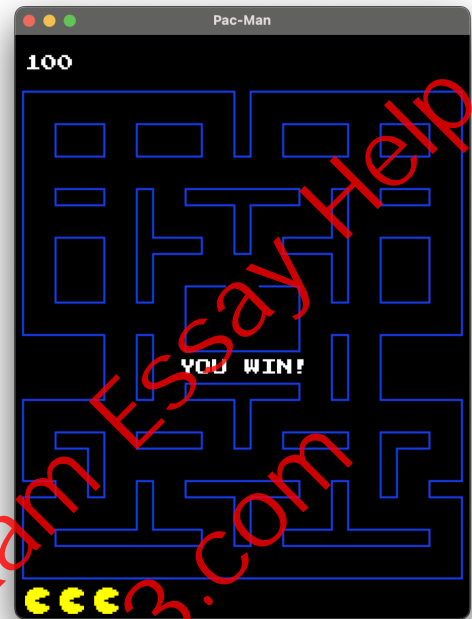
- Before the start of each level (including Game launch)
- After Pac-Man has lost a life (and still has lives left)



If Pac-Man runs out of lives, the **GAME OVER** screen is displayed for 5 seconds before the application ends. Note, that when Pac-Man has lost all of his lives, he and all the ghosts disappear from the screen.



When the Player has completed all the levels provided in the configuration file, the YOU WIN! screen is displayed for 5 seconds before the application ends. Note that when Pac-Man has finished all of the levels, he and all the ghosts disappear from the screen.



We have provided the font file in the resources folder of the provided code base. **You are required to implement these screens using the Observer Pattern.**

**IMPORTANT:** You can only implement the above features in your current assignment and should not include any other features. Mark deductions will be applied otherwise. You may want to refer to the general problem description in assignment 1 and consider whether your design could be extended to allow them or would need to be completely rewritten.



## Report Task (6 marks)

For your report, you are given a maximum of 1000 words with the following sections:

1. **Design Overview.** Include a UML class diagram of the entire system, including design patterns.
  - As your UML class diagram may be large and complex, aim to show only important attributes and methods in each class (i.e. not getters/setters). Additionally, only show significant relationships between modules.
  - If your UML diagram is too large, then you need to:
    - (a) include the whole UML diagram in this section; AND
    - (b) include enlarged versions of the key components in the Design Patterns section.
  - You must ensure your diagram is readable/understandable to the marker (e.g. avoid crossing lines where possible). If your marker cannot understand your diagram, zero marks will be given for the corresponding section.
  - Your marker will not download your PDF locally, so please ensure your diagram is readable in Canvas. Please note the maximum amount you can zoom in on Canvas is 200%. If your diagram is not clearly readable at 200% zoom, please use a higher screen quality when exporting it from your UML tool of choice, or export it as a PDF and combine it with your report PDF.
2. **Design Patterns.** Describe all design patterns used in your implementation with the following details:
  - The classes involved and their roles. You may include an enlarged class diagram of the participating classes here.
  - What this pattern does for your code in terms of **SOLID/GRASP principles**
  - What overall benefits this pattern provides (**be specific** to your code, not the pattern in general)
  - What drawbacks this pattern causes (**be specific** to your code, not the pattern in general)
  - For the Singleton pattern, explain why you chose this/these class(es) to be a Singleton.
3. **Assignment 1 reflection.** In this section, provide a discussion on how your design (i.e., class diagram) for assignment 1 helped or hindered your design in this assignment
4. **Change Justification.** Briefly rationalise the changes you have made to your assignment 1 design.
5. Any acknowledgement/reference required.

## Submission Details

You are required to submit all assessment items to their submission portals by the due date. This includes:

- **Report** (6 marks): Submit the report as a **SINGLE PDF** document on the [Canvas page for the report](#).
- **Code** (9 marks): Your code should be submitted as a **ZIP file containing only your src folder, build.gradle, and README** on the [EdStem page for the code](#).

### – Compile & Run

- \* **IMPORTANT:** The sample JSON configuration file and map.txt file must be included in your `src/main/resources` folder. Mark deductions will be applied if they are placed incorrectly.
- \* The README file has to cover any details you want your marker to know. In your README, you must include the following items:
  - how to run your code (e.g., any quirks to run your application)
  - which files and classes are involved in each design pattern implemented
  - anything else that you would like your marker to know
- \* We will execute your code by running `gradle clean build run` in the terminal with the environment configuration below:
  - Gradle 7.4.2
  - JDK 17
  - Unix-based System
- \* If your code fails to run using the instructions provided above, you will receive a **ZERO** mark for the coding portion of this assessment.

### – Style & Documentation

- \* Please follow the [Google Java Style Guide](#)
- \* Ensure names used are meaningful and the structure is clear and consistent
- \* Javadoc is not required, but please ensure you provide comments when needed.
- \* Put your pattern-related files in packages with pattern-related names, e.g. `observer`, `command`, `factory`