The University of Melbourne
School of Computing and Information Systems
SWEN20003 Object Oriented Software Development

ShadowPac

Project 2, Semester 1, 2023
Released: Friday, 21st April 2023 at 4:30pm AEST
Project 2A Due: Monday, 1st May 2023 at 4:30pm AEST
Project 2B Due: Friday, 19th May 2023 at 4:30pm AEST

Please read the complete specification before starting on the project, because there are important instructions through to the end!

Overview

In this project, you will create a maze game called *ShadowPac* in the Java programming language, continuing from your work in Project 1. We will provide a full working solution for Project 1; you **may** use all or part of it, provided you add a comment explaining where you found the code at the top of each file that uses the sample code.

This is an **individual project**. You may discuss it with other students, but all of the implementation must be your **own work**. By submitting the project you declare that you understand the University's policy on academic integrity and are aware of consequences of any infringement, including the use of **artificial intelligence**.

You may use any platform and tools you wish to develop the game, but we recommend using IntelliJ IDEA for Java development as this is what we will support in class.

Note: If you need an extension for the project, please complete the Extension form in the **Projects** module on Canvas. Make sure you explain your situation with some supporting documentation such as a medical certificate, academic adjustment plan, wedding invitation, etc. You will receive an email saying if the extension was approved or if we need more information.

If you submit late (**either** with or without an extension), please complete the Late form in the **Projects** module on Canvas. For both forms, you need to be logged in using your **university** account. Please **do not** email any of the teaching team regarding extensions or late submissions. All of this is explained again in more detail at the end of this specification.

There are two parts to this project, with different submission dates. The first task, **Project 2A**, requires that you produce a class design demonstrating how you plan to implement the game. This should be submitted in the form of a UML diagram showing all the classes you plan to implement, the relationships (e.g. inheritance and associations) between them, and their attributes, as well as their primary public methods. You **do not** need to show constructors, getters/setters, dependency, composition or aggregation relationships. If you so choose, you may show the relationship on a separate page to the class members in the interest of neatness, but you must use correct UML notation. Please submit as a **PDF file** only on Canvas.

The second task, **Project 2B**, is to complete the implementation of the game as described in the rest of this specification. You **do not** need to strictly follow your class design from Project 2A; you will likely find ways to improve the design as you implement it. Submission will be via GitLab and you must make **at least 5 commits** throughout your project.

Game Overview

*"The **player** controls **PacMan**, the infamous yellow main character, through an enclosed maze. To win, move around the **walls**, eat all the **dots** in the maze and avoid the **ghosts**! In the second level, the player has the ability to attack the ghosts when they eat a **pellet** and the ability to gain bonus points when they eat a **cherry**. "*

The game features two levels : *Level 0* and *Level 1*, both of which are mazes. In Level 0, the player will be able to control PacMan who has to move around the walls, eat the dots and avoid the *red ghosts*, who are stationary. If the player collides with a ghost, the player will lose a life. The player has 3 lives in total. To finish the level, the player needs to eat (collide) with all the dots. If the player loses all 3 lives, the game ends. You have already implemented Level 0 in Project 1 (the only change required is to the winning message screen which is explained later).

When the player finishes Level 0, Level 1 starts. To win the level and the game, the player must reach a score of 800. However, the player has to deal with 4 types of ghosts (*red, blue, green & pink*). The ghosts will be moving in different directions (as explained later). If the player collides with the *pellet*, the game goes into a *frenzy mode*, where the player can gain extra points when colliding with the ghosts and not lose any lives. The player can also gain more points if they collide with a *cherry*. Outside of the frenzy mode, if the player collides with a ghost, they will lose a life and losing all 3 lives means the game will end.

An Important Note

Before you attempt the project or ask any questions about it on the discussion forum, it is crucial that you read through this entire document thoroughly and carefully. We've covered every detail below as best we can without making the document longer than it needs to be. Thus, if there is any detail about the game you feel was unclear, try referring back to this project spec first, as it can be easy to miss some things in a document of this size. And if your question is more to do on **how** a feature should be implemented, first ask yourself: *'How can I implement this in a way that both **satisfies the description** given, and helps make the game **easy and fun to play**?'* More often than not, the answer you come up with will be the answer we would give you!

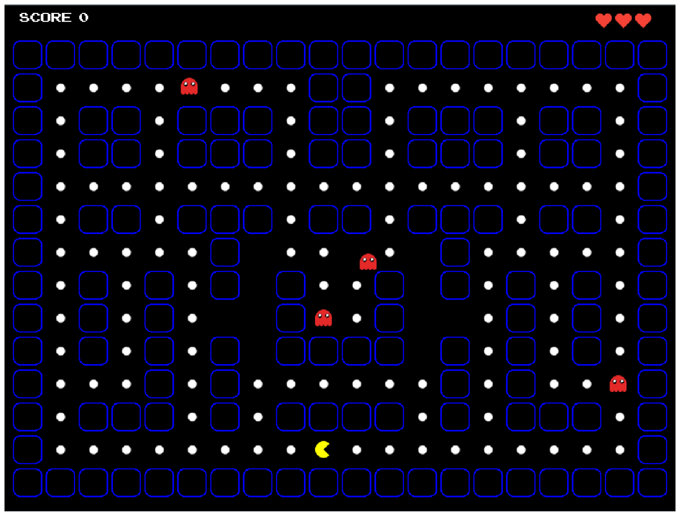


Figure 1: Completed Level 0 Screenshot

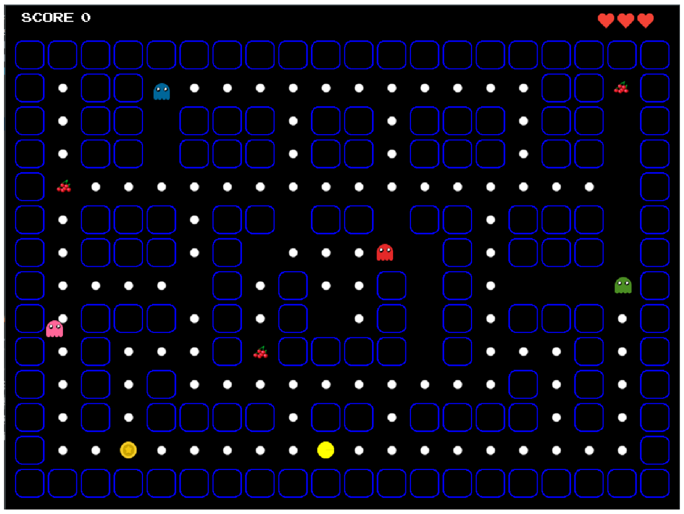


Figure 2: Completed Level 1 Screenshot

Note : the actual positions of the entities in the levels we provide you may not be the same as in these screenshots.

The Game Engine

The **B**asic **A**cademic **G**ame **E**ngine **L**ibrary (Bagel) is a game engine that you will use to develop your game. You can find the documentation for Bagel here.

Coordinates

Every coordinate on the screen is described by an (x, y) pair. $(0, 0)$ represents the top-left of the screen, and coordinates increase towards the bottom-right. Each of these coordinates is called a *pixel*. The Bagel `Point` class encapsulates this.

Frames

Bagel will refresh the program's logic at the same refresh rate as your monitor. Each time, the screen will be cleared to a blank state and all of the graphics are drawn again. Each of these steps is called a **frame**. Every time a frame is to be rendered, the `update()` method in `ShadowPac` is called. It is in this method that you are expected to update the state of the game.

Your code will be marked on **60Hz screens**. The refresh rate is typically `60` times per second (Hz) but newer devices might have a higher rate. In this case, when your game is running, it may look different to the demo videos as the constant values in this specification have been chosen for a refresh rate of 60Hz. For your convenience, when writing and testing your code, you **may** either change these values to make your game playable or lower your monitor's refresh rate to 60Hz. If you do change the values, **remember** to change them back to the original specification values before submitting, as your code will be **marked on 60Hz screens.**

This specification requires certain gameplay to only happen for a given number of frames. These numbers are **not fixed** and we'll allow some room for error - as long as the game still plays roughly as intended!

Collisions

It is sometimes useful to be able to tell when two images are *overlapping*. This is called **collision detection** and can get quite complex. For this game, you can assume images are rectangles and that the player meeting any other entity is a collision. Bagel contains the `Rectangle` class to help you.

The Levels

Our game will have two levels, each with messages that would be rendered at the start and end of the level.

Window and Background

The background (`background0.png`) should be rendered on the screen to completely fill up your window throughout the game (for the instructions, Level 0 and Level 1). The default window size should be `1024 * 768` pixels. The background has already been implemented for you in the skeleton package.

Level Messages

All messages should be rendered with the font provided in `res` folder (`FSO8BITR.ttf`), in size `64` (unless otherwise specified). All messages should be centered both horizontally and vertically (unless otherwise specified).

Hint: The `drawString()` method in the Font class uses the given coordinates as the bottom left of the message. So to center the message, you will need to calculate the coordinates using the `Window.getWidth(), Window.getHeight() and Font.getWidth()` methods, and also the font size.

Level Start

When the game is run, Level 0 should start with a title message that reads `SHADOW PAC` should be rendered in the font provided. The bottom left corner of this message should be located at `(260, 250)`.

Additionally, an instruction message consisting of 2 lines:

```
PRESS SPACE TO START
USE ARROW KEYS TO MOVE
```

should be rendered **below** the title message, in the font provided, in size `24`. The bottom left of the first line in the message should be calculated as follows: the x-coordinate should be increased by `60` pixels and the y-coordinate should be increased by `190` pixels.

There must be **adequate spacing** between the 2 lines to ensure readability (you can decide on the value of this spacing yourself, as long as it's not small enough that the text overlaps or too big that it doesn't fit within the screen). You can align the lines as you wish.

At the start of Level 1, the following instruction message with these 3 lines should be shown:

```
PRESS SPACE TO START
USE ARROW KEYS TO MOVE
EAT THE PELLET TO ATTACK
```

This message should be rendered in the font provided in size `40` and the bottom left of the first line in the message should be located at `(200, 350)`. The spacing and alignment of the lines is the same as described above.

Each level begins once the start key (space bar) is pressed. To help when testing your game, you can allow the user to skip ahead to the Level 1 start screen by pressing the key 'W' (this is not assessed but will help you when coding, especially when working on Level 1).

World File

All the entities will be defined in a **world file**, describing the types and their positions in the window. The world file for Level 0 is `level0.csv` and Level 1 is `level1.csv`. A world file is a comma-separated value (CSV) file with rows in the following format:

```
Type, x-coordinate, y-coordinate
```

An example of a world file:

```
Player,474,662
GhostRed,374,362
Cherry,374,512
Pellet,174,662
Wall,12,50
```

You must actually load both files—copying and pasting the data, for example, is not allowed. **Note:** You can assume that the player is always the first entry in both files, the Level 0 world file will have a maximum of 271 entries and the Level 1 world file will have a maximum of 266 entries.

Win Conditions

For Level 0, once the player eats all the dots, this is the end of the level. A winning message that reads `LEVEL COMPLETE!` should be rendered as described earlier in the *Level Messages* section. This message should be rendered for **300 frames** before displaying the start screen for Level 1.

In Level 1, once the player reaches a score of **800**, this is considered a win (note that the player does not have to eat all the dots to complete the level). A winning message that reads `WELL DONE!` should be rendered as described in the *Level Messages* section.

Lose Conditions

On either level, while there is no win, the game will continue running until it ends. As described earlier, the game can only end if the player's number of lives reduce to 0. A message of `GAME OVER!` should be rendered as described in the *Level Messages* section.

If the player terminates the game window at any point (by pressing the Escape key or by clicking the Exit button), the window will simply close and no message will be shown.

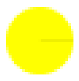
The Game Entities

All game entities have an associated image (or multiple!) and a starting location `(x, y)` on the map which are defined in the CSV files provided to you. Remember that you can assume the provided images are rectangles and make use of the `Rectangle` class in Bagel; the provided `(x, y)` coordinates for a given entity should be the **top left** of each image.

Hint: `Image` has the `drawFromTopLeft` method and `Rectangle` has the `intersects` method for you to use, refer to the Bagel documentation for more info.

The Player

In our game, the player is represented by PacMan. The player is controlled by the four arrow keys and can move continuously in one of four directions (left, right, up, down) by **3 pixels per frame** whenever an arrow key is **held down**.


(a) pac.png

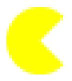
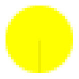

(b) pacOpen.png

Figure 3: The player's images

The player is represented by the images shown above. Every **15 frames**, the image rendered should switch between the two (i.e. it should look like the player opening and closing its mouth). Initially, the player will start by facing right, as shown above. Based on the direction the player is moving, the image being rendered needs to be **rotated**. For example, if the player is moving down, the images need to be rotated by either 90 degrees clockwise or 270 degrees anti-clockwise, as shown below.


(a) pac.png after rotation

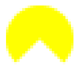

(b) pacOpen.png after rotation

Figure 4: The player's images rotated when moving downwards

Hint: The `drawFromTopLeft` method has an overloaded method that takes a `DrawOptions` object as a parameter. `DrawOption` objects have a `setRotation` method, that allow the rotation to be set in radians.

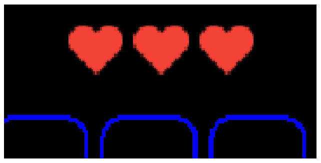


Figure 5: Player's lives

The player has **3 lives**. If the player collides with a ghost (of any colour), the player will lose a life, **both** ghost and player's positions will be reset to the **starting position**. If the player loses all 3 lives, the game ends. Each life is represented by `heart.png`. The hearts are rendered in the top right corner of the screen. The first heart should be rendered with the top left coordinate at `(900, 10)`. The x-coordinate of each heart after should be increased by `30`, as shown on the left.

The player has an associated **score**. When the player collides with a dot, the player's score increases by `10` (the points value of the dot). If the player collides with a cherry, the player's score increases by `20` (the points value of the cherry). The score is rendered in the top left corner of the screen in the format of `"SCORE k"` where `k` is the current score. The bottom left corner of this message should be located at `(25, 25)` and the font size should be 20.

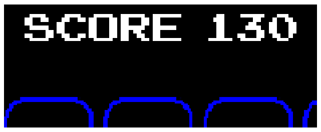

SCORE 130

Figure 6: Player's score rendering

When the player collides with a **pellet**, the game goes into **frenzy mode**. In this mode, if the player collides with a ghost, they will not lose a life and their score increases by `30` points for each ghost collision. The player's speed also increases by 1 to **4 pixels per frame**. The frenzy mode lasts for **1000 frames**. Once this ends, the player's speed reduces by 1 to its original speed and the collision behaviour reverts to normal.

Enemies

The enemies are the four ghosts that can attack the player. Note that enemies are allowed to overlap with each other during movement.

Red Ghost

Red ghosts feature in both levels and are represented by `ghostRed.png`. In Level 0, they are **stationary** (like in Project 1) but in Level 1, they move in the **horizontal** direction at a a speed of **1** pixel per frame. They will start initially by moving **right**.

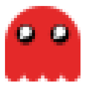


Figure 7: Red Ghost

A ghost cannot overlap or move though a wall. When there is a collision with a wall, the red ghost's movement **reverses** in the horizontal direction (for e.g: if the ghost hits a wall while moving right, the ghost will move left after the collision). When there is a collision with the player, the player and the ghost will reset to the starting position. If a ghost collides with a dot or a cherry, they will simply move through (the order in which you render them is up to you).

Blue Ghost

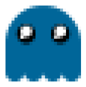


Figure 8: Blue Ghost

Blue ghosts feature only in Level 1 and are represented by `ghostBlue.png`. They move in the **vertical** direction at a a speed of **2** pixels per frame. They will start initially by moving **downwards**.

A ghost cannot overlap or move though a wall. When there is a collision with a wall, the blue ghost's movement **reverses** in the vertical direction (for e.g: if the ghost hits a wall while moving down, the ghost will move up after the collision). When there is a collision with the player, the player and the ghost will reset to the starting position. If a ghost collides with a dot or a cherry, they will simply move through (the order in which you render them is up to you).

Green Ghost

Green ghosts are only in Level 1 and are shown by `ghostGreen.png`. They can move in one of two directions (horizontal or vertical), **randomly** se-

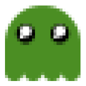


Figure 9: Green Ghost

lected at creation at a speed of **4** pixels per frame. The ghost will start by initially increasing in the selected direction (i.e, if the vertical direction is chosen, the ghost will move down and if the horizontal direction is chosen, the ghost will move right).

A ghost cannot overlap or move though a wall. When there is a collision with a wall, the green ghost's movement **reverses** in the selected direction (for e.g: if a ghost moving in the vertical direction hits a wall while moving up, the ghost will move down after the collision). When there is a collision with the player, the player and the ghost will reset to the starting position. If a ghost collides with a dot or a cherry, they will simply move through (the order in which you render them is up to you).

Pink Ghost

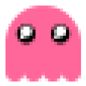


Figure 10: Pink Ghost

Pink ghosts are only in Level 1 and are rendered by `ghostPink.png`. They can move in one of four directions (left, right, up and down), **randomly** selected at creation at a speed of **3** pixels per frame.

A ghost cannot overlap or move though a wall. When there is a collision with a wall, the pink ghost's direction will be **randomly** chosen again from the four possible directions. When there is a collision with the player, the player and the ghost will reset to the starting position. If a ghost collides with a dot or a cherry, they will simply move through (the order in which you render them is up to you).

The Ghosts during Frenzy Mode

During frenzy mode, all of the ghosts images will be changed to `ghostFrenzy.png` as shown on the right. Their speeds will correspondingly decrease by **0.5 pixels per frame**. During this mode, if a ghost collides with the player, the ghost will **disappear** from the screen. All other behaviour remains the same during this mode.

Figure 11: A Ghost during frenzy mode

Once the frenzy mode ends, the ghosts that disappeared will reappear at their **starting** positions (i.e. the original positions in the csv file). Any ghosts that didn't disappear, will continue to move as normal. All of the ghosts behaviour returns to **normal** including the speed, the images and the collision behaviour.

Stationary Entities

These are entities placed throughout the level that do not move, at locations specified by the level CSV file (*the Level 0 Red Ghost is also stationary but is not included here as it has been explained earlier*). These may apply some effect on the moving entities that collide with them, and may need to disappear at some point (i.e. the game should stop rendering and updating them).

Wall

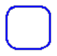


Figure 12: Wall

A wall is a stationary object, shown by `wall.png`. The player **shouldn't** be able to overlap with or move through the walls, i.e. the player must move the player around any areas on the level where walls are being rendered.

Dot

A dot is a stationary object, shown by `dot.png`, with a points value of `10`. When the player collides with a dot, the player's score increases by the dot's point value and the dot disappears from the screen.

Figure 13: Dot

Cherry

Figure 14: Cherry

A cherry is a stationary object, shown by `cherry.png`, with a points value of `20`. When the player collides with a cherry, the player's score increases by the cherry's point value and the cherry disappears from the screen.

Pellet

A pellet is a stationary object, shown by `pellet.png`. When the player collides with it, the game goes into frenzy mode and the pellet disappears from the screen. If a ghost collides with the pellet, nothing happens and the ghost is able to pass through.

Figure 15: Pellet

Entities Summary

Entity	**Image filenames**	**Movement speed**	**Health lives**	**Points**	**Movement direction**	**Collision effect on the player**
Player	pac.png OR pacOpen.png (both need to be rotated in movement direction)	3 OR 4 (during frenzy)	3	-	Based on user input	-
Red Ghost	redGhost.png OR ghostFrenzy.png (during frenzy)	1 OR 0.5 (during frenzy)	-	30 (during frenzy)	Horizontal (initially moves right)	Player loses life OR player score increases by points value (during frenzy)
Blue Ghost	blueGhost.png OR ghostFrenzy.png (during frenzy)	2 OR 1.5 (during frenzy)	-	30 (during frenzy)	Vertical (initially moves down)	Player loses life OR player score increases by points value (during frenzy)
Green Ghost	greenGhost.png OR ghostFrenzy.png (during frenzy)	4 OR 3.5 (during frenzy)	-	30 (during frenzy)	Horizontal OR vertical (randomly chosen, initially increases in chosen direction)	Player loses life OR player score increases by points value (during frenzy)
Pink Ghost	pinkGhost.png OR ghostFrenzy.png (during frenzy)	3 OR 2.5 (during frenzy)	-	30 (during frenzy)	Up, down, left or right (randomly chosen)	Player loses life OR player score increases by points value (during frenzy)

Wall	wall.png	-	-	-	-	Prevents the player from moving through it
Dot	dot.png	-	-	10	-	Increases player score by points value
Cherry	cherry.png	-	-	20	-	Increases player score by points value
Pellet	pellet.png	-	-	-	-	Starts frenzy mode

Your Code

You must submit a class called `ShadowPac` that contains a `main` method that runs the game as prescribed above. You may choose to create as many additional classes as you see fit, keeping in mind the principles of object oriented design discussed so far in the subject. You will be assessed based on your code running correctly, as well as the effective use of Java concepts. As always in software engineering, appropriate comments and variables/method/class names are important.

Implementation Checklist

To get you started, here is a checklist of the game features, with a suggested order for implementing them (in addition to the features in Project 1):

- Implement the end of Level 0 screen.
- Implement the Level 1 start screen.
- Render the Level 1 background and place the game entities read from the CSV file.
- Implement the player's behaviour/logic.
- Implement each ghost's behaviour.
- Implement the cherry's behaviour.
- Implement the pellet and frenzy mode behaviour.
- Implement win detection and end of Level 1 screen.
- Implement lose detection for Level 1.

Supplied Package and Getting Started

You will be given a package called `project-2-skeleton.zip` that contains the following: (1) Skeleton code for the `ShadowPac` class to help you get started, stored in the `src` folder. (2) All graphics and fonts that you need to build the game, stored in the `res` folder. (3). The `pom.xml` file required for Maven. You should use this template exactly how you did for Project 1, that is:

1. Unzip it.
2. Move the **content** of the unzipped folder to the local copy of your [username]-project-2] repository.
3. Push to Gitlab.
4. Check that your push to Gitlab was successful and to the correct place.
5. Launch the template from IntelliJ and begin coding.
6. Commit and push your code regularly.

Customisation (optional)

We want to encourage creativity with this project. We have tried to outline every aspect of the game design here, but if you wish, you may customise any part of the game, including the graphics, types of actors, behaviour of actors, etc (for example, an easy extension could be to introduce a new level with different items). You can also add entirely new features. For your customisation, you **may** use additional libraries (other than Bagel and the Java standard library).

However, to be eligible for full marks, you **must** implement all of the features in the above implementation checklist. Please submit the version **without** your customisation to [username]-project-2 repository, and save your customised version locally or push it to a new branch on your Project 2 repository.

For those of you with far too much time on your hands, we will hold a competition for the best game extension or modification, judged by the lecturers and tutors. The winning three will have their games shown at the final lecture, and there will be a prize for our favourite. Past modifications have included drastically increasing the scope of the game, adding jokes and adding polish to the game, and even introducing networked gameplay.

If you would like to enter the competition, please email the head tutor, Tharun Dharmawickrema at `dharmawickre@unimelb.edu.au` with your username, a short description of the modifications you came up with and your game (either a link to the other branch of your repository or a .zip file). You can email Tharun with your completed customised game anytime before **Week 12**. Note that customisation does **not** add bonus marks to your project, this is completely for fun. We can't wait to see what you come up with!

Submission and Marking

Project 2A

Please submit a **.pdf** file of your UML diagram for Project 2A via the Project 2A tab in the Assignments section on Canvas.

Project 2B - Technical requirements

- The program must be written in the Java programming language.
- Comments and class names must be in English **only**.
- The program must not depend upon any libraries other than the Java standard library and the Bagel library (as well as Bagel's dependencies).
- The program must compile fully without errors.
- For full marks, **every** public attribute, method and class must have a short, descriptive Javadoc comment (which will be covered later in the semester).

Submission will take place through GitLab. You are to submit to your `<username>-project-2` repository. At the **bare minimum** you are expected to follow the structure below. You **can** create more files/directories in your repository if you want.

```
username-project-2
├─ res
│  └─ resources used for project 2
└─ src
   ├─ ShadowPac.java
   └─ other Java files
```

On 19th May 2023 at 4:30pm, your latest commit will automatically be harvested from GitLab.

Commits

You are free to push to your repository post-deadline, but only the latest commit on or before 19th May 2023 4:30pm will be marked. You **must** make at least 5 commits throughout the development of the project, and they must have meaningful messages (commit messages must match the code in the commit). If commits are anomalous (e.g. commit message does not match the code, commits with a large amount of code within two commits which are not far apart in time) you risk penalization.

Examples of **good, meaningful** commit messages:

- implemented movement logic
- fix the player's collision behaviour

- refactored code for cleaner design

Examples of **bad, unhelpful** commit messages:

- fesjakhbdjl
- yeah easy finished the player
- fixed thingzZZZ

Good Coding Style

Good coding style is a contentious issue; however, we will be marking your code based on the following criteria:

- You should *not* go back and comment your code after the fact. You should be commenting as you go. *(Yes, we can tell.)*
- You should be taking care to ensure proper use of visibility modifiers. Unless you have a very good reason for it, all instance variables should be private. (Constants are allowed to be public or protected).
- Any constant should be defined as a final variable. Don't use magic numbers!
- Think about whether your code is written to be easily extensible via appropriate use of classes.
- Make sure each class makes sense as a cohesive whole. A class should have a single well-defined purpose, and should contain all the data it needs to fulfil this purpose.

Extensions and late submissions

If you need an **extension** for the project, please complete Extension form in the **Projects** module on Canvas. Make sure you explain your situation with some supporting documentation such as a medical certificate, academic adjustment plan, wedding invitation, etc. You will receive an email saying if the extension was approved or if we need more information.

The project is due at **4:30pm sharp** on Monday 1st May 2023 (Project 2A) and on Friday 19th May 2023 (Project 2B). Any submissions received past this time (from 4:30pm onwards) will be considered late unless an extension has been granted. There will be no exceptions. There is a penalty of 1 mark for a late project, plus an additional 1 mark per 24 hours. If you submit **late** (*either* with or without an extension), please complete the Late form in the **Projects** module on Canvas. For both forms, you need to be logged in using your **university** account. Please **do not** email any of the teaching team regarding extensions or late submissions (as you will be redirected to the online forms).

Marks

Project 2 is worth **22** marks out of the total 100 for the subject. You are **not required** to use any particular features of Java. For example, you may decide not to use any interfaces or generic classes.

You will be marked based on the **effective and appropriate** use of the various object-oriented principles and tools you have learnt throughout the subject.

- Project 2A is worth **8 marks**.
  - Correct UML notation for methods: **2 marks**
  - Correct UML notation for attributes: **2 marks**
  - Correct UML notation for associations: **2 marks**
  - Good breakdown into classes: **1 mark**
  - Appropriate use of inheritance, interfaces and abstract classes/methods: **1 mark**
- Project 2B (feature implementation) is worth **10 marks**.
  - Correct implementation of start screen and entity creation: **0.5 marks**
  - Correct implementation of the player's behaviour (including collisions and score): **1 mark**
  - Correct implementation of each ghost's behaviour (including images, movement and collisions): **4 marks**
  - Correct implementation of stationary items' behaviour: **1 mark**
  - Correct implementation of frenzy mode behaviour for player and ghosts: **2 marks**
  - Correct implementation of level transition: **0.5 marks**
  - Correct implementation of win and end of game logic: **1 mark**
- Coding Style is worth **4 marks**.
  - Delegation: breaking the code down into appropriate classes: **0.5 marks**
  - Use of methods: avoiding repeated code and overly complex methods: **0.5 marks**
  - Cohesion: classes are complete units that contain all their data: **0.5 marks**
  - Coupling: interactions between classes are not overly complex: **0.5 marks**
  - General code style: visibility modifiers, magic numbers, commenting etc.: **1 mark**
  - Use of Javadoc documentation: **1 mark**