Exercise 1 - Design Patterns

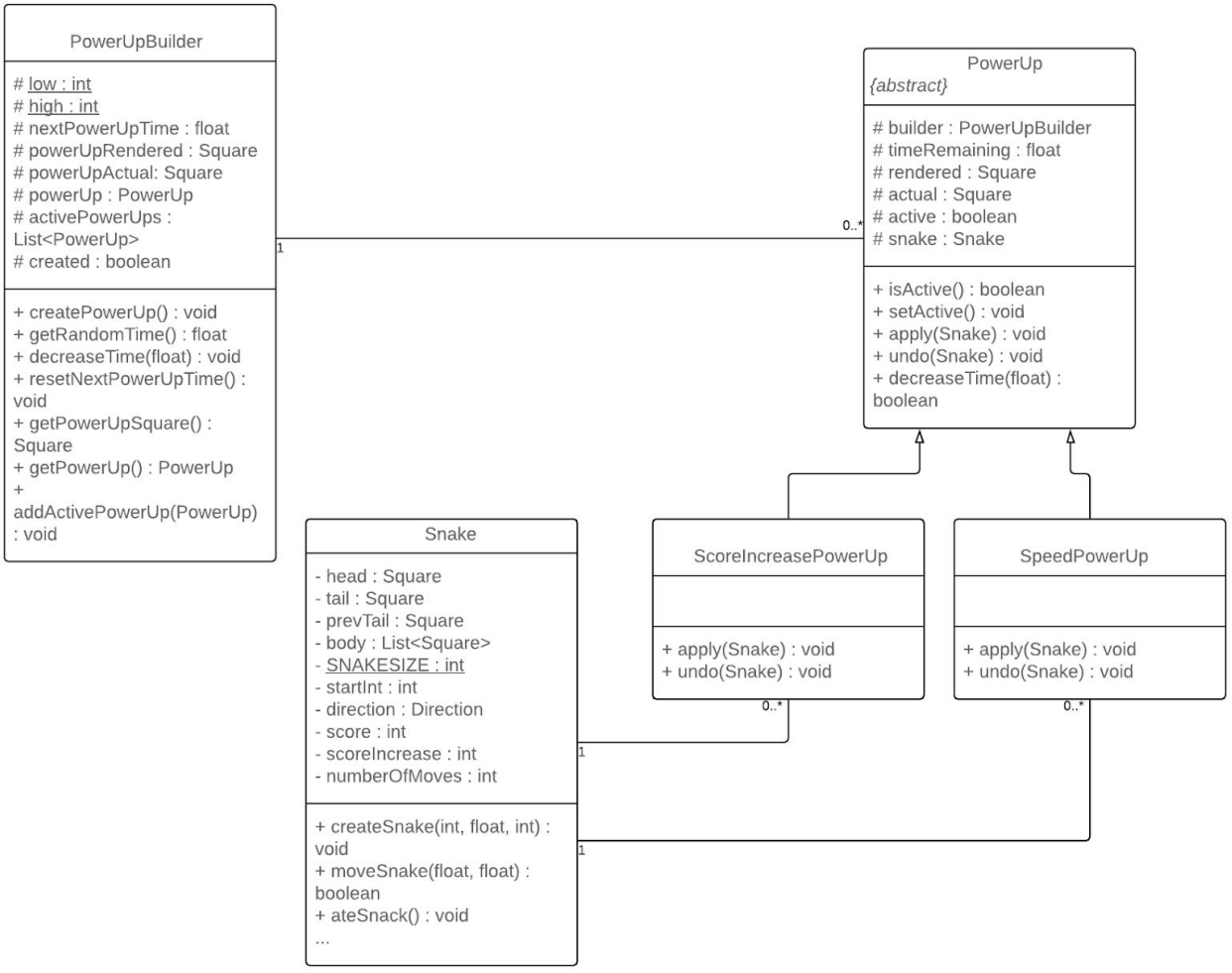
Design Pattern 1 - Command

1 -

We use the command pattern to avoid long if-statements and it lets us encapsulate the actions using an object making it easy for the invoker (in our case the PowerUpBuilder) to execute the given commands.

The pattern is implemented by having a PowerUpBuilder class which creates the PowerUp and keeps track of all of the currently active PowerUps. If the PowerUp is then picked up, the apply method is invoked. The apply method has as parameter, the target snake, on which the action is operated. After a certain amount of time has passed (based on the amount of time which has passed since the previous frame), the PowerUpBuilder makes sure that the command is undone and the effects are reversed.

2 -



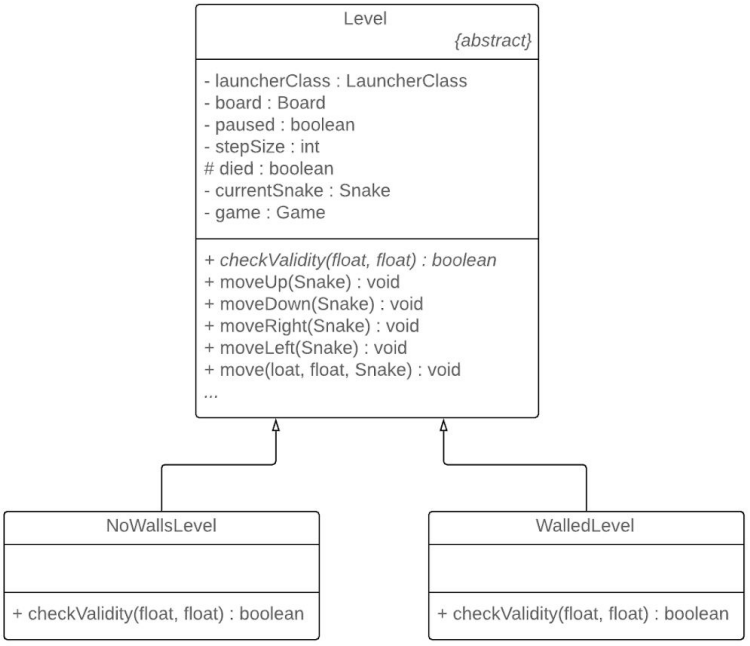
Design Pattern 2 - Template

1 -

We implement the Template pattern so that we do not have to implement overlapping functionality. If we were to not use this pattern then we would have a lot of code duplication which would result in a lower maintainability.

We are using this design pattern in the implementation of the level. We have a main abstract level class which is responsible for implementing the invariant parts of the level class (e.g the move method). However, the WalledLevel and the NoWallsLevel both have different validity checkers for the move method. By subclassing the level, we prevent code duplication and are able to only redefine the varying parts of the algorithm.

2 -



Exercise 2 - Software Architecture

Software architecture - Layered Architecture

We implement the layered architecture to ensure that there is a clear separation of the graphical user interface and the logic of the game. To ensure this separation, we make use of different layers with each its own responsibility.

There are three layers:

1 -

The logic layer, this is the core layer of the game. This component is responsible for handling the main logic of the game and, for example, moving the snake of the player. The main API for interaction with the GUI goes through the game class, which is aware of all of the different components of the game logic. The game package has the responsibility of keeping track of whether the game is over and the player has died and the package is responsible for creating new snacks and/or powerups. While the game class has methods for moving the player, the actual moving is done in the level class which is contained in the game class. The level package is aware of the board and while it does not keep track of the player, it is responsible for the validation of moves which is dependent on which type of level is instantiated. The level is also responsible for checking whether or not the player interacts with a snack and/or power up. Finally, the move method is then called on the player and the snake class takes care of moving the body.

2 -

The Graphical User Interface layers is the core layer of visuals. This component is responsible for presenting what is on the screen at a given time and handling the user input that is given to that screen. This component consists of multiple screen classes, each with the responsibility for one particular part of the GUI. In general there are no method calls from one screen class to a method of another screen class. There are certain screens which require interaction with the interface of the logic of the game while other components only require interaction with the storage layer. The game screen requires the logic layer to update the body of the snake and do all the necessary move validation, after which it renders the updated snake. However, in for example the login screen, an interaction is necessary with the Dao to ensure that user credentials are correct.

3 -

The final layer is the storage layer which is responsible for interacting with the database and keeping track of textures, music, etc. The Dao is responsible for maintaining a connection to the database which keeps track of the user information. It provides an API for several interactions with the database. There is however one more component in the storage layer which is responsible for keeping track of the music/sounds played and the textures of the snack/powerups.

Why we chose to implement this architecture

We chose to implement this architecture over other architectures because we wanted to ensure that we could test the game logic separately from the GUI. This also means that if we change a component of the game logic, the implementation of the GUI did not need to be changed if the API remained the same which is desirable for maintainability.

We chose this architecture over, for example, pipe and filter because we wanted to make sure that the response time was low. Furthermore, by keeping the amount of layers low we ensure that the performance can stay on a high level while in other architectures this might introduce a higher complexity into the system.