System design document for and I OOPP

Jacob Bengtsson, Anton Ekström, Elin Nilsson
Amanda Papacosta, Arvid Svedberg

October 24, 2021

1 Introduction

This document describes the tower defense game. While playing the game, the player has to place towers in order to defend the goal. Towers in this case are stationary characters which serve to stop the approaching enemies. The game is divided into lanes which are divided into cells. The enemies walk along the lanes and the player can place tower in the cells. Each tower has a different attack and the enemies have their own weakness, making the choice of tower a strategic one.

1.1 Definitions, acronyms, and abbreviations

Tower defense is a subgenre of video games within the more encompassing strategy genre. In general, the player has to place towers to defend something and it's up to the player to make sure that the correct towers are placed to ensure a strong enough defense to protect the objective at hand.

Towers are the characters which are placed in the game world which by different means defeat incoming enemies to protect the objective.

This tower defense game is built upon the concept of **lanes** and **cells**. Lanes are horizontal paths in which the enemies can move. These lanes are then divided into cells to create a grid and within each of the cells a tower can be placed. The tower can then attack all enemies which are approaching in the same lane as the tower.

The attacking enemies are sent out in **waves**. A wave is a collection of enemies sent out to different lanes at different intervals. Once all enemies in a wave a new wave can be started. Waves further into a session will have more enemies as well as tougher enemies. This means the player has to improve their defense over time to accommodate the increasingly dangerous competition.

2 System architecture

The application is built using an MVC architecture, in which the view and model listen to a clock that triggers updates on regular intervals. The view receives the model in order to render the game. The view and renderer are independent of each other. Both should also be easily interchangeable, because they are only interacted with through their respective interfaces. User interaction and manipulation of the model is done through controllers which are applied when the game is initialized.

2.1 Application entrypoint

The entrypoint of the application is through the Game class. Its role is to initialize and compose the various views, controllers and systems with the model. The instance of Game is updated by a clock which in turn updates all the systems and views that are stored within the game.

2.2 Program flow

Initialization and setup occurs in the MarioGame class. It configures and connects different parts of the program such as views and controllers. The MarioGame instance is then hooked up to a clock which updates the model and view repeatedly. The only components which are invoked repeatedly by the clock are the views and systems. The primary program loop is thusly the update method on MarioGame which updates each system. Systems apply the logic of the game to the model. Some systems and services produce events which are listened to by other services or controllers. Examples of such events are when the player uses their mouse or when an enemy dies. These events are therefore also actions which affect the state of the model.

2.3 Model

The model stores the game state and provides a facade for interacting with the game. Though, most of the business logic is not part of the actual Model class and is instead separated into what are called systems. A system is an object that implements the functional interface System, which has an update method that performs some action on the model. These are invoked repeatedly by the games clock.

The model does not know about how the game should be displayed. Therefore, it does not have any reference to images and sprites or pixel coordinates. The model package has classes which represent our domain and how it is structured. There are classes to represent enemies, towers and the playing-field on which these entities are located.

2.4 View

The purpose of the view is to define how to render the game. It does not mutate the model in any way. The view is responsible for determining where and how things should look. The actual rendering technique is handled by the renderer.
A view is an object that implements the functional interface View, which takes a renderer and the model as arguments.

2.5 Graphics

The graphics layer abstracts any implementation details of rendering. Currently, the renderer is implemented in JavaFX. Though, it could rather easily be implemented with another graphics solution such as Java Swing.

2.6 Controller

Controllers receive user input from the view and in turn interacts with the model. Controllers may need a reference to some parts of the view, for example to determine if the user clicks on a certain element. A controller then forwards this intent by calling methods on the model.

2.7 Services

Services provide functionality for the model. These are essentially classes which don't fit into the other categories (systems, views or controllers).

3 System design

The MVC pattern is implemented by and consists of the model, view and controller packages. The model consists of domain objects, systems and services. Domain objects are interacted upon by systems which apply some game logic through the update method. Services provide additional functionality for the model. The view observes the model for changes. The controller observes both and translates actions on the view into operations on the model. The view is also dependent on a graphics package in order the carry out its rendering.

The systems top-level packages are divided into the following:

- **model**
  Describes the domain and business logic.

- **view**
  Presents the game to the player.
- **controller**
  Takes user interactions and performs actions in the game.
- **game**
  Combines components of the program.
- **common**
  Provides general utilities.
- **main**
  Starts the application.

3.1 Factory Pattern

In the program, factories play the role of creating a more concrete implementation of objects based on certain interfaces, for example the TransformFactory class. The classes EnemyFactory and TowerFactory also work as factories for creating enemies and towers since they hold the basic methods for creating said objects.

3.2 Model View Controller

Furthermore, the program follows the structure of the MVC pattern. The program is separated into different parts that all have different areas of expertise. The model class holds all the basic functionality to run the program, the view is only responsible for rendering the actual game on the screen and the controller takes user input to then send to the model. In code, the view class contains majority of the functions needed for the view. However, the other parts of the GUI (such as the coin box and tower cards) are in other parts of the program, but still fall under the category ”view” 1.

3.3 Observer Pattern

The observer pattern is implemented in multiple parts of the program. There is a package ”observable” which provides interfaces and classes to facilitate a consequent use of the pattern. It is used in order to reduce the number of dependencies certain classes have.

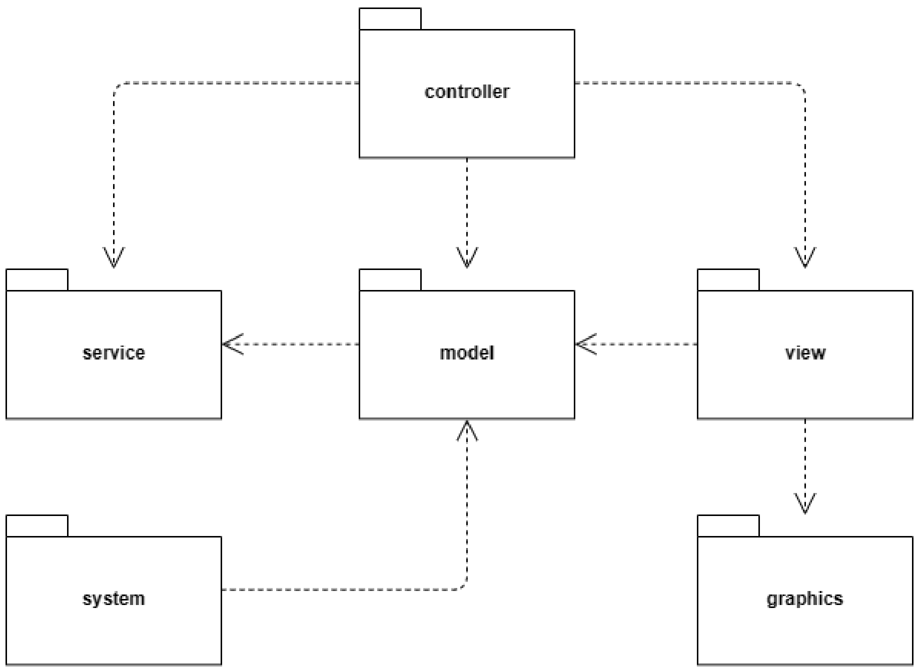

Figure 1: Package diagram for top-level packages

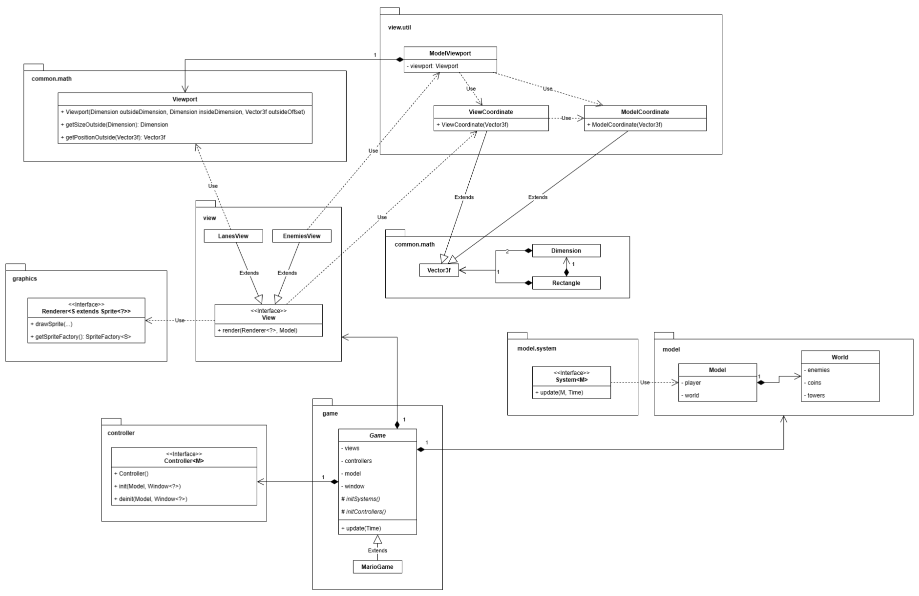

Figure 2: Class diagram for top-level classes

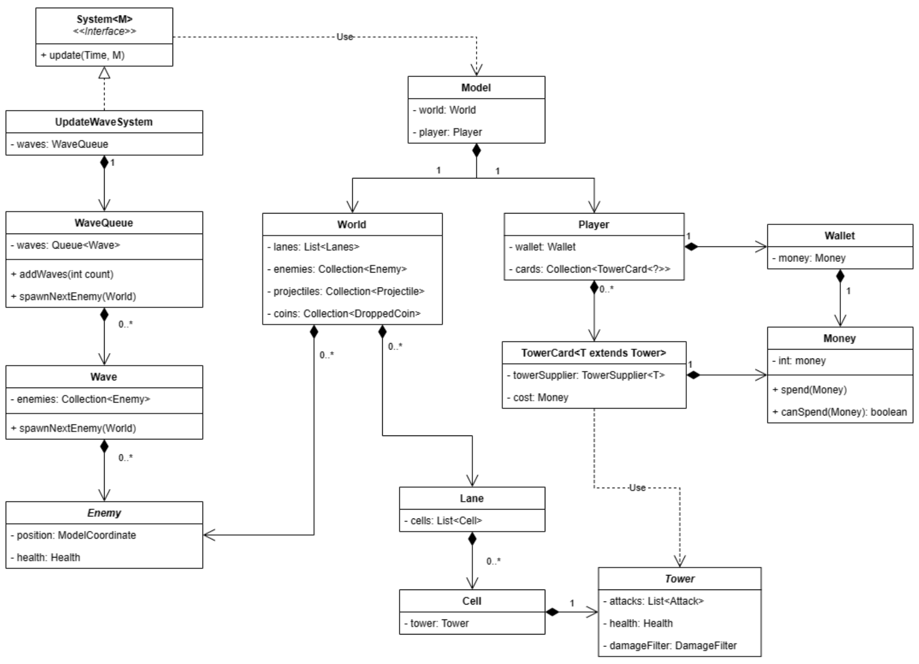

Figure 3: Class diagram for model

4 Quality

4.1 Testing

Unit-tests are located in the 'test' package which is located in the source directory next to the java source folder. The folder structure of the tests mirror that of the java source packages.

The tests are run as a part of the CI pipeline. The tests are executed both by GitHub actions as well as Travis CI whenever a pull request is made.

A pull request also requires at least two approvals from team members before it can be merged into main.

4.2 Known issues

Here is a list of all known issues in the program. Most issues have already been remedied during development.

- When all waves have been sent out the program crashes.
- The Bob-omb tower does not have an explosion animation.
- There is no feedback given to the player when the game ends.

4.3 Code quality

A PMD run shows several small improvements that can be made in many parts of the codebase. The amount of warnings have been reduced during development. Still, some warnings are left but most of the issues are very minor.
STAN shows a mostly good relation between classes dependency-wise.

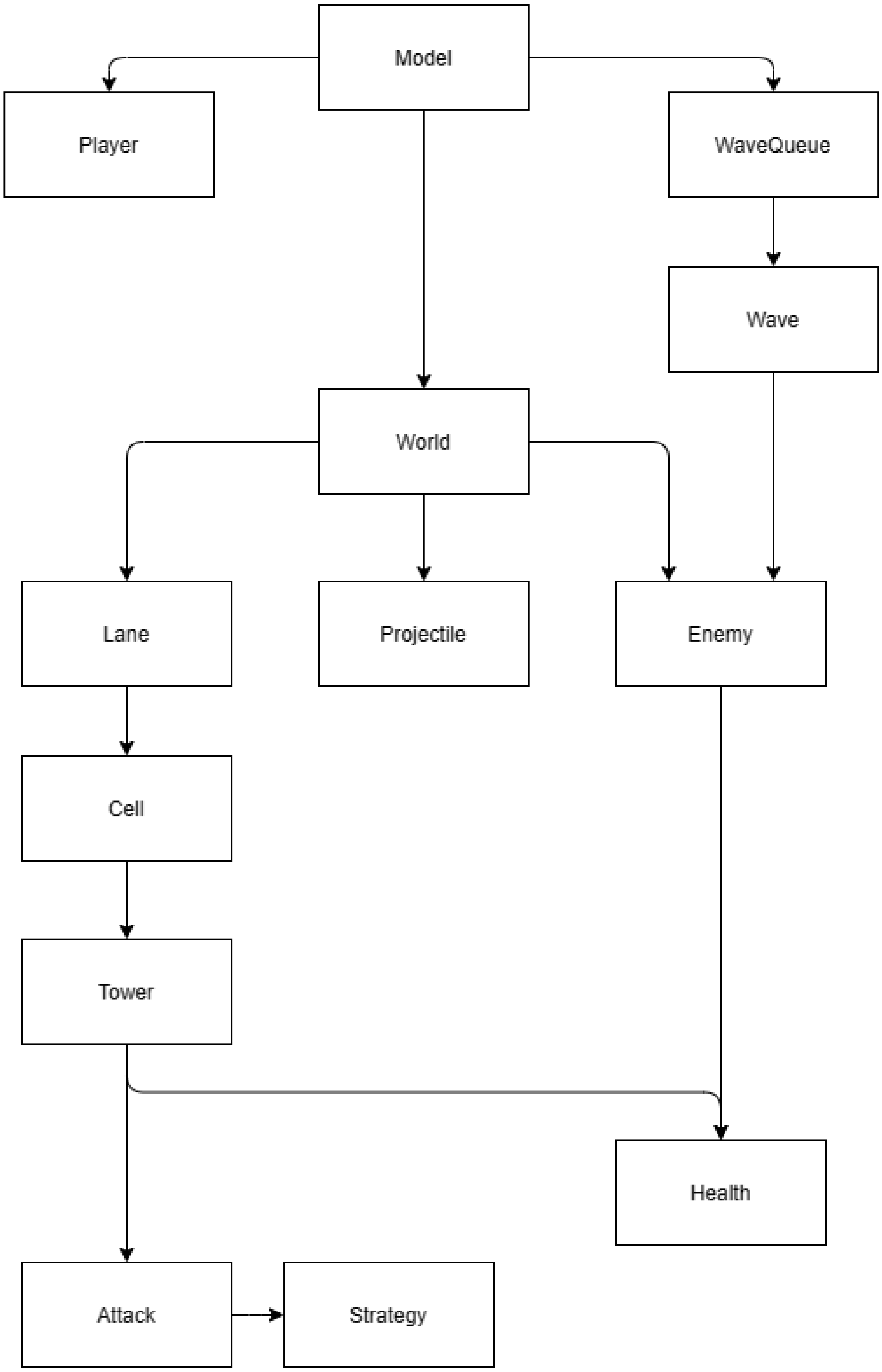

Figure 4: Diagram for domain model

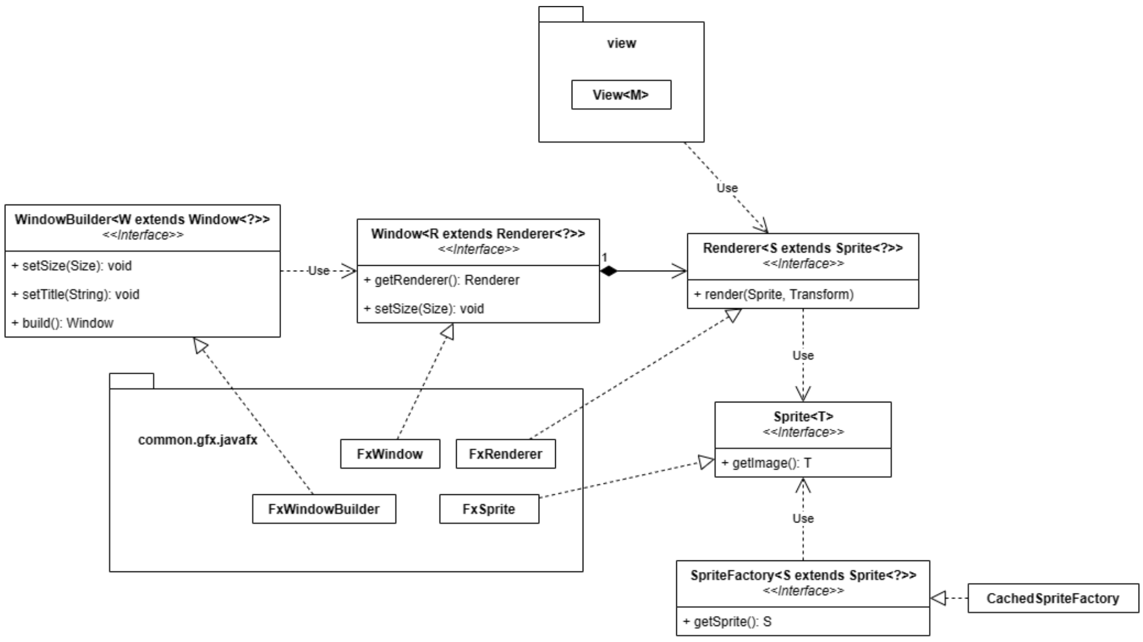

Figure 5: Class diagram for graphics

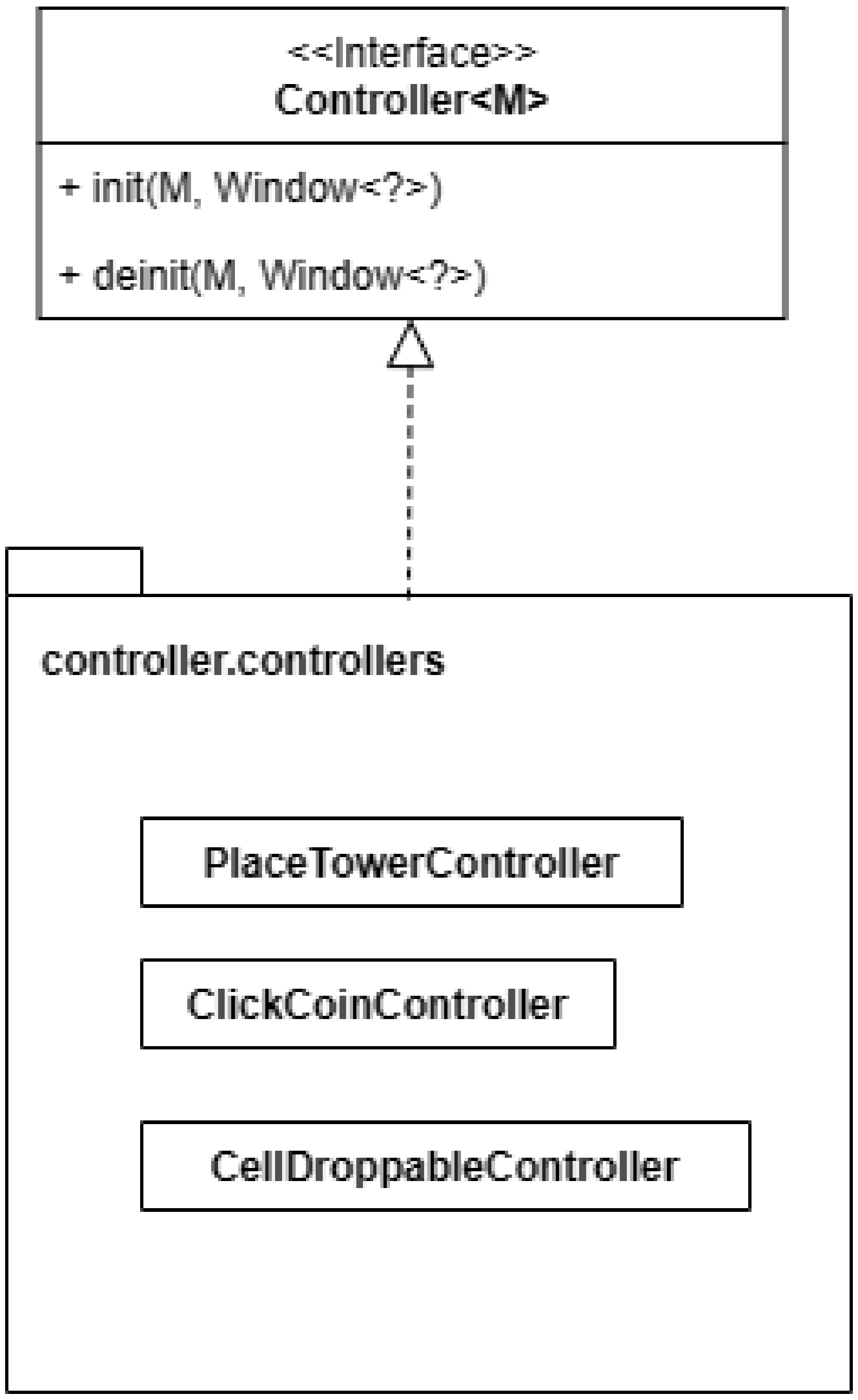

Figure 6: Class diagram for controller

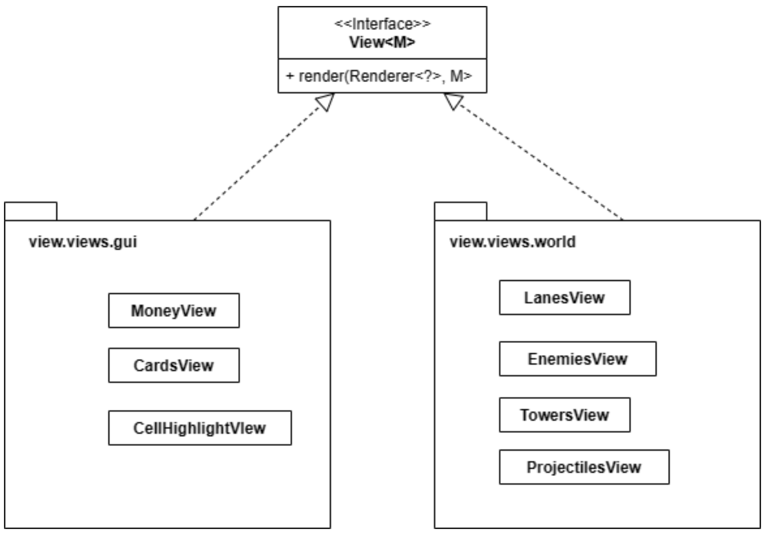

Figure 7: Class diagram for view