TDA367 System Design Document

Smurfs vs. Gargamel

Group 9

Introduction

This System Design Document describes the architecture and the design of Smurfs vs. Gargamel. The document is intended to serve as a guide for the development team to ensure consistency in implementation and make sure the project is adjusted for future enhancements of the system. It provides a overview of the design through class diagrams, sequence diagrams, and state diagrams.

Smurfs vs. Gargamel is a tower-defense game where players strategically place smurf units to defend against waves of Gargamel's forces. The game emphasizes resource management, fun and innovative units, together with increasingly challenging levels.

The goal with the project is to provide an engaging and strategic gameplay experience, with the benefit of introducing players into the smurf universe.

System Architecture

High-level Architecture

The system uses a modular, object-oriented design with several high-level modules. The main module for game logic is the `Model`. Within the `Model` module, there is information about where all sprites currently are together with their projectiles, and the functions to add new sprites, such as attackers or defenders. `Model` makes sure to avoid unnecessary complexity by using several different managers for different responsibilities.

Another module is `Board`, containing information about the different lanes with their respective cells. There also a module for the panels, together with one for the renderers. The panels store input recievers, notifying `Model` about specific inputs from the user.

A high-level architecture can be diagrammed like this:

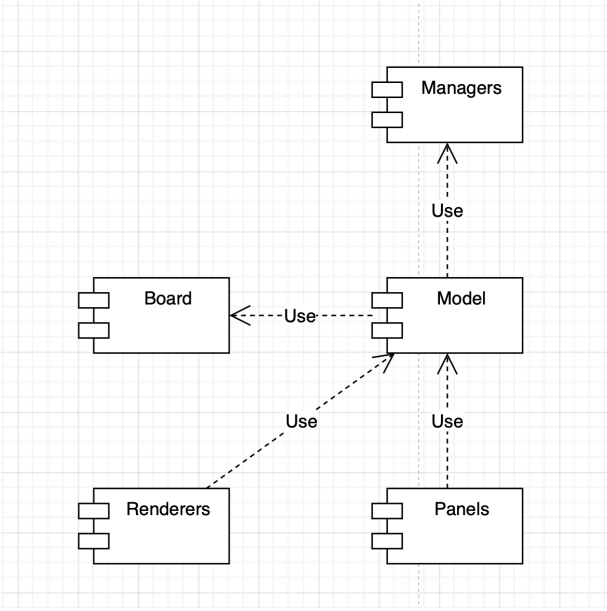

Figure 1: High-Level Architecture of the System

Low-level Architecture

View

Within `View`, the user interface is initialized. The software makes use of two categories of views. These are panels and renderers. In short, a panel is a part of the screen, holding some set of information or functionality. All panels make use of the Java Swing component `JPanel`. The panels store the different parts of the view. For example: `GamePanel` stores the components for the game board and characters, and `VillagePanel` stores the villages. The job for all renderers is the same, render the different elements on the screen. The information to render is fetched from the model. The panels and renderers are summarized as such:

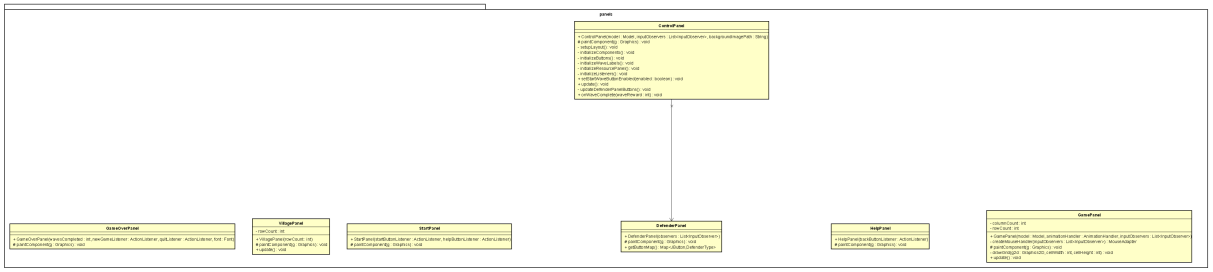

Figure 2: Panels of the View

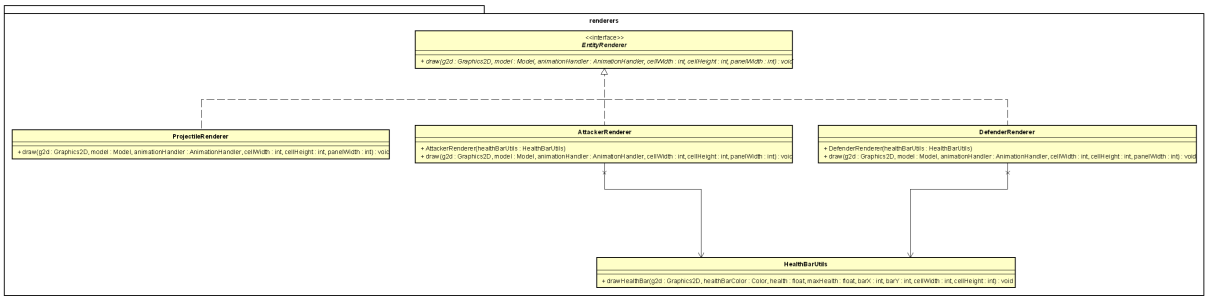

Figure 3: Renderers of the View

Connected with the whole view, the diagram looks like this:

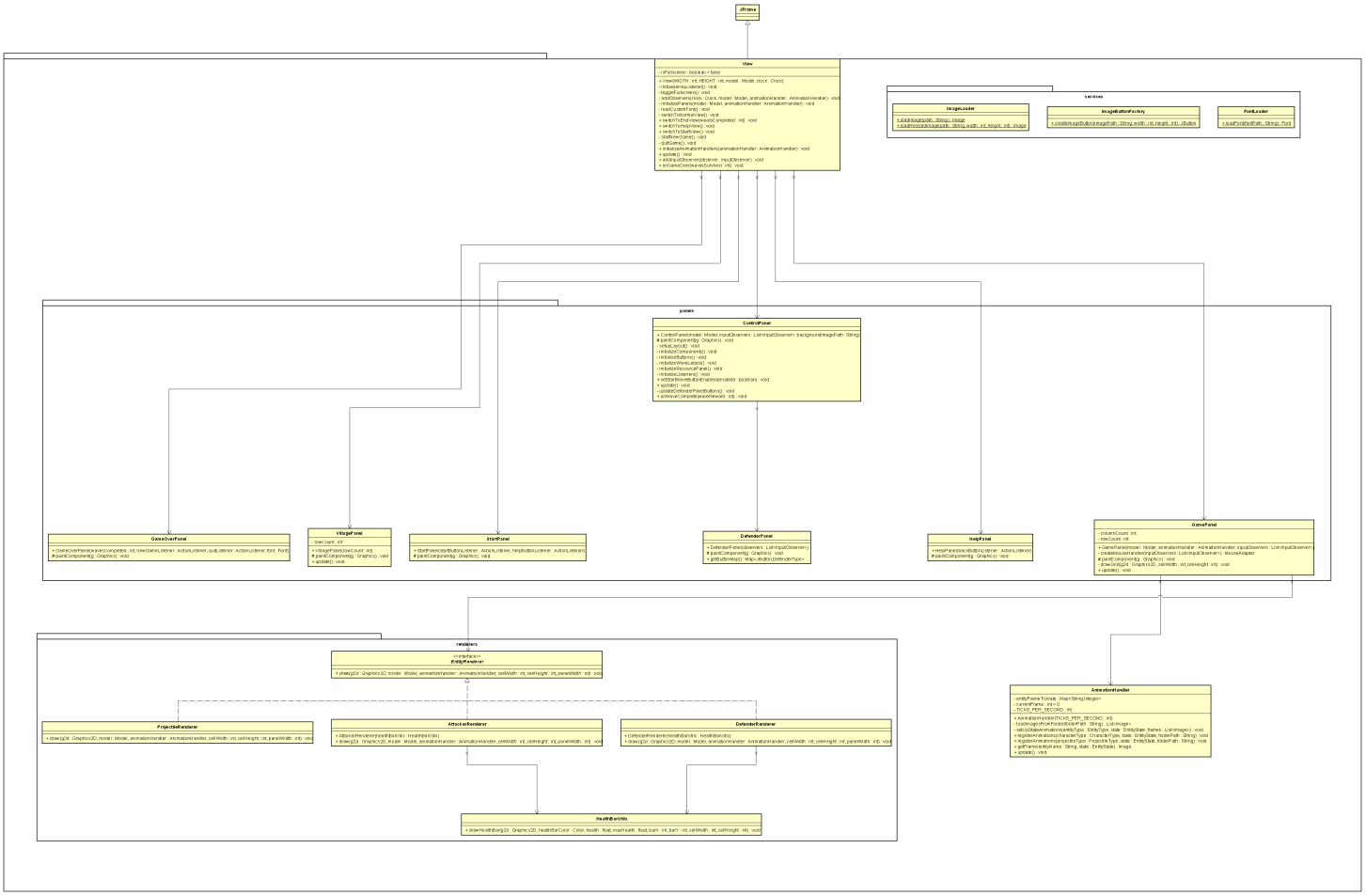

Figure 4: The View

Controller

The `Controller` has the resposibilty to handle and respond on calls from events, such as userinputs. It then brings about changes in the model based on these calls. Our controller mainly consits of the `GameController` class, managing named responsibilities. It acts on mouseclicks on gridcells(for placing defenders), on the defenders in our defenderpanel(for choosing defenders to place) and on the " Start Wave " and " Reset Game " buttons.

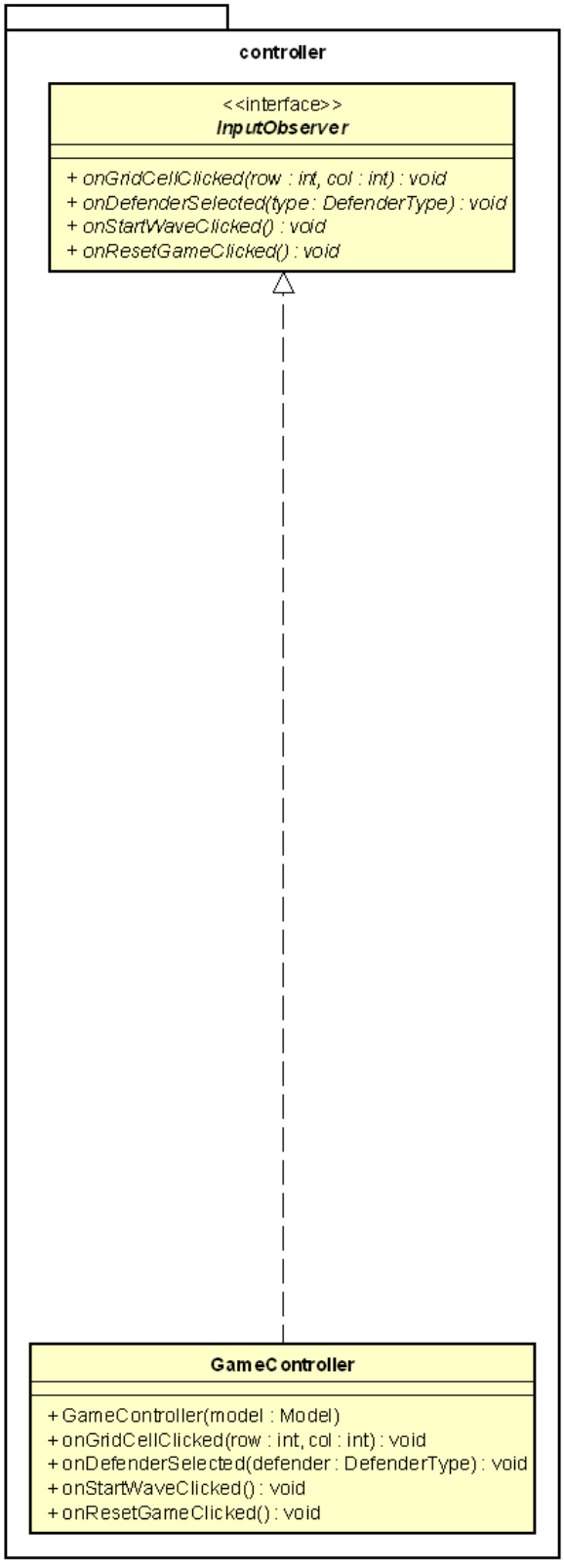

Figure 5: Controller communication

Model

The final part to showcase is the model, and its communication with the board, the managers, and the entities. `Model` is a facade for the game logic. Here lies methods to retrieve and alter information about the state of the game. To split up the responsibilities, `Model` uses a set of managers. It is through the managers that `Model` controls the game. Within the game logic, there also exist some factories. They are used to create entities of varying categories, namely: `AttackEntityFactory` and `DefenceEntityFactory`. Here, the different entities are created to be added into the model. The last main component of the model is `Board`. `Board` represents the main game area. It is the grid on which players place their defenders. Inside the board class, there is logic to handle each lane and its contents. A lane keeps track of its attackers, as well as a number of gridcells. To help the board, there is a `Lane` class, and a `GridCell` class. `GridCell` also stores any placed defender.

Here are the aforementioned components of the model:

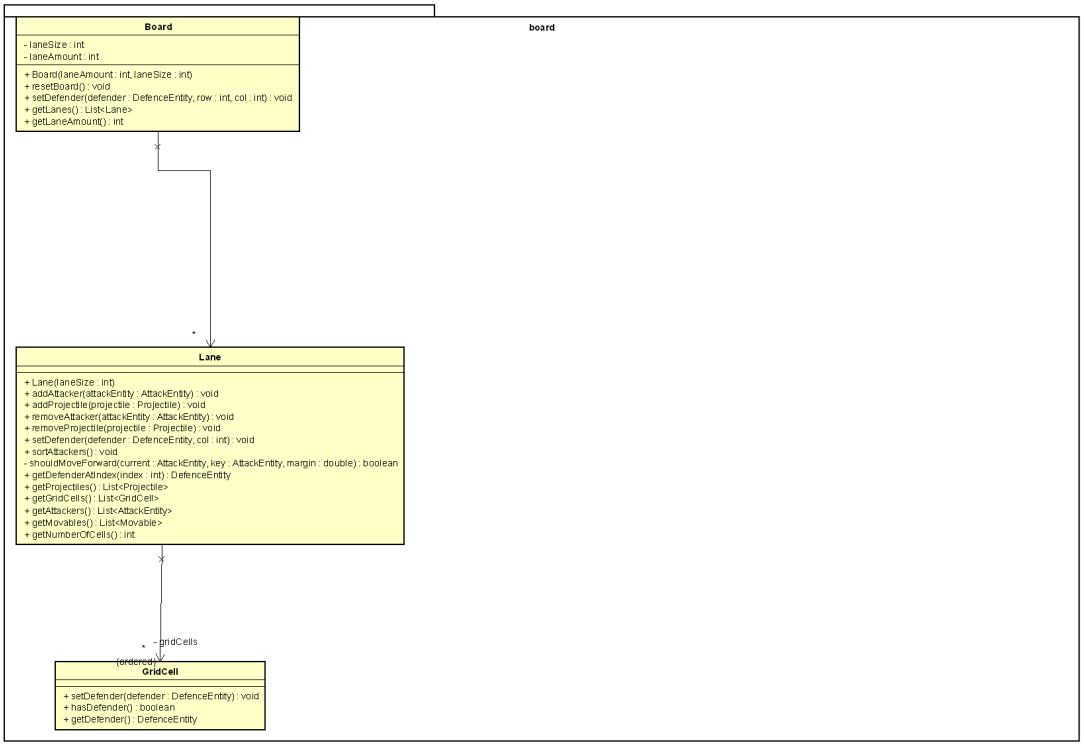

Figure 6: The board component

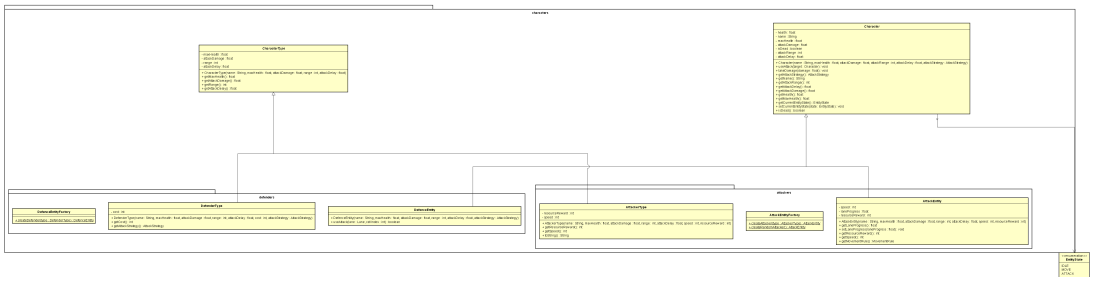

Figure 7: The entities with their factories

Here are also how attacks, projectiles and managers work:

Figure 8: Attacks component

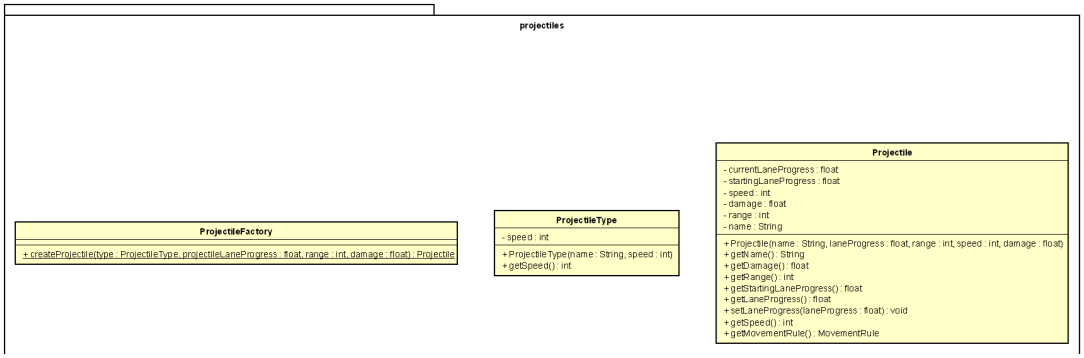

Figure 9: Projectile component

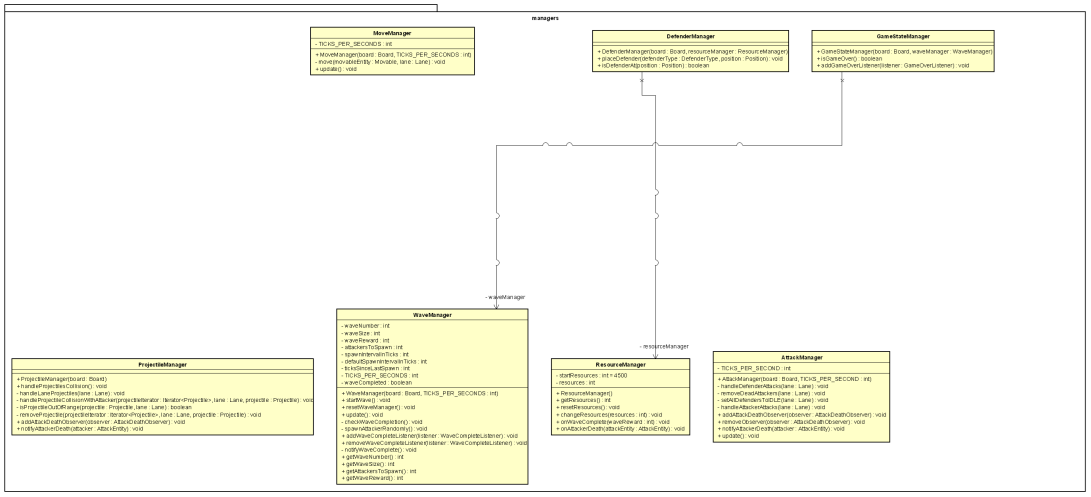

Figure 10: Managers component

In complete, the model looks like this:

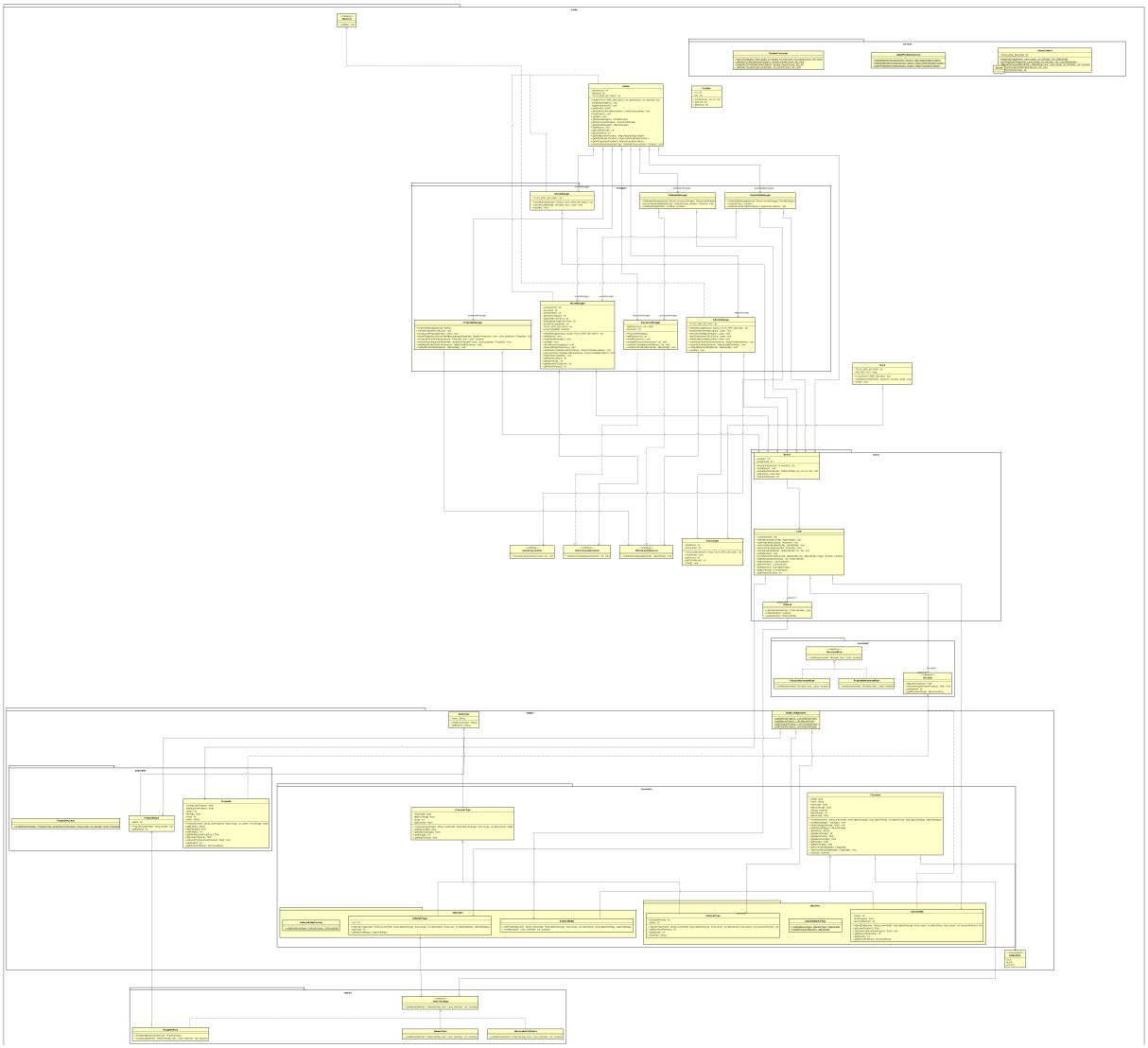

Figure 11: The complete model

Here is a link to the entire class diagram: Diagram

Use of Design Patterns

The system takes advantage of several different design patterns.

MVC Pattern

The system is built using the Model-View-Controller pattern. The model is responsible for the game logic, the view is responsible for the user interface, and the controller is responsible for handling user input and updating the model accordingly. The model is the core of the system, and the view and controller are dependent on the model. The MVC pattern separates the concerns of the system, making it easier to maintain and extend. It also makes the system more testable, as the different components can be tested independently. It also makes it easy to change controllers or views without affecting the model.

Facade Pattern

The model is utilizing a facade pattern to have a clear entry point for packages that are dependent on the model (`View` and `Controller`). Model is delegating each method call to its different managers, which hides the underlying logic from classes dependent on the model.

Observer Pattern

During the game, there are multiple events that are interesting to more than one component. To communicate when such an event happens, we make use of the observer pattern. There are three main observers implemented: `ClockObserver`, `AttackDeathObserver`, and `WaveCompleteObserver`. `ClockObserver` is implemented by every class that should be updated every tick. That means, whenever a new tick is stepped, `GameModel` notifies all classes that have implemented `ClockObserver` that it is time to update. Same applies for the other interfaces: they are notified when an attacker dies, and when a wave is completed, respectively.

Factory Pattern

To make the creation of entities more flexible, there are implementations of two factory classes. These are `AttackEntityFactory` and `DefenceEntityFactory`. The factories are used to create entities of varying categories. This way, the creation of entities is decoupled from the rest of the system.

Strategy Pattern

The strategy pattern is used to make the different entities more flexible. Each entity has a strategy for how they should behave. For example, `Archer` has `RangedAttack` as its attack strategy, and`Movables` have a strategy `canMove` which each movable implements.