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:

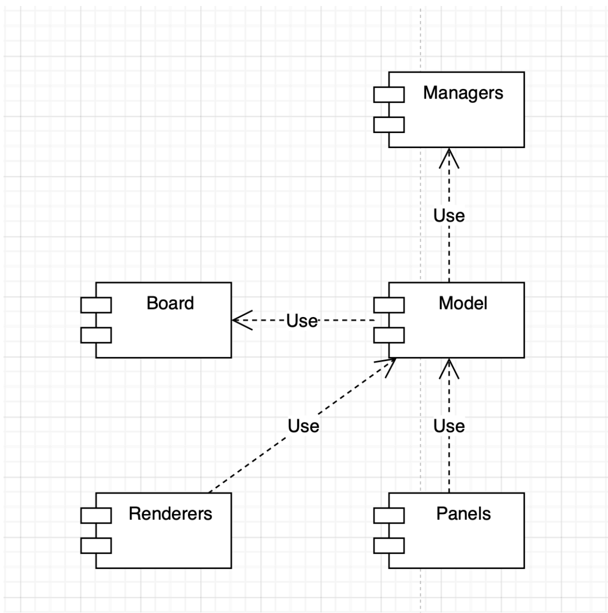

Figure 1: High-Level Architecture of the System

Detailed design

The high-level diagram may be decomposed into smaller, more detailed diagrams. First off, the project should be started from an application class. Within the application, a model should be instantiated to make sure that each component of the project uses the same model. Inside the application, a clock is also instantiated. The purpose of the clock is to keep track of time intervals inside the game. The clock makes sure that updates (such as attacks) happens when they should. `View` is created and recieves the model, while `GameManager` starts the game. The summary is described in a class diagram:

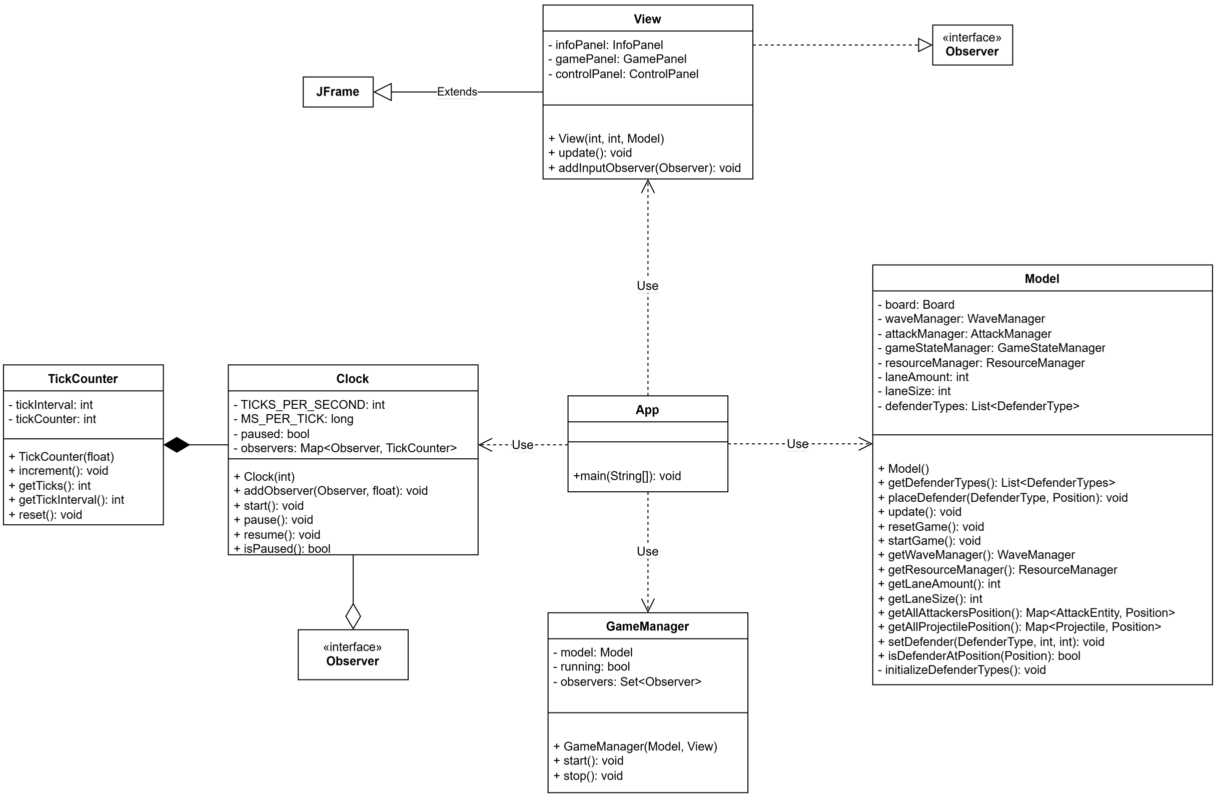

Figure 2: Starting the App

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`. Three main panels exist: `InfoPanel`, `GamePanel`, and `ControlPanel`. `InfoPanel` displays relevant metrics about the game. It includes information about the wave (such as wave number and remaining attackers) as well as the player's current resources. `ControlPanel` displays the controls available to the player. In the controls there exists a shop, with the options to purchase different defenders, and also the ability to start the next wave. `GamePanel` is the main comoponent in the view. Here, the board is displayed, including all entities that comes with it. It is within `GamePanel` that renderers exist. Here are also three main renderers: `AttackRenderer`, `DefenceRenderer`, and `ProjectileRenderer`. The job for all renderers is the same, render the different elements on the screen. The information to render is fetched from the model. When composed into a class diagram, the result becomes:

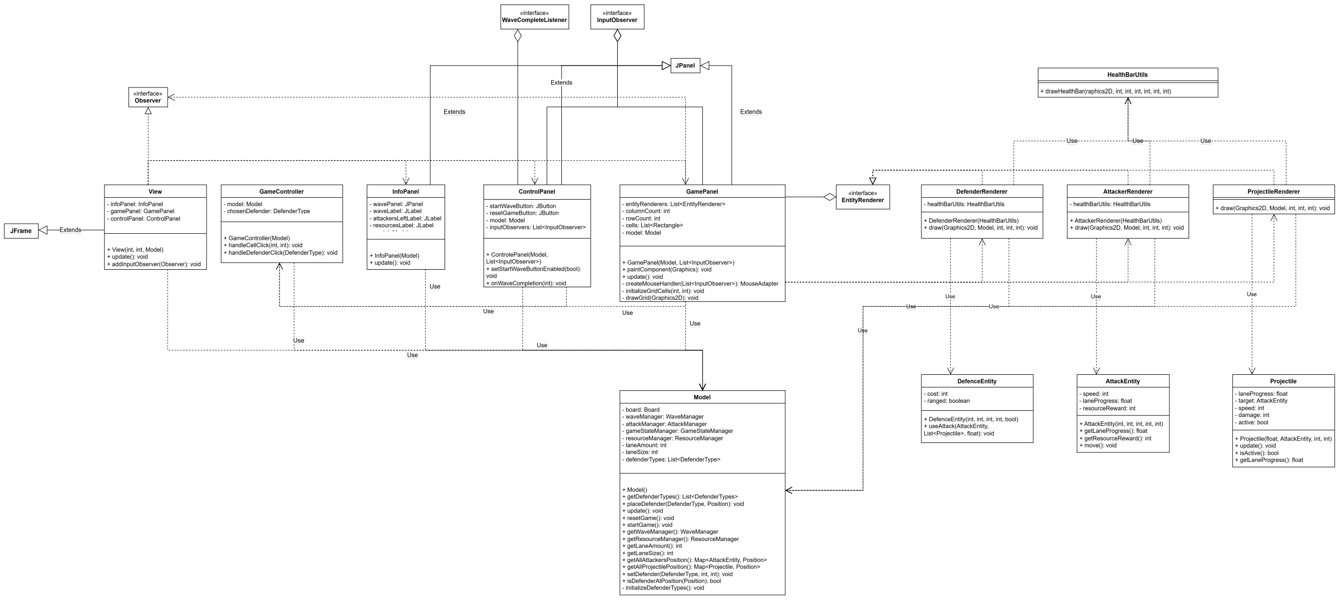

Figure 3: Initializing the View

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. In diagram:

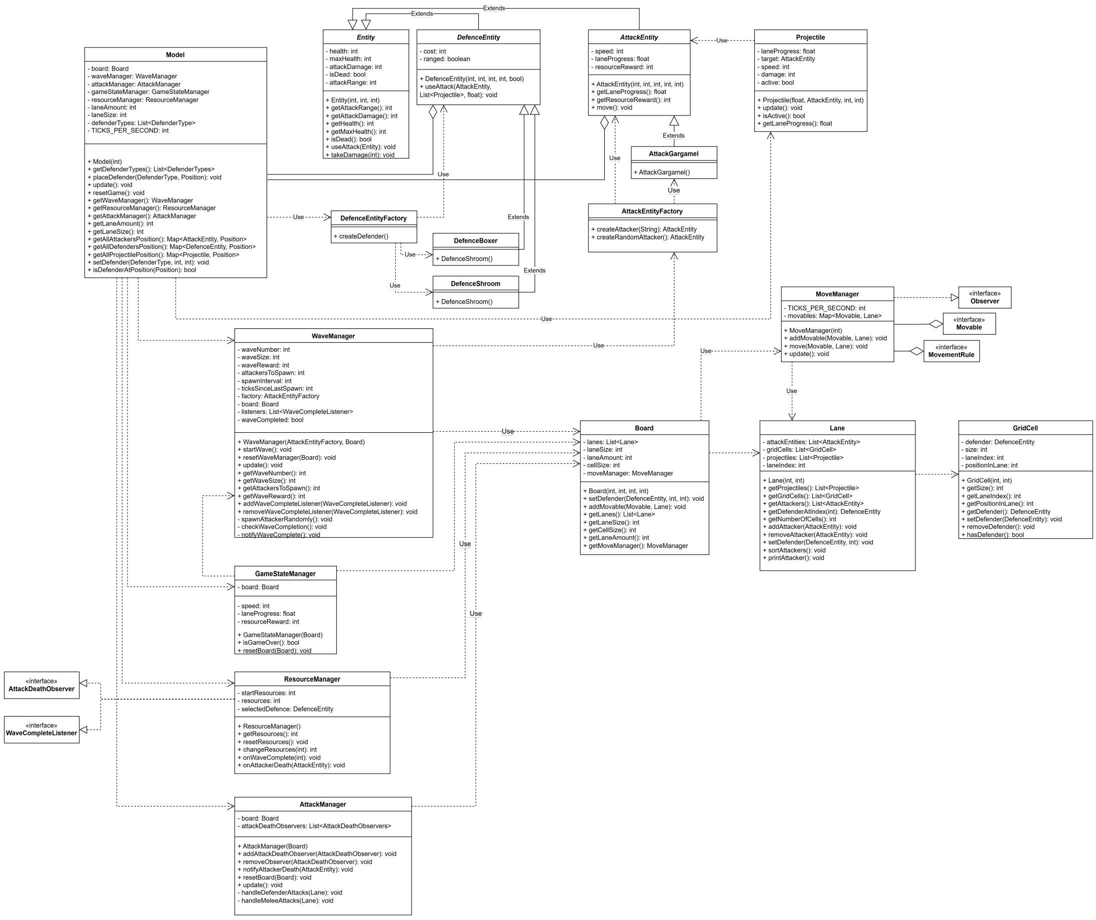

Figure 4: The model with dependencies

Here is a link to the entire class diagram: Diagram

Design Patterns

The system takes advantage of several different design patterns.

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: `Observer`, `AttackDeathObserver`, and `WaveCompleteListener`. The latter two are descriptive, but the first one is not explicit. `Observer` 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 `Observer` that it is time to update. Same applies for the other interfaces: they are notified when an attacker dies, and when a wave is complete, respectively.

Factory Pattern

To make the creation of entities more flexible, there are implementations of two factory classes.