System design document for Alien Defence

Hannes Svahn Felix Nilsson Oscar Forsyth
Jon Emilsson Simon Larsson

September 2020

1 Introduction

This is the System design document Alien Defense. It is a game currently being developed as part of a project in an object oriented programming course. As such, the structure of the code will adhere to basic principles and laws of object oriented programming, which will be described and explained in this document. The application itself is in the classic Tower Defense genre, and the theme of which is Aliens.

1.1 Definitions, Acronyms and Abbreviations

- Sprite: An image that is connected to an object in the game
- MVC: Model-View-Controller, A common architectural pattern in software design.
- TD: Tower Defense, A video game genre, where the player places towers to defend their base from a stream of enemies.
- GUI: Graphical User Interface. What the user can see.
- IDE: Integrated Development System. Where one can write code, quickly compile, run and use other related tools.
- OOP, Object-Oriented Programming. A programming paradigm characterized by the use of objects containing data and code.

2 System architecture

Since the application is a game and therefore combines graphics with user inputs, it follows the classic MVC design pattern. Hence the top-level structure consists of three major components: Model, View and Controller.

The model is responsible for the functionality of the game, for example pathfinding and movement, and thus it contains most of the logic. As for the View, it is what is presented to the user, such as visuals and audio. Meanwhile user inputs are handled by the Controller. Together they function like this: The user performs an action which is then communicated to the Model from the Controller. Whatever is changed in the game, depends on the logic in the Model that has been connected to that particular action. The View is then updated to present the current state of the game.

3 System design

3.1 Model View Controller

The MVC design pattern was implemented in the following manner. If we start with the model, it can, as previously stated be seen as the central part of the game, as it hosts the pure logic and state at any given time. The way that this information is communicated to the view is that outside clients (such as a particular view) can add themselves as observers. This is done by calling the method addObserver in the Observer class and game class, which simply adds the given observer to a certain list in the class, and that list gets notifications every time something relevant to that particular observer occurs.

To give an example, say a part of the view is responsible for updating the projectiles flying across the screen. Then it only needs to *implement* the ProjectileObserver interface, and only receive updates relevant to projectile, not for example if the round is over or if an enemy is dead. This follows the **interface segregation principle**.

Worth noting about this relationship, is that the observer/observable relationship only goes one way, thereby enabling the model to not rely on the outside view.

The way that the controller interacts with the model is that it uses the more direct, public methods hosted in model, which serves to affect the state of the game. This is because controllers purpose is to handle user input, and since the application is a video game and not a simulation, at some point these inputs needs to affect the model.

An example of this is when the player uses tells the game to move on to the next round, or wave, a part of the controller calls the method nextRound in the Game class.

Finally, there is an interaction between the Controller and the View, where the controller more or less initiates the view. This relationship is not preferable and does not follow the MVC or OOP guidelines as they are too closely related and is a leftover from using JavaFX. More on this in **List of all known Problems**.

3.2 Other Patterns

Besides MVC there were other patterns that were used.

Observer Pattern was used to let view observe the model, as previously described.

Decorator Pattern was used to upgrade the towers during run time. At present, the upgrades are simple stat changes, but the use of this pattern allows for more substantial, *behavioral* changes in the future. This is done with an abstract class TowerUpgrade, which is extended to by specific upgrades, such as MageTowerRightUpgrade or ArcherTowerLeftUpgrade. Instances of these upgrade act as wrappers of towers, and will make specific changes to methods while leaving others alone. This is done by delegating the unchanged behaviour to the tower that is being upgraded, which it has a refernce to.

Factory method Pattern was used to create new enemies, with constant, hard coded values. This is implemented by using an interface, TowerFactory, with the method createTower. For every tower, there is a corresponding towerFactory that creates it. Later in production, it became apparent that the factories needed a method getPrice as well, since towers have a cost associated with them, that needs to be compared before placing them. The factory method pattern delegates the responsibility of creating a *new* turret, which limits the risk of creating towers unnecessarily or with the wrong stats.

3.3 Package Diagram

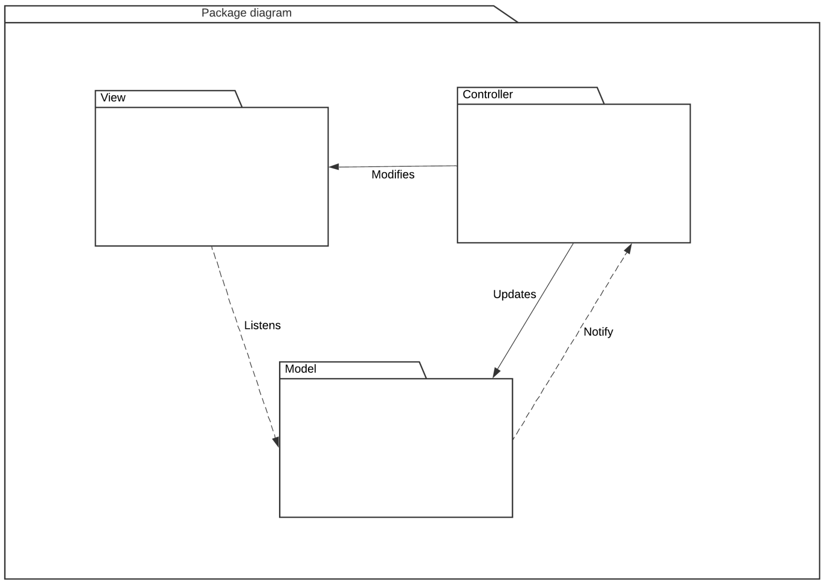

Figure 1: Package Diagram

3.4 Class Diagram

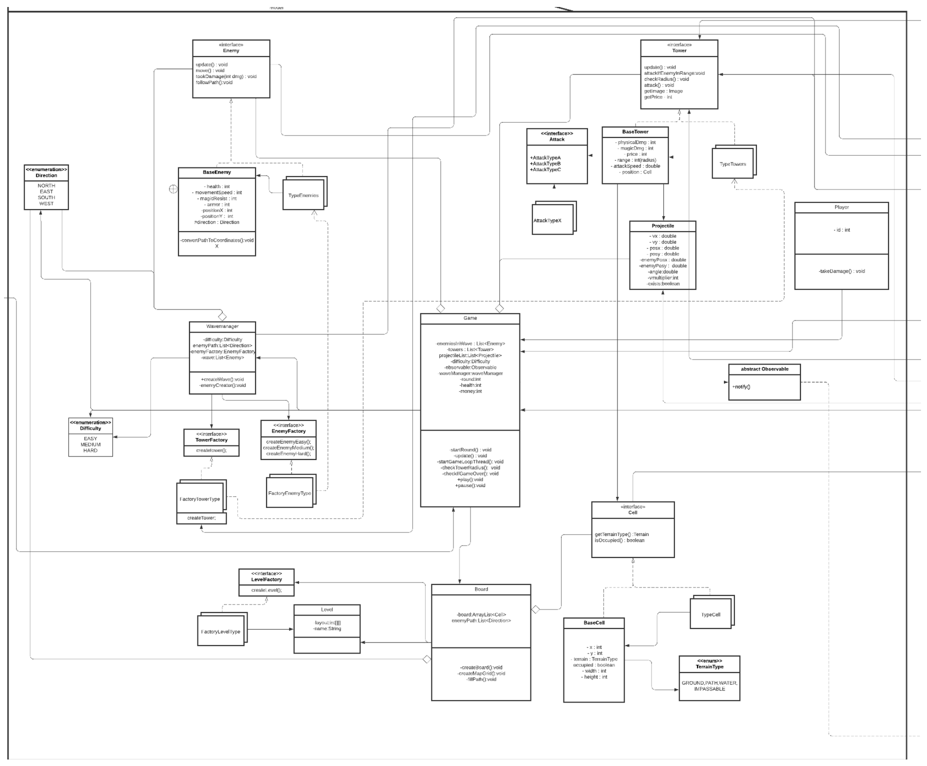

Figure 2: Model UML

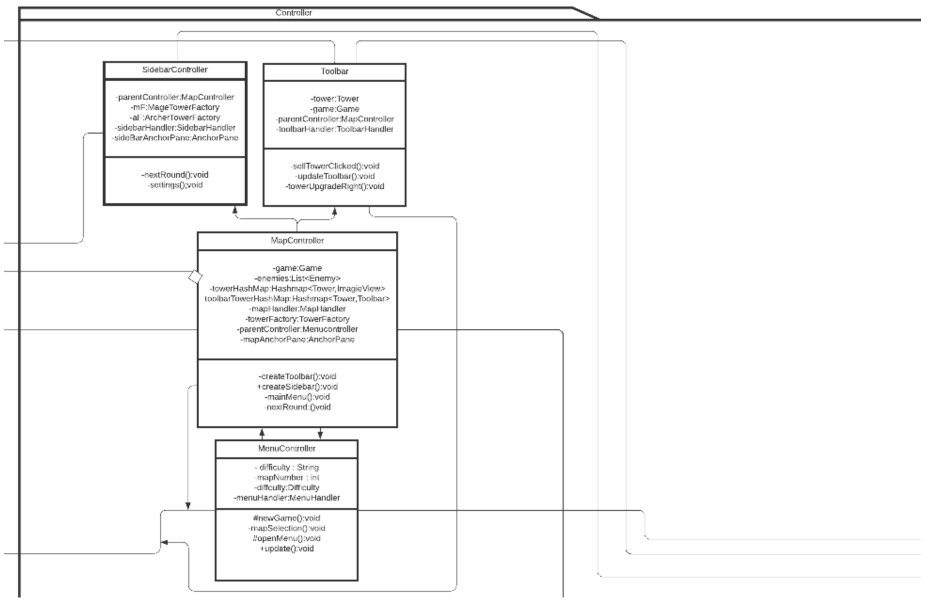

Figure 3: Controller UML

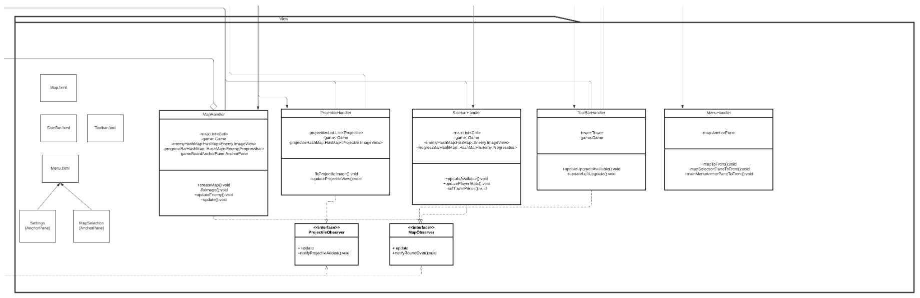
View
Map.fxml
SideBar.fxml
Toolbar.fxml
Menu.fxml
Settings (AnchorPane)
MapSelection (AnchorPane)
MapHandler
-map:List<Cell>
-game: Game
-enemyHashMap:HasMap<Enemy,ImageView>
-progressBarHashMap: HashMap<Enemy,Progressbar>
-gameBoardAnchorPane:AnchorPane
+createMap():void
-fixImage():void
+updateEnemy():void
+update():void
ProjectileHandler
-projectilesList:List<Projectile>
-game: Game
-projectileHashMap:HasMap<Projectile,ImageView>
-fixProjectileImage():void
+updateProjectileView():void
SidebarHandler
-map:List<Cell>
-game: Game
-enemyHashMap:HasMap<Enemy,ImageView>
-progressBarHashMap: HashMap<Enemy,Progressbar>
+updateAvailable():void
+updatePlayerStats():void
-setTowerPrices():void
ToolBarHandler
-tower:Tower
-game:Game
+updateUpgradeAvailable():void
+updateLeftUpgrade():void
MenuHandler
-map:AnchorPane:
+mapToFront():void
+mapSelectionPaneToFront():void
+mainMenuAnchorPaneToFront():void
<<interface>> ProjectileObserver
+ update
+notifyProjectileAdded():void
<<interface>> MapObserver
+ update
+notifyRoundOver():void

Figure 4: View UML

3.5 Domain model

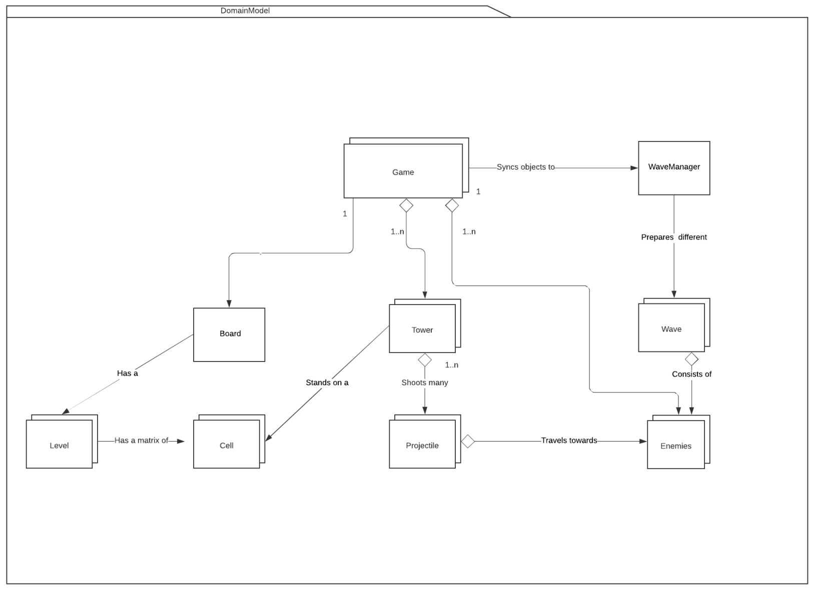

Figure 5: Domain model

4 Persistent data management

Presently, there are some persistent data: Sprites for the monsters, the towers, as well as icons scattered around the game are stored in a dedicated 'img' -directory in the View-package. There is also a soundfile played evert time a menu choice is made, which is stored in a 'sound' -directory.

5 Quality

5.1 Testing

During the development of the project testing was used throughout to assure quality. This was done by the use of JUnit, a unit testing framework for java, which helped out with assessing specific parts of the code without having to run the entire project. The testing also led to catching unexpected effects of other changes which would not have been caught otherwise.

These tests can be found in the directory 'test', which is stored in the 'src' directory.

5.2 List of all known problems

The program has known issues at a top level in the form of Controller having parts that belong in view. At first, this could seemingly be refactored without major issues but it soon became apparent that the typical MVC structure would be almost impossible to implement while continuing to use JavaFX.

The problem is that the controller is very tightly connected to view, which of course results in high coupling, breaking the principle of **High cohesion - Low coupling**. This is because of controllers in JavaFX (Note: controllers in JavaFX and MVC are two different things) having a close relation to their respective FXML-files, and for all intents and purposes does not differentiate what components pertains to View and Controller in MVC. Currently the way its structured to circumvent this is that the View directory consists of handler classes for the view related components that are mainly visual, while the controller classes are holding the input related components.

Other issues include an error in properly running threads. Where the program crashes when two threads try have access to the same list and one manipulates the list while an other reads it causing the program to crash (from a concurrent modification error). There is also the fact that pausing and unpausing the game while the full wave is not on the map yet causes the enemies to clump together.

Another error that occurs is IndexOutOfBounds exception. This is also connected to the multiple threads that are running during the game. This problem is somewhat hard to fix as none of the group members has extensive knowledge of parallel programming and the timeframe in which this project is supposed to be due.

At the moment, Game stores information about the cost of different towers in methods getMageTowerPrice() and getArcherTowerPrice(), which is used in sidebarhandler for example. This information should most likely be communicated differently, like for an example a static getPrice method in the tower classes. If more towers were added, they would all need a method each added in Game, which is not modular.

5.3 Dependencies

We ran dependency analysis through Intellj itself, which is our IDE of choice. Specifically we analyzed for any cyclic dependencies and inspected the dependency matrix. For more top level analysis we uses intellij to produce a UML diagram of model, view and controller, to make sure that the dependencies made sense. There are no cyclic dependencies and since we favored composition over inheritance almost entirely, the code is largely flexible and open or extension where it makes sense, such as adding turrets or enemeies, following **The Open Closed principle**.

5.4 Access control and security

This Tower Defense game does not currently have any sort of access control or security. This is because currently the game is only a single player experience with no way of saving the data.

6 References

-JavaFX for View and Controller; Graphics and input from the player

-Scene Builder for JavaFX modification

-Maven for build automation

-JUnit for testing

-Intellij as IDE