System design document for Alchemy-defense

Willem Brahmstaedt, Felix Jönsson,
Johan Lindén, Valdemar Stenhammar

October 24, 2021
Version 1.0

1 Introduction

This document describes the underlying structure of the application. It goes into detail of how the MVC-pattern is implemented and displays the internal structure of each package by describing design patterns used and dependencies between different modules.

1.1 System introduction

Alchemy Defense is a tower defense game implemented written in java. The application is structured according to the MVC pattern and makes use of the JavaFX framework to present the graphical user interface. The backend of the application aims to be as modular as possible, enabling seamless exchange of frontend implementation. Further more, the application strives to enable easy extension of future functionality and possible domain objects such as additional enemies and towers, or even new type of objects that can function within the game.

1.2 Definitions, acronyms, and abbreviations

- Foe: Tries to reach a set end goal to inflect damage on the player.
- Spawn: Instantiate a new foe.
- Tower: Placed by the player upon the map to hinder the foes from reaching a set end position, where upon reaching the foe will inflict damage to the players hit points.

- Board: The main stage of the game. Uses a underlying 2D-grid which can contain and update both enemies and towers.
- Tile: Each cell in the 2D-grid represents a tile that can hold a tower or an enemy.
- Pathfinding: Plotting and generating the (shortest) route possible given a speciefied search condition.
- Breadth First Search: A simple pathfinding algorithm.

2 System architecture

The system architecture is structured according to the MVC pattern and can be divided into three separate parts with distinct responsibility.

2.1 Model

The model contains all game logic and tracks each "living" game object and their state. All computations such as calculating tower range and allocating damage to enemies are done within the model using the standard Java library to minimise external dependencies. Within the application, the model has no external dependencies other than itself.

2.2 Controller

The controller is the entrance class of the application and should be set up with a launcher. It sets up both the model and the view which it acts as a mediator in between. When the user submits input to the view, the controller parses these and triggers various game events within the model. Vice versa, when the model is updated the controller acknowledges this and translates the current game state to view that renders the appropriate images for the user to communicate the game state.

2.3 View

The view only connects to the controller. Its responsibility are restricted to injected images from the controller and send out user input events for the controller to parse. No domain logic should be contained here.

2.4 Basic flow of the application

The application is launched via the class App that extends Application. App holds an attribute *model* that is an instance of the class GameModel which acts as a *facade* to the whole Model-package. The method *start* in App initialises all components that are required to run the application, i.e. both Controllers and all Views visible to the user.

When the application is launched the board presents itself to the player. The player can choose to place towers on the board or start the first wave. Towers can also be placed during an active wave. The wave is over when all foes are gone from the board, i.e. when all foes are killed or has reached the goal. When a foe is killed the player receives a set amount of gold which can be used to buy more towers. A tower can also be sold for a set amount depending on what type of tower it is. When a foe reaches the goal, the player loses one health point.

The goal of the game is to survive as many waves as possible and therefore the game continues to run until the player loses all its heath points.

3 System design

This section discusses the whole structure of the applications design. UML-diagrams are used to facilitate the understanding of the structure.

The structure design of the application follows the MVC-pattern. The idea behind the MVC-pattern is to isolate the back end (the game logic) from the front end (the gui) by seperating the responsibilities into the three modules Model, View and Controller. View displays output to the user, Controller receives input from the user and sends it to Model that processes the input. How this application implements the MVC-pattern will be discussed in a later section.

3.1 Project UML diagram

The following UML-diagrams are those of the highest level packages in the MVC-structure, i.e. the Model, View and Controller packages.

3.1.1 Controller diagram

This package contains all controllers and their interfaces used in the application. Figure 1 shows the cohesion in the controller module.

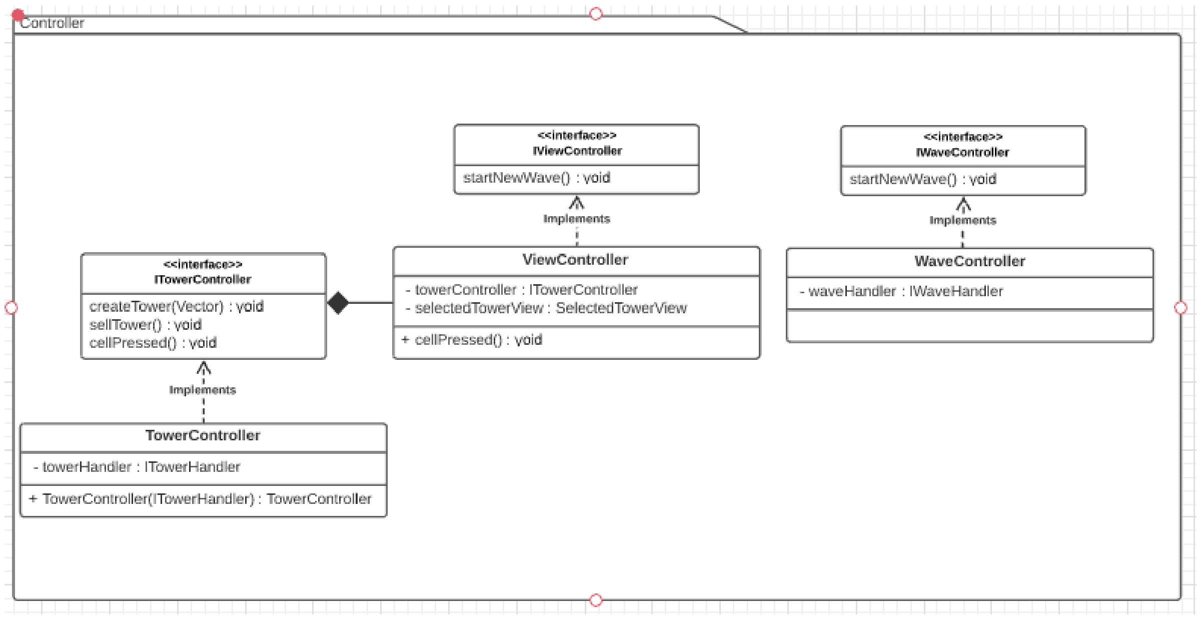


Figure 1: UML diagram over package Controller

3.1.2 View diagram

This package contains all the classes for the GUI. Figure 2 shows the cohesion in the controller module.

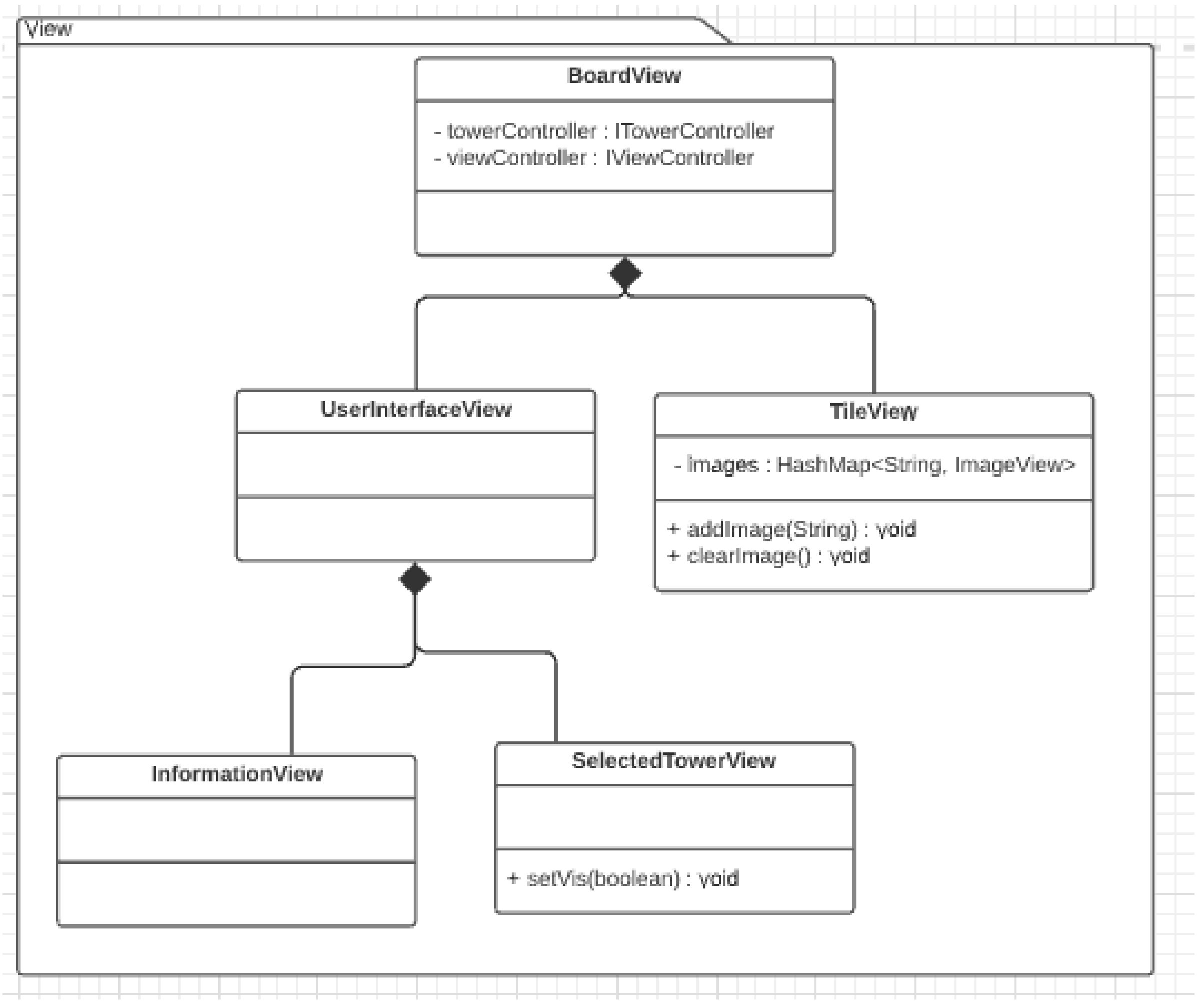

View
BoardView
- towerController : ITowerController
- viewController : IViewController
UserInterfaceView
TileView
- images : HashMap<String, ImageView>
+ addImage(String) : void
+ clearImage() : void
InformationView
SelectedTowerView
+ setVis(boolean) : void

Figure 2: UML diagram over package View

3.1.3 Model diagram

This package contains the back end and represents the whole logic of the application. Figure 3 shows the internal package-structure of Model package. Each individual package will be presented with a class diagram in a later section.

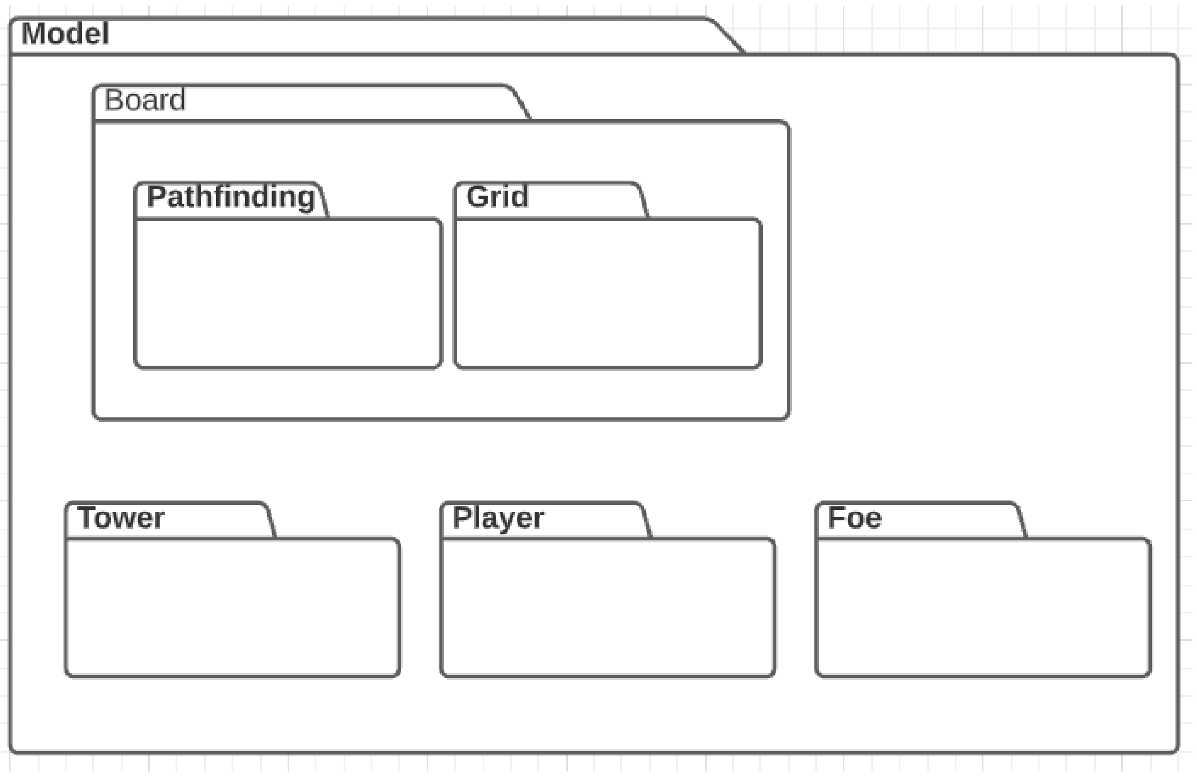


Figure 3: UML diagram over package Model

3.1.4 Implementation of the MVC-pattern and the relations between the modules

Figure 4 shows the implementation of the MVC-pattern in the application. Each view in the View-module has its own graphical appearance and those views that contains buttons also holds a reference to a specific controller. The reference is not to a concrete controller, but rather to the interface that the controller implements. This is done to facilitate a possible new implementation of the controllers in the future. The reference between the Controller and Model shares the same idea to make each module stand alone which subsequently makes each module easy to replace with a new implementation.

The main idea of the MVC-pattern is that the Controller manipulates the Model which in turn updates the View, and that the user sees the View, but uses the Controller. In this application the Model updates the View by the usage of the Observer design pattern. This was chosen due to two reasons: (i) it is an easy way to notify the View that there has been a change in the Model and (ii) it is easy to add multiple listeners to the same part of the module.

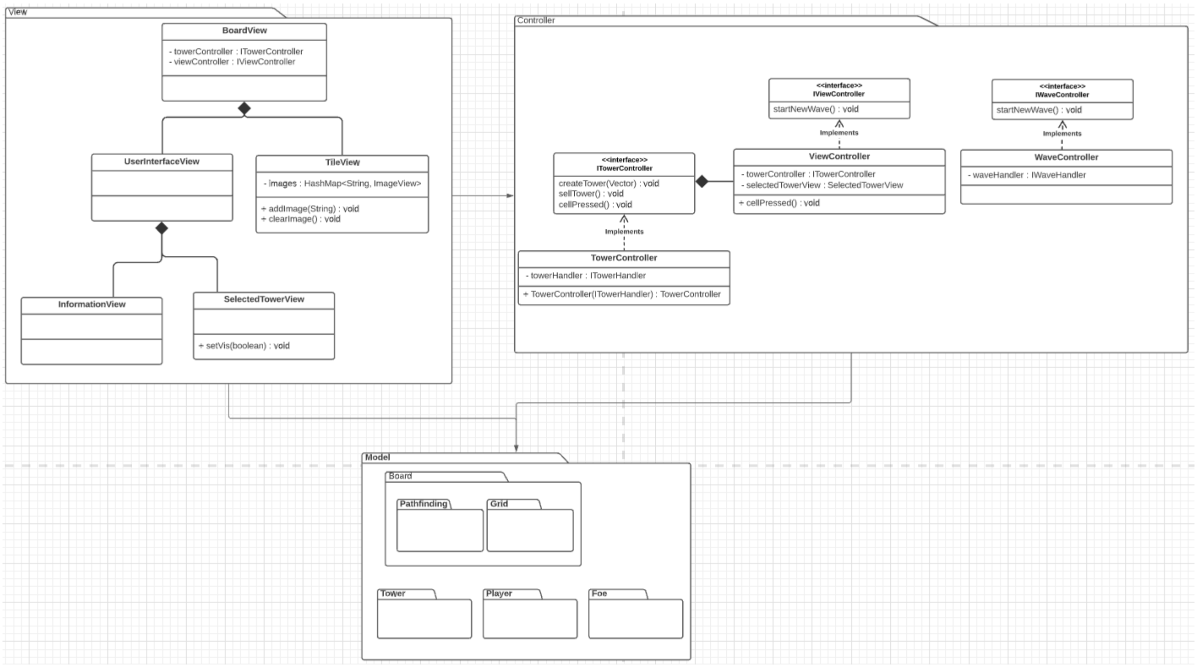


Figure 4: UML diagram displaying the implementation of the MVC-pattern.

3.2 Class diagrams

The following diagram are those of the lower level packages within the Model package. Each diagram presents that packages internal structure.

3.2.1 Tower package

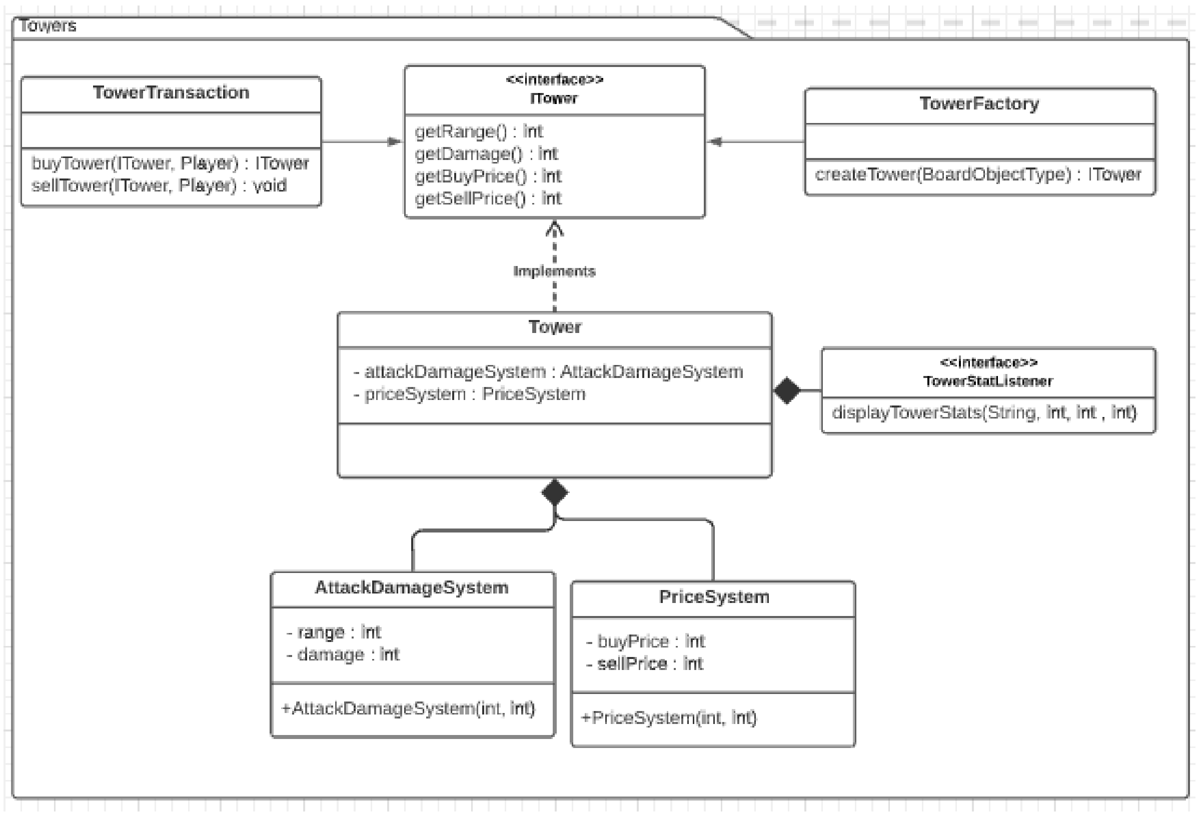


Figure 5: UML-diagram over package Tower

3.2.2 Grid package

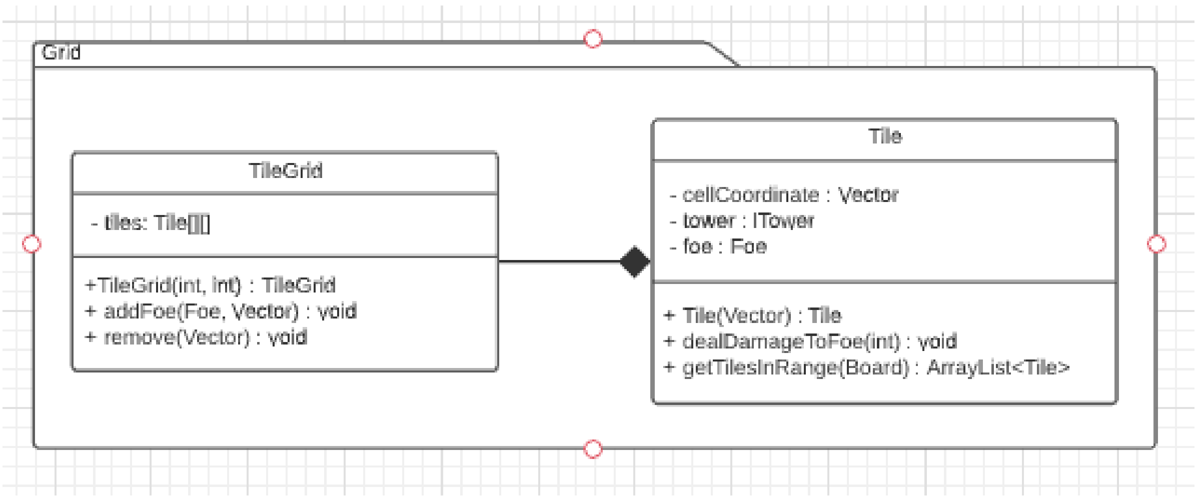

Grid
TileGrid
- tiles: Tile[][]
+TileGrid(int, int) : TileGrid
+ addFoe(Foe, Vector) : void
+ remove(Vector) : void
Tile
- cellCoordinate : Vector
- tower : ITower
- foe : Foe
+ Tile(Vector) : Tile
+ dealDamageToFoe(int) : void
+ getTilesInRange(Board) : ArrayList<Tile>

Figure 6: UML-diagram over package Grid

3.2.3 PathFinder package

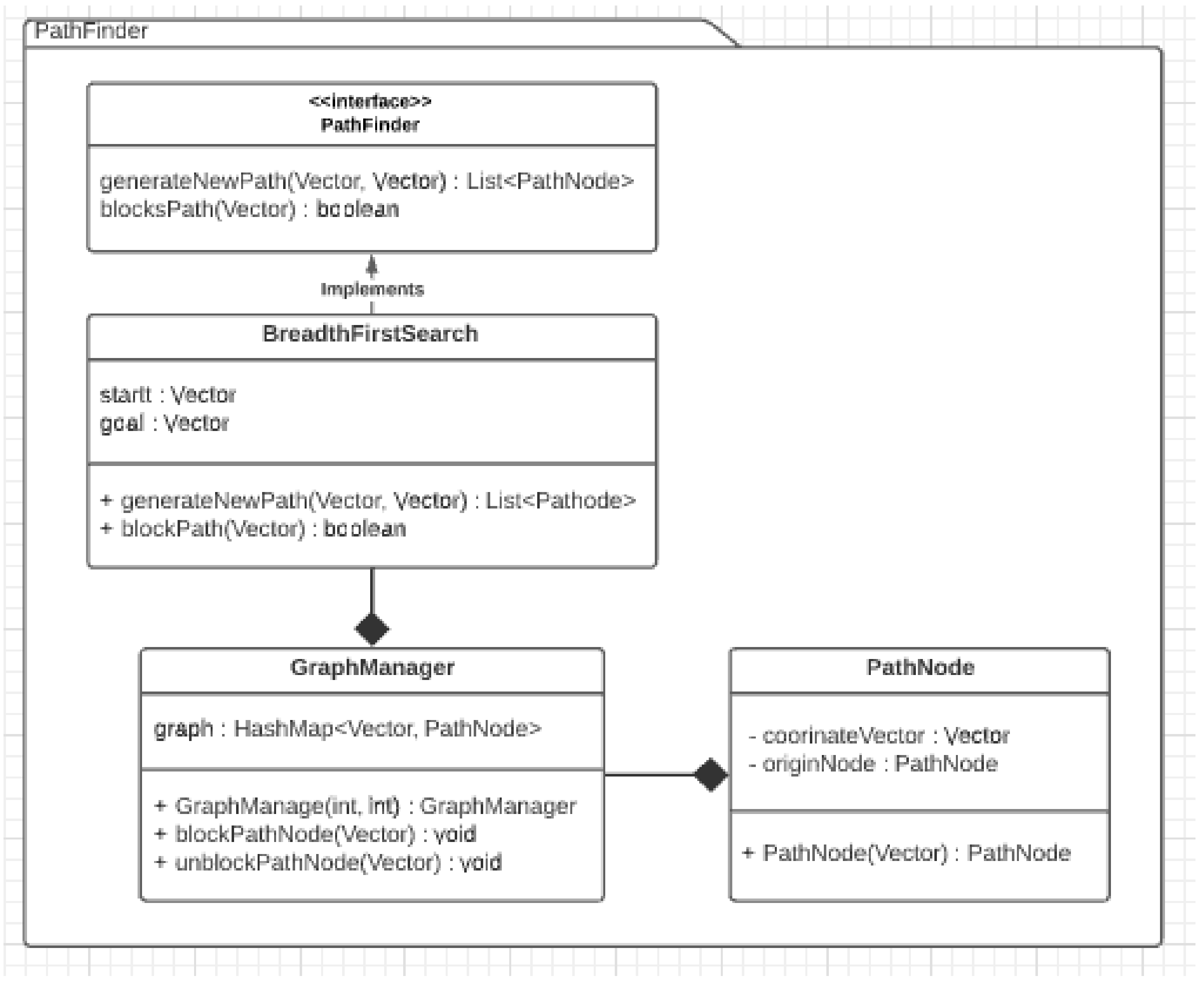

PathFinder
<<interface>>
PathFinder
generateNewPath(Vector, Vector) : List<PathNode>
blocksPath(Vector) : boolean
Implements
BreadthFirstSearch
startt : Vector
goal : Vector
+ generateNewPath(Vector, Vector) : List<Pathode>
+ blockPath(Vector) : boolean
GraphManager
graph : HashMap<Vector, PathNode>
+ GraphManage(int, int) : GraphManager
+ blockPathNode(Vector) : void
+ unblockPathNode(Vector) : void
PathNode
- coorinateVector : Vector
- originNode : PathNode
+ PathNode(Vector) : PathNode

Figure 7: UML-diagram over package PathFinder

3.2.4 Board package

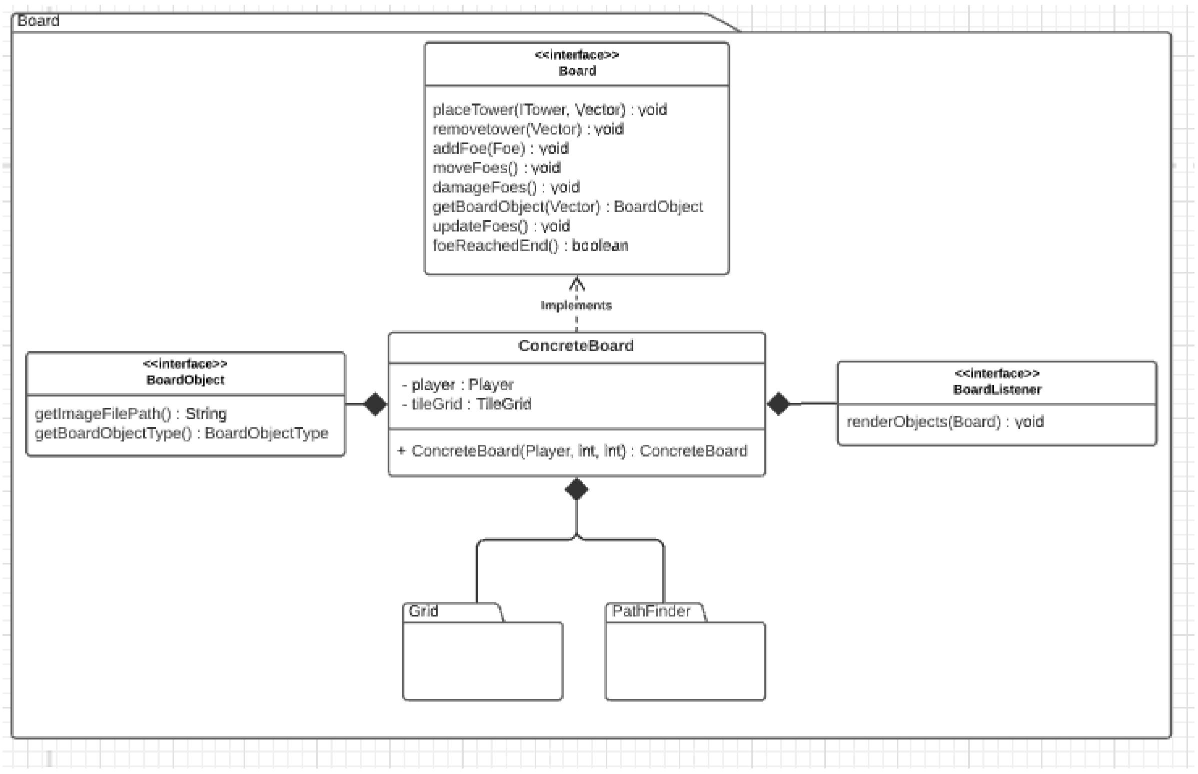

Board
<<interface>>
Board
placeTower(ITower, Vector) : void
removetower(Vector) : void
addFoe(Foe) : void
moveFoes() : void
damageFoes() : void
getBoardObject(Vector) : BoardObject
updateFoes() : void
foeReachedEnd() : boolean
Implements
ConcreteBoard
- player : Player
- tileGrid : TileGrid
+ ConcreteBoard(Player, int, int) : ConcreteBoard
<<interface>>
BoardObject
getImageFilePath() : String
getBoardObjectType() : BoardObjectType
<<interface>>
BoardListener
renderObjects(Board) : void
Grid
PathFinder

Figure 8: UML-diagram over package Board

3.2.5 Foe package

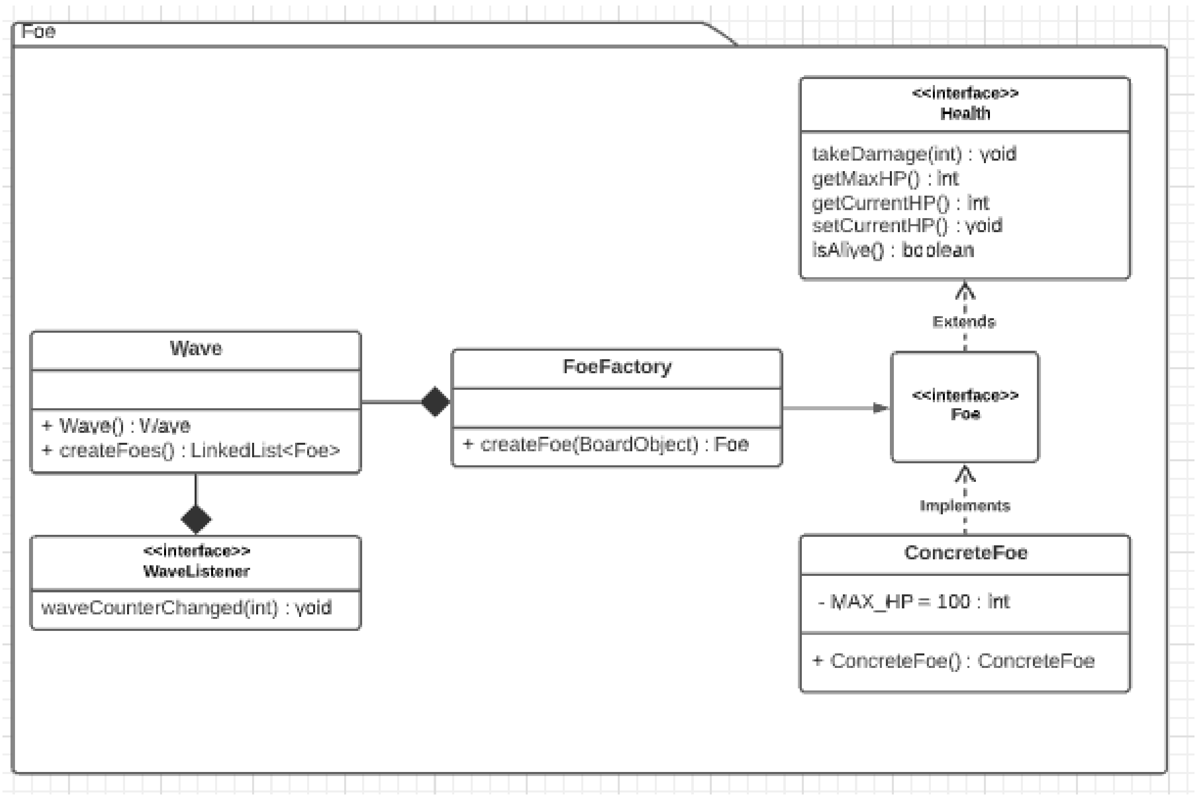


Figure 9: UML-diagram over package Foe

3.2.6 Player package

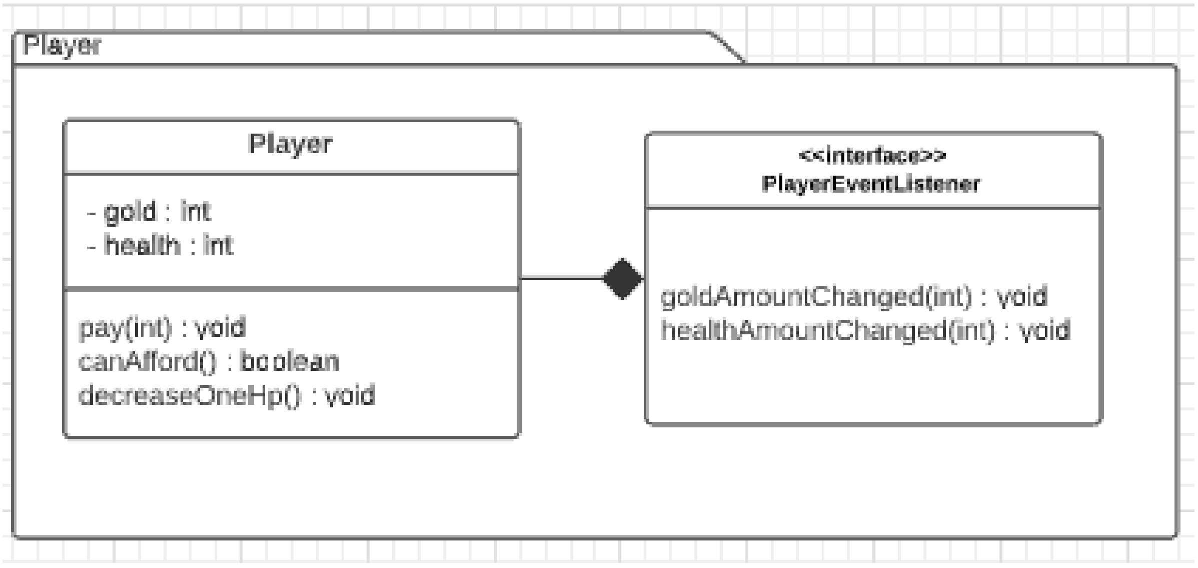


Figure 10: UML-diagram over package Player

3.3 Design patterns used

3.3.1 MVC-pattern

Pattern used to divide a program into three different parts; Model, View and a Controller.

1. The model is smart and models the program. It must not be dependent on the other two modules.
2. The view is dumb and visualises the content of the model without knowing the inner workings of the model.
3. The controller is thin and interprets the user input to then modify the model.

The program uses the MVC-pattern as its basic structure.

3.3.2 Factory pattern

Factory pattern is a creational pattern that decouples the user of a class and the class itself by using a Factory-class. This class returns an abstract instance of the class needed, most often an interface, which the client class then uses. The pattern is used when creating foes and towers.

3.3.3 Dependency injection

Dependency injection pattern describes that an object receives other objects that it depends on, rather than creating them itself. This leads to lower coupling since an object no longer depends on concrete classes but rather on abstractions. Concrete classes which implement/extend these abstractions can then be changed at runtime. The pattern is used multiple times in the program, for example when creating a tower or BoardView.

3.3.4 Observer pattern

Observer pattern enables an object to listen to changes of another class without needing to be dependent on it directly. Lets say class B wants to listen to class A. Implementing the observer pattern in this case would mean B implements an interface that has a method that shall be called upon when A changes. Then one adds class B to a set of said interface in A. A iterates through the set and tells each instance that A has been changed. Alchemy-defense implements this pattern on multiple occasions, there is for example an interface for listeners of the game board or tower stats.

3.3.5 Facade pattern

A facade is an object that 'hides' complex underlying implementation of certain functionality from classes using the facade. This decreases the complexity and therefore the lowers understanding needed to be able to use the functionality. The GameModel class can be seen as a facade for other classes dependent on the module Model.

4 Persistent data management

NA.

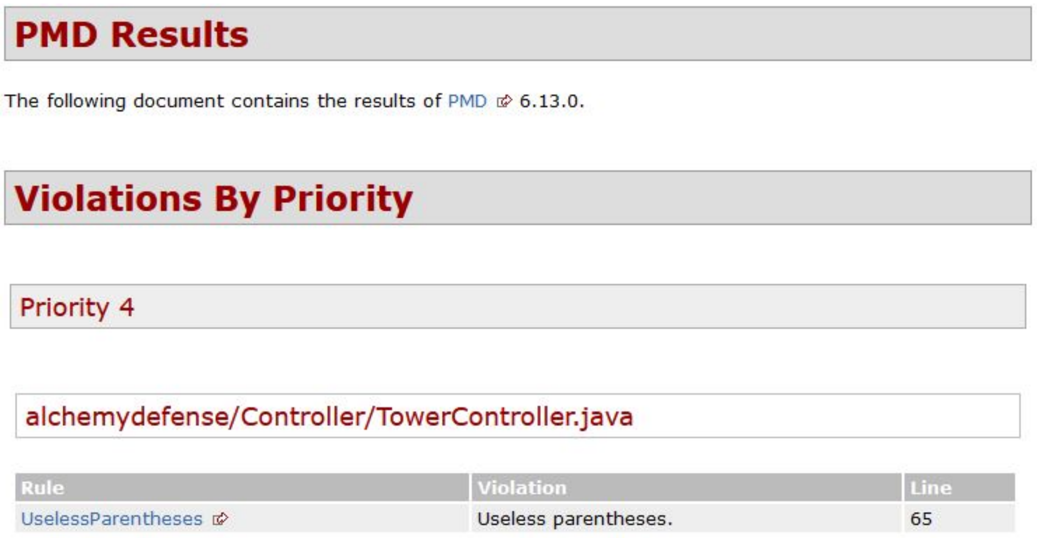
5 Quality

5.1 Tests

The JUnit tests can be found in `/src/test/java`. The test line coverage of the module Model is 93%. However, the coverage on the class GameModel is only 72% which might appear to be low, but the methods not tested are for adding listeners and updating said listeners, i.e. methods for the View. The use of static attributes made writing tests that do not interfere with each other more difficult than it needs to be. Travis was used for continuous integration (link). All Travis credits have been used up, meaning it no longer builds the project automatically. However, the tests still pass when running them on a local computer.

5.2 Analytical tool results

5.2.1 Quality tool report


PMD Results

The following document contains the results of PMD 6.13.0.

Violations By Priority

Priority 4

alchemydefense/Controller/TowerController.java

Rule	Violation	Line
UselessParentheses	Useless parentheses.	65

Figure 11: All violations which PMD found

5.2.2 Dependencies

Below are images from CodeMR, an architectural software quality and code analysis tool. Blue arrows represents a ”has-a” relationship.

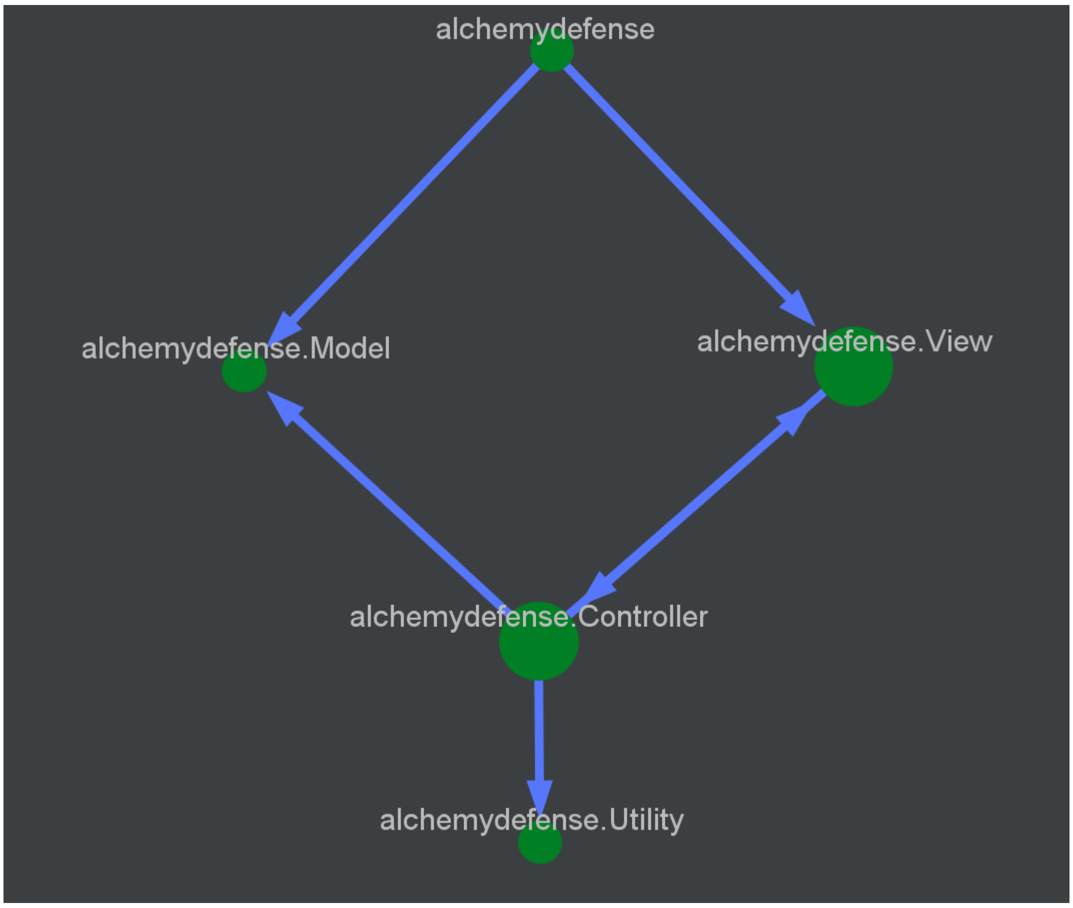


Figure 12: Overall structure of program.

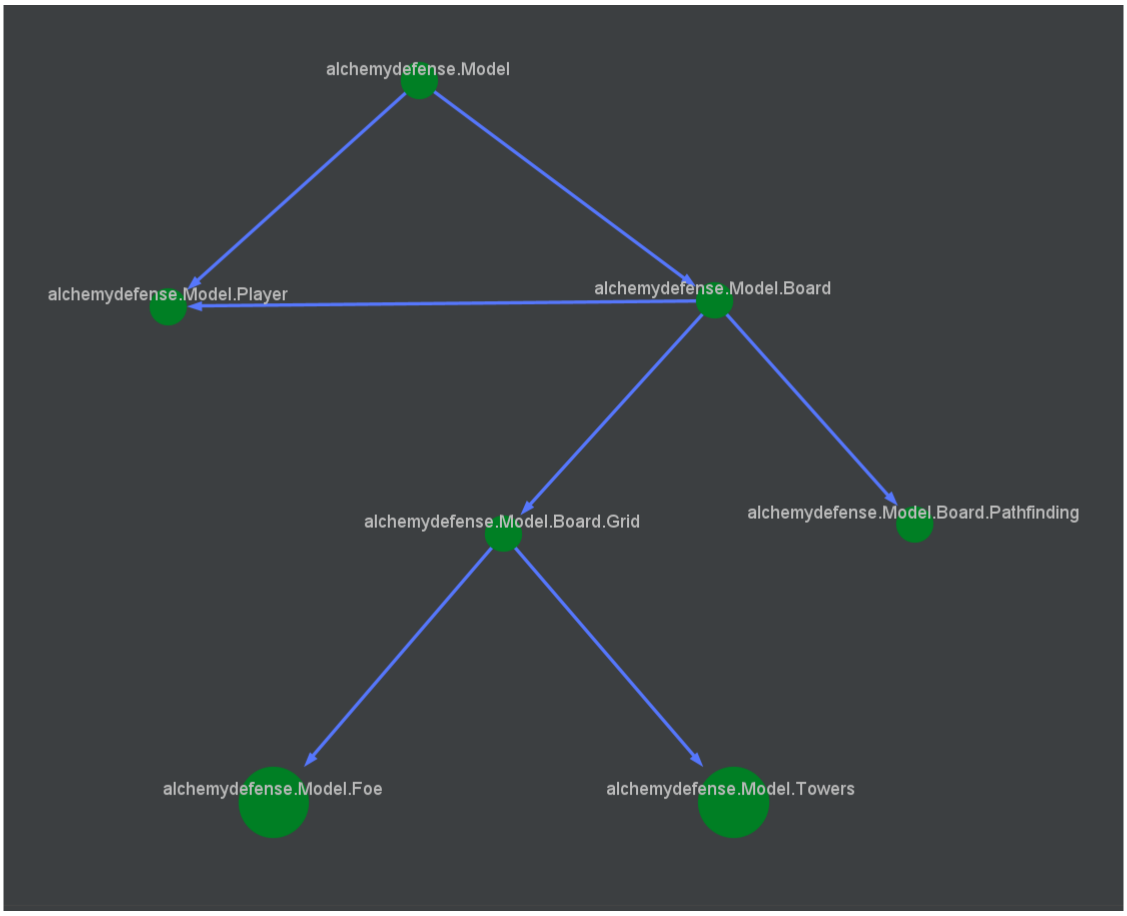

alchemydefense.Model
alchemydefense.Model.Player
alchemydefense.Model.Board
alchemydefense.Model.Board.Grid
alchemydefense.Model.Board.Pathfinding
alchemydefense.Model.Foe
alchemydefense.Model.Towers

Figure 13: The structure in the Model package.

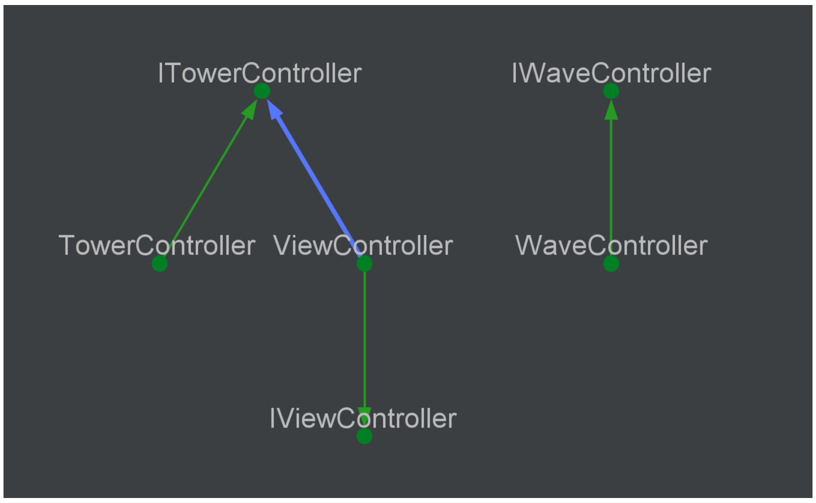


Figure 14: The structure in the Controller package. Green arrow represents implementation.

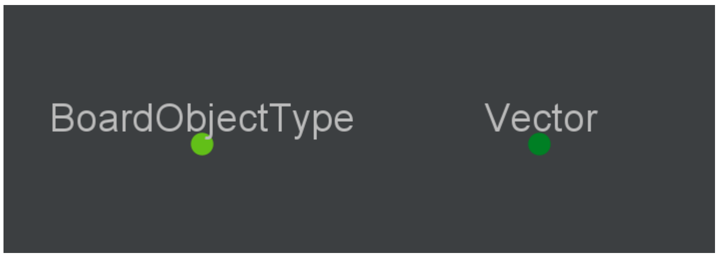


Figure 15: The structure in the Utility package.

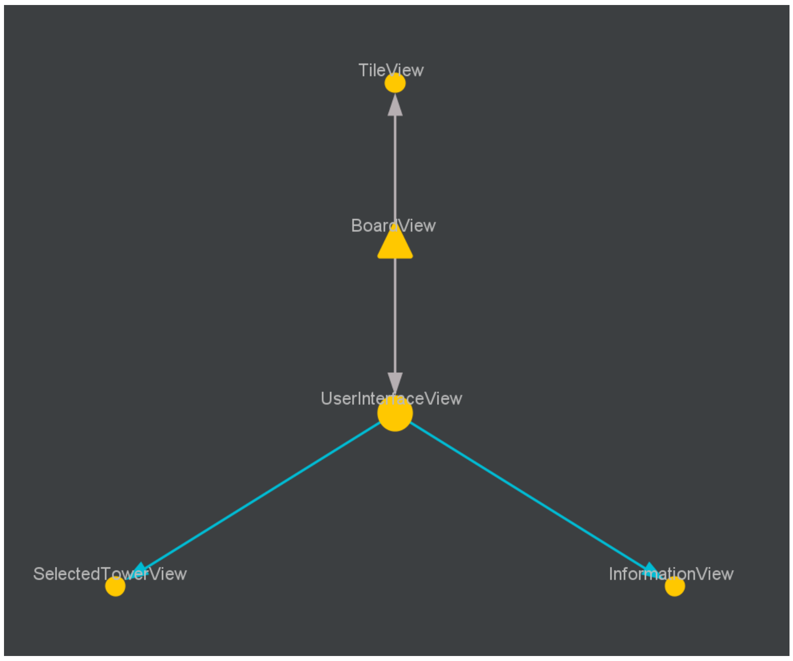


Figure 16: The structure in the View package. Turquoise arrow represents that the class has a parameter of the other. Grey arrow represents a call.

5.3 Known issues

Listed below are the known issues in the code. These would be fixed in later iterations, if given more time.

- A tile of the grid can only be occupied by one foe. Consequently, if two foes try to move to the same square the first one is overwritten by the second one and only the second foe remains. A concrete example of this is when two foes meet at the last tile before the goal; one foe is cancelled out and the player loses one health point instead of two.
- All towers damage all foes within their range instead of targeting just one. This enables the possibility to have a certain structure of the towers that will always kill all foes no matter how many foes are spawned.

- There's only one type of Foe and it always starts with 100 health points. This is an extension of the above issue since a certain tower structure will always kill all foes.
- When expanding the program to full screen there's a white bar on the right edge of the window. Additionally, for some members of the group, when trying to place a tower and clicking on the left side of the tile, the tower is placed in the tile to the left.
- When the player dies, i.e. when the player has 0 or less health points nothing happens. It's still possible to start a new wave and the players health turns to negative.

5.4 Access control and security

NA.

6 References

- **JUnit:** Library used for unit testing Java code.
- **JavaFX:** Library used for creating a graphical interface in Java.