System Design Document for 0x2EE

Group 1
Arthur Alexandersson, Gustav Gille, Herman Norén, Kasper Ljunggren, Rickard Leksell

Date: 2022-10-20

1 Introduction

0x2EE is a real time game with an isometric perspective based on Boxhead[1] where the goal is to acquire the largest score before the player dies by damage taken from enemies. Score is accumulated through killing enemies. The enemies can drop coins which can be used to buy upgrades, they can also drop potions which will heal you after you take damage.

The application is designed by a Model View Controller architecture [2], as in our view becomes the players' user interface where it then communicates input through a controller which then communicates changes in the model which will be represented through the view.

- **Boxhead** - a game created by Sean Cooper [1] where the player tries to acquire the largest amount of points by killing enemies, before they kill the player.
- **Health Potions -** a potion that gives the player more health
- **Coins -** Currency of the game
- **Armor -** Reduces the enemies' damage, can be upgraded to reduce damage even more.
- **Weapon -** The player has a weapon which can be upgraded to deal more damage.
- **Enemy -** Entities that try to harm the player.

2 System architecture

The components of the program are those of a classic Model-View-Controller application, having an application class that initiates all of the MVC components. The game logic is in the model package, which can operate independently from view and controller, while the controller package and view package are tied to each other and the model package.

Starting the game initiates all the necessary components and starts the game loop through the game loop controller class. The game loop controller then calls on the model to update 200 times per second. There is an option to pause this game loop, and resume it seamlessly. This feature is implemented whenever the game is paused or the player enters the shop.

The view operates based on data it gets from the model when it comes to the ingame sprites. Menus are handled internally by the view, making sure that the different menus are not tied to the ingame happenings at all.

3 System design

For UML DIAGRAMS, see appendix at the bottom.

Principles

Single Responsibility Principle (SRP)

- Through segregating classes code is prohibiting unnecessary coupling in its classes, for example how the shop logic is split up into a distinct class for handling transactions and another specific class which acts as an area where the player can then enter the shop from. This also creates the environment where changes to one specific part only affects one specific class, for example if the shop's coordinates were to move, this would only reflect a change in the Shop class and not the TransactionHandler class.

Open Closed Principle (OCP)

- Throughout the code we follow the open closed-principle by taking advantage of polymorphism. Instead of iterating through concrete classes, we iterate through objects through their common interfaces and call needed methods through it. We can therefore extend the code without having to modify what's already there. One example of this is how the Enemy, Player and BossEnemy classes all inherit the abstract class Entity. The Entity class contains the majority of shared logic (disregarding distinct functionality as the Player's weapon and the Enemy's association with Astar Algorithm movement) and easily creates opportunities for extension of new entities as the "same" methods and variables are in use and no new modifications are needed when creating a new entity from the Entity class. One example of this is how all enemies are updated dynamically through the same method in an enemy list, regardless if it is a BossEnemy or NormalEnemy:

```
1 usage  Rickard Leksell +3 *
private void updateEnemies(double dt){
    Iterator<Enemy> enemyIter = getEnemies().iterator();
    while (enemyIter.hasNext()) {
        Enemy enemy = enemyIter.next();
        if (enemy.getHealth() <= 0) {
            spawner.spawnItem();
            player.addScore(enemy.getKillReward());
            enemies.remove(enemy);
            break;
        }
        enemy.update(dt);
        //Check if enemy is close enough to damage player, could be done somewhere else also.
        if (CollisionHandler.testCollision(player, enemy)) {
            this.player.damageTaken(enemy.getDamage());
        }
        checkIfProjectileHitsEnemy(enemy);
    }
}
```

Liskov's Substitution Principle (LSP)

- In the code there are multiple implementations of LSP. One example is the mention above, how a superclass of enemy is used as a parameter to update different types of enemies.
  Another example being how objects with common superclasses can be substituted with each other and still work. Although we do not implement this as we instead rely on interfaces, a higher form of abstraction, it is possible to change their parameters to Upgradable instead of the interface IUpgradable and achieve these goals, see the pictures below for reference.

```
2 usages  Gustav Gille
private void upgrade(IUpgradable toBeUpgraded){
    newPlayerMoneyAmount( amountAfterTransaction: player.getMoney() - toBeUpgraded.upgradeCost());
    toBeUpgraded.upgrade();
}
```

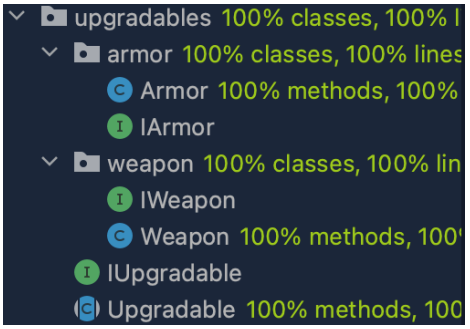
```
2 usages  Gustav Gille *
private void upgrade(Upgradable toBeUpgraded){
    newPlayerMoneyAmount( amountAfterTransaction: player.getMoney() - toBeUpgraded.upgradeCost());
    toBeUpgraded.upgrade();
}
```

(need to change the type of armor and weapon from IUpgradable to Upgradable as they are different types, can alternatively be casted to Upgradable)

Interface Segregation Principle (ISP)

- By creating specific interfaces for classes we can limit the amount of choices and streamline the methods of the classes which implement them. One

example is how segregating the Armor and Weapon classes specific interfaces IArmor and IWeapon from the IUpgradable interface makes the code better suited for extension, thus not inheriting unnecessary methods from the interface when scaling is involved (for example four Upgradable extensions would create at least three unnecessary implementations per specific extension of Upgradable).



- We hold true to the interface segregation principle by creating a number of interfaces for Game so that classes only have access to relevant and abstract methods and data. In this way, you can easily create a version of the game without enemies or projectiles for example, picking and choosing what you want to implement.

Dependency Inversion Principle (DIP)

- Many of the project's classes depend on interfaces rather than on concrete implementations. The most visible implementation of this is for Game, where each class that needs something from Game takes it in as an interface with very few abstract methods, limiting their reach and encapsulating them. Another example of this implementation is how the TransactionHandler class takes in the specific interface IHasUpgrades as a type for player, which only gives the TransactionHandler the following methods (see picture below), limiting the accessibility of the class by depending on the interface, thus depending on the abstraction rather than the concrete class implementation.

Patterns

State pattern - We implement the state pattern in our View, where MainPanel holds a

panelstate. The panelstates call upon the changePanelState method in MainPanel to switch which panel should be painted in the window, meaning that the View can easily change between menus in an object oriented manner.

Factory - These panel states are created in a factory, removing unnecessary dependencies both between the main panel and the different panel states and between the different panel states themselves. Instead they use a factory to create the correct panel state with the help of an enum.

Observer pattern - For now the only observer in the code is the MainPanel class. The reason we implement this pattern is that the code should be open for extension by adding new observers, for example additional panels or a completely new implementation of the graphical interface for the game.

4 Persistent data management

The game's sounds and images are stored in a package within the repository.

There is also a highscore file which is present in the repository to which scores are added when the player completes a game. The player can therefore see the highest scores managed on the device the game is played on, and that information is stored across sessions.

5 Quality

Tests are done through JUnit[3] and have 100% method and 95% line coverage in the model. There are currently 247 tests which all pass, which are located in the testfolder inside the src folder.

PMD has been selected as a quality control tool and has aided in finding otherwise fault code such as possibly dangerous naming conventions in the code-style. Note to make is that all tests represent prone errors in the naming conventions, as underscores are used instead of camel case.

Remaining errors are, in code style, 246 naming conventions errors as we chose to

do our tests with underscores instead of camelcase (one test is written in camel case). This to segregate tests from regular methods using camel cases.
The 5 errors in design are throwing raw exception types and the error prone error consists of calling methods from the constructor, such as init() methods. Last part is performance, where the error present tells us to avoid fileStreaming. The group found none of these errors as threatening but are aware of these errors and how they may affect the code further down the line.

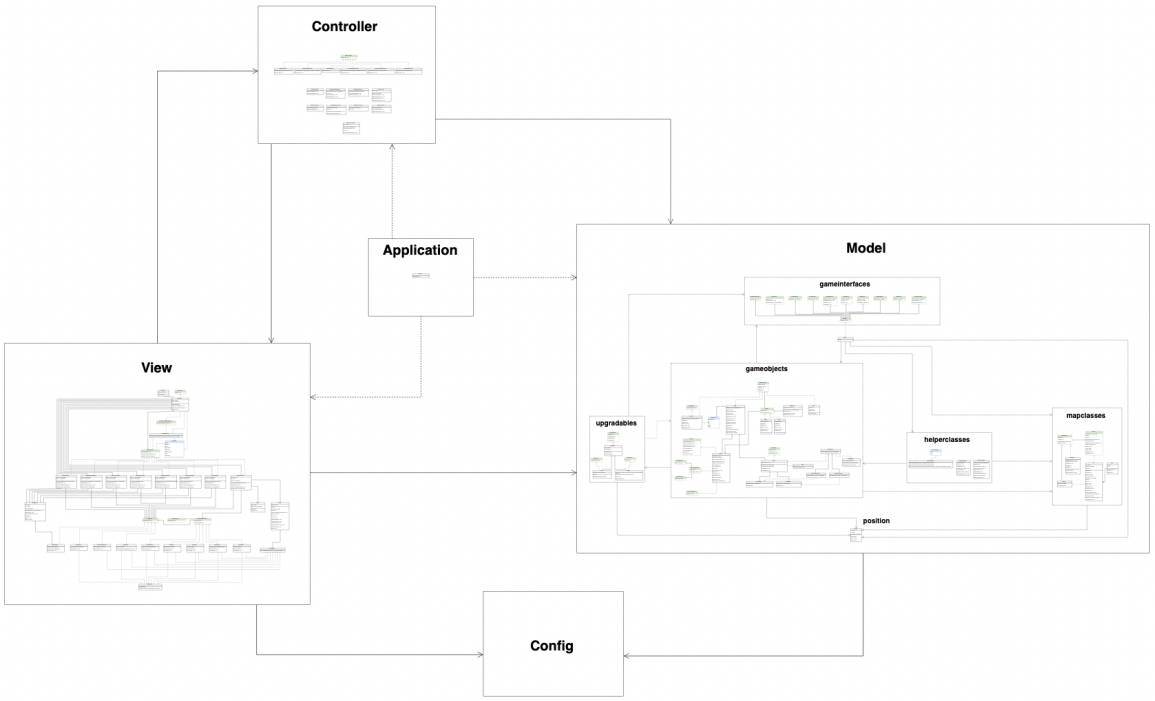
Known issues in the game are that the hitboxes are 48x48 units large, and since the player uses acceleration to move it is difficult to move between two non-passable tiles, which are one passable tile apart, without having another non-passable tile to line up the movement pixel-perfectly.

Since the player's hitbox is exactly the same size as the path between the non-passable tiles the user has to move pixel-perfectly to move successfully in these scenarios. Above this the pictures the view is using does not fully fill the whole 48x48 hitbox which leads to a visual issue where the player can collide with the unpassable tiles without the pictures directly touching each other, which can create visual confusion for the user.

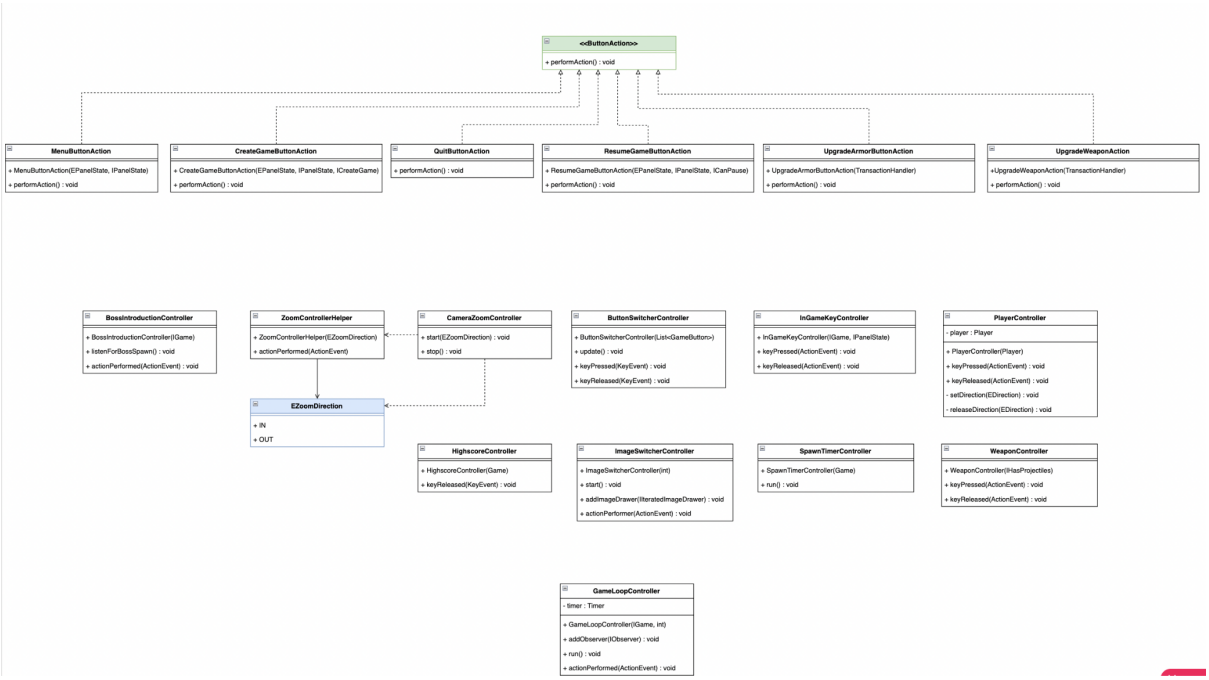
Appendix:

Top Level UML Diagram

[Link to UML (better quality)]

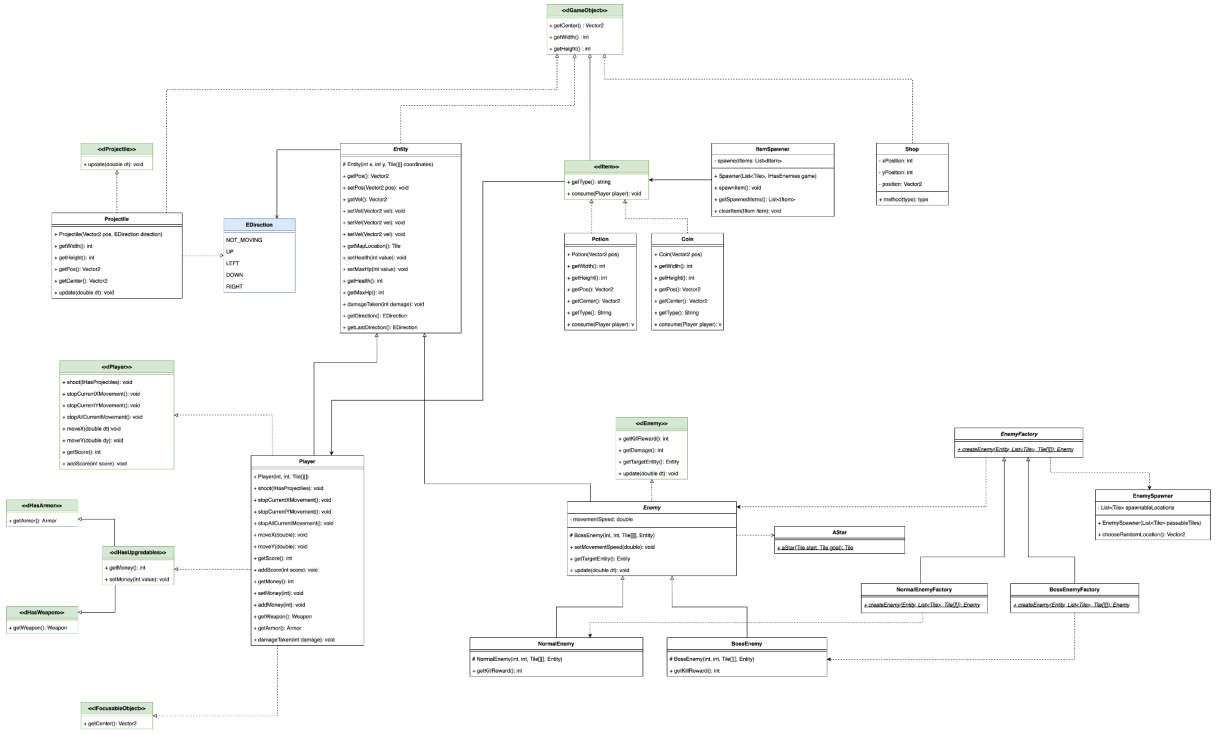

Controller
Application
Model
gameinterfaces
gameobjects
upgradables
helperclasses
mapclasses
position
View
Config

UML for Controller Package:

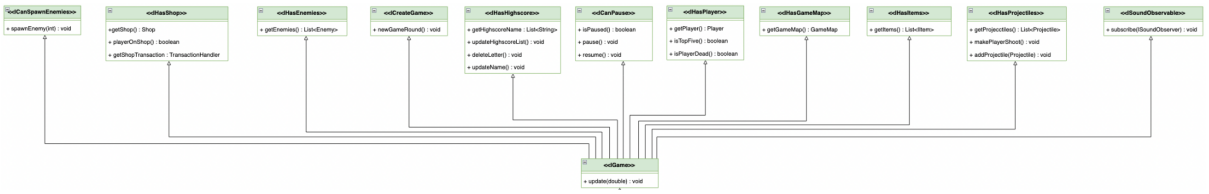


UML for different parts of Model Package:

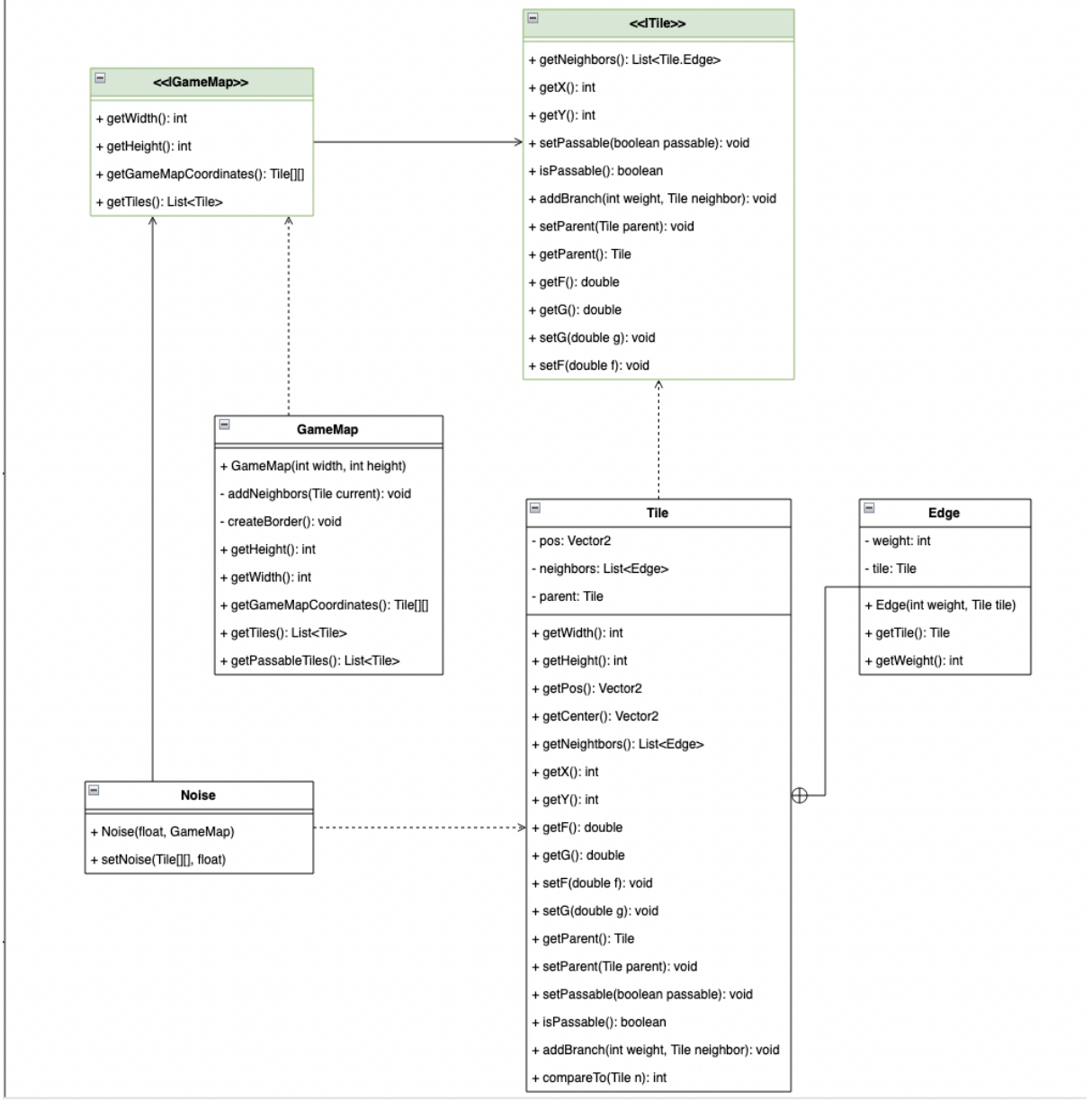
GameObjects



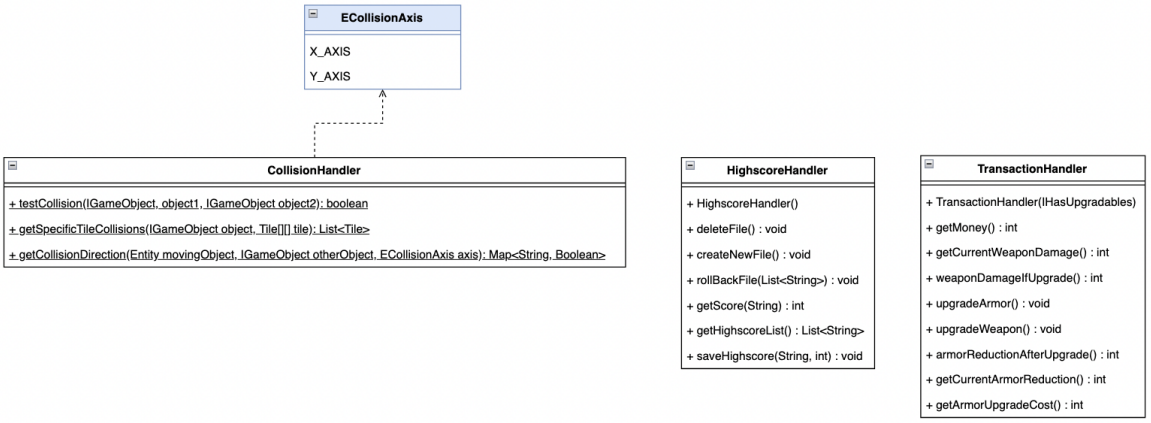
Game Interfaces



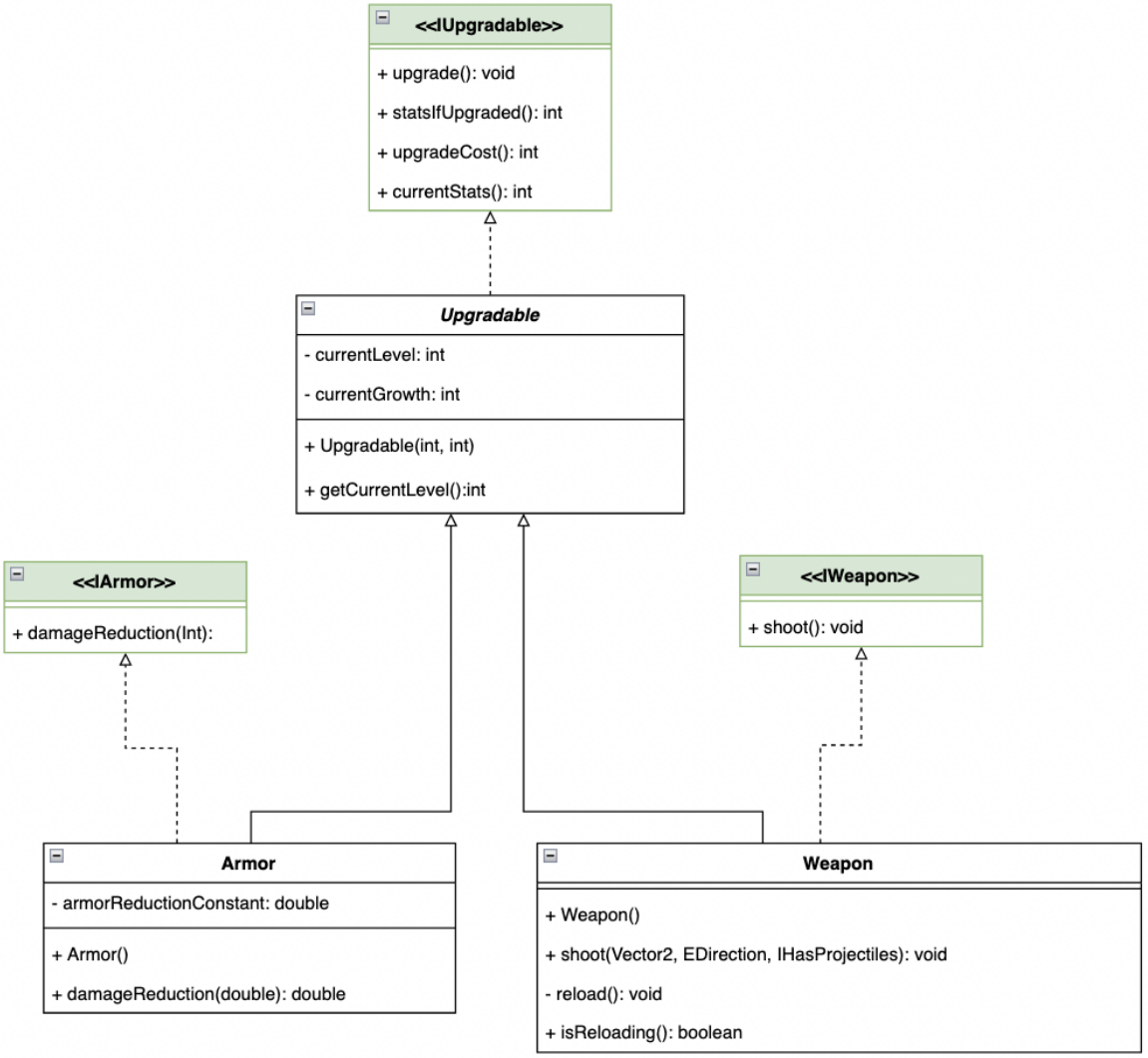
mapclasses



helperclasses

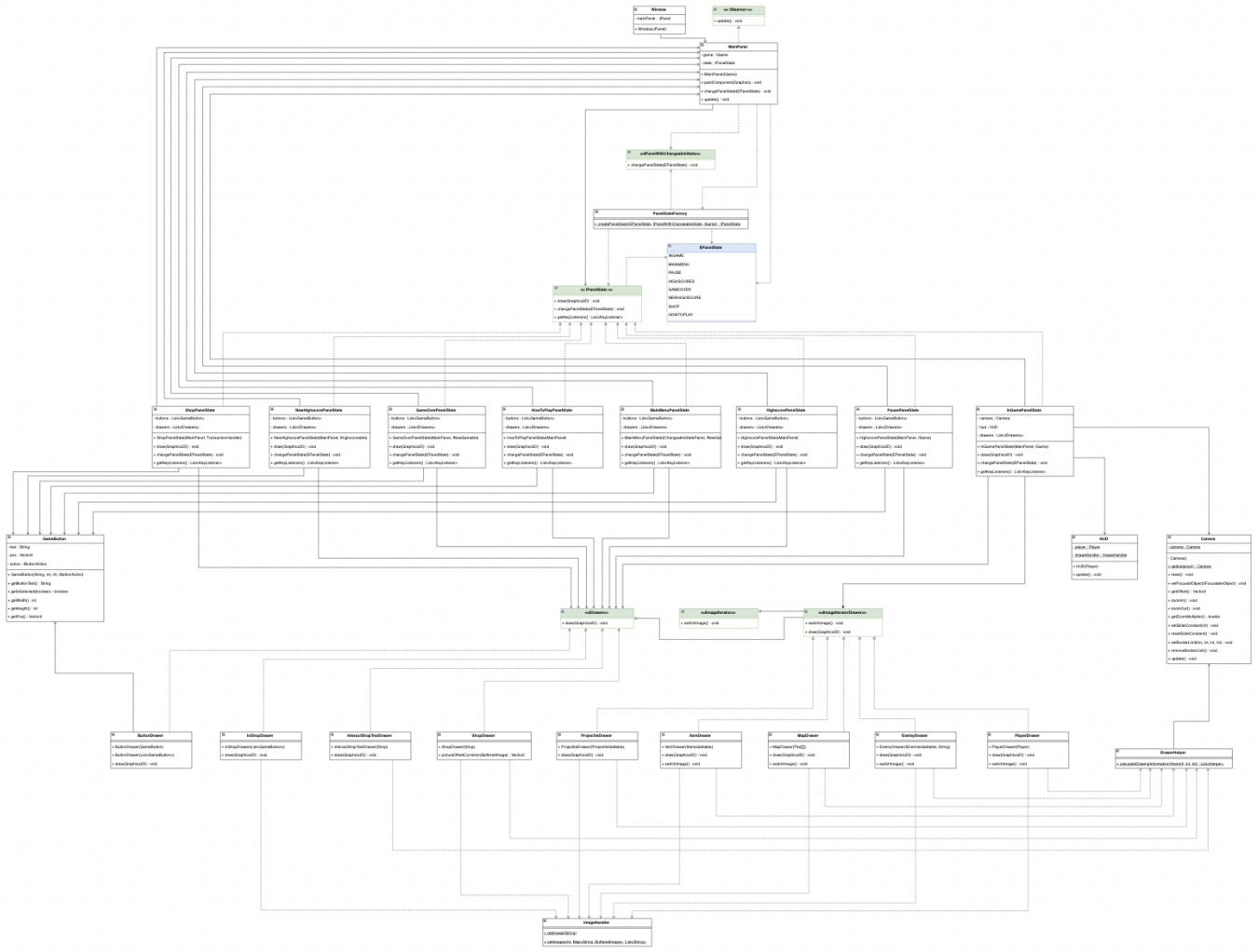


upgradables



UML for View Package:

View



References:

[1] Anonymous author, Wiki. 2022. *Boxhead (series)*. Fandom. Available at: https://boxhead.fandom.com/wiki/Boxhead_(series) [Accessed 3 October 2022].

[2] Kaalel (2022). *MVC Framework Introduction*. GeeksForGeeks. Available at: https://www.geeksforgeeks.org/mvc-framework-introduction/ (Accessed: October 3, 2022).

[3] *The 5th major version of the programmer-friendly testing framework for Java and the JVM*. 2022. *JUnit 5*. Available at: https://junit.org/junit5/ (Accessed: October 20, 2022).