System design document for Smurf Survivors

Isac Snecker, Isak Söderlind, Emil Svensjö,
Jakob Schült

2023-12-15

Version 1.0

1. Introduction **2**
1.1. System Introduction 2
1.2. Definitions, acronyms, and abbreviations 3
1.3. The program's functionality 3
2. System architecture **3**
2.1. The MVC-design Pattern 3
3. System design **4**
3.1 Model 4
3.1.1 Weapons Module 5
3.1.2 Entity Module 6
3.1.3 Handler Module 7
3.2 View 7
3.3 Controller 9
3.4 Other Design Patterns 9
3.4.1 Observer Pattern 9
3.4.2 Factory Pattern 10
3.4.3 Composite Pattern 12
3.4.4 Module Pattern 12
3.4.5 Dependency Injection Pattern 12
4. Quality **13**
4.1 Improvable parts of the program 13
4.2 Further development of the program 15
5. References **15**

1. Introduction

This document aims to show our program's design using class diagrams. It aims to show the connections and dependencies between classes/modules of our program and how we have done this using object oriented design principles and patterns.

1.1. System Introduction

The written program is an arcade action game called “Smurf Survivors”. The point of the project was to create an object oriented application, with the model written in java. Based on this requirement, the choice was to make an arcade action game based on the game “Vampire Survivors”, tentatively called “Smurf Survivors”. The program, which has been completely written in java, uses the framework libGDX for the graphical interface and the framework JUnit for the writing of unit tests. As the program is supposed to be object oriented, such principles/patterns have been used for the fundamental design of the program. Most notably the MVC-design pattern, with the architecture of the program being divided into distinct parts.

1.2. Definitions, acronyms, and abbreviations

- **Enemy**: An entity with the objective to inflict damage on the player
- **Player**: An entity in the game controlled by the user
- **Map**: The main stage of the game. Consists of a tile map in which entities roam upon.
- **XP**: Experience points given to the player after killing a enemy which levels up the player and unlocks access to new weapons
- **HP**: Health points related to the player. When HP reaches zero the game is over.
- **Weapon**: Object used to inflict damage on the enemies

1.3. The program’s functionality

When running the application the user is met with the main menu. Here they can choose to start the game, change the settings, or close the application. Pressing "Settings" opens another screen where the player can change the volume and the difficulty. Pressing "Start" results in the game being started. The player can move around by using either WASD or the arrow keys, which affect the direction their weapons are being fired. By killing enemies with these weapons the player will gain experience points. When the player gains a certain amount of “Experience Points” they will level up and gain upgrades to their current weapons as well as new weapons. There is also a pause screen where the player can choose to go into the settings or end the game, accessed by pressing “esc”. The objective of the game is to survive as long as possible and the time currently survived is shown at the top of the screen.

The game ends when the player runs out of “Health Points” which occurs when the player comes in contact with enemies.

2. System architecture

2.1. The MVC-design Pattern

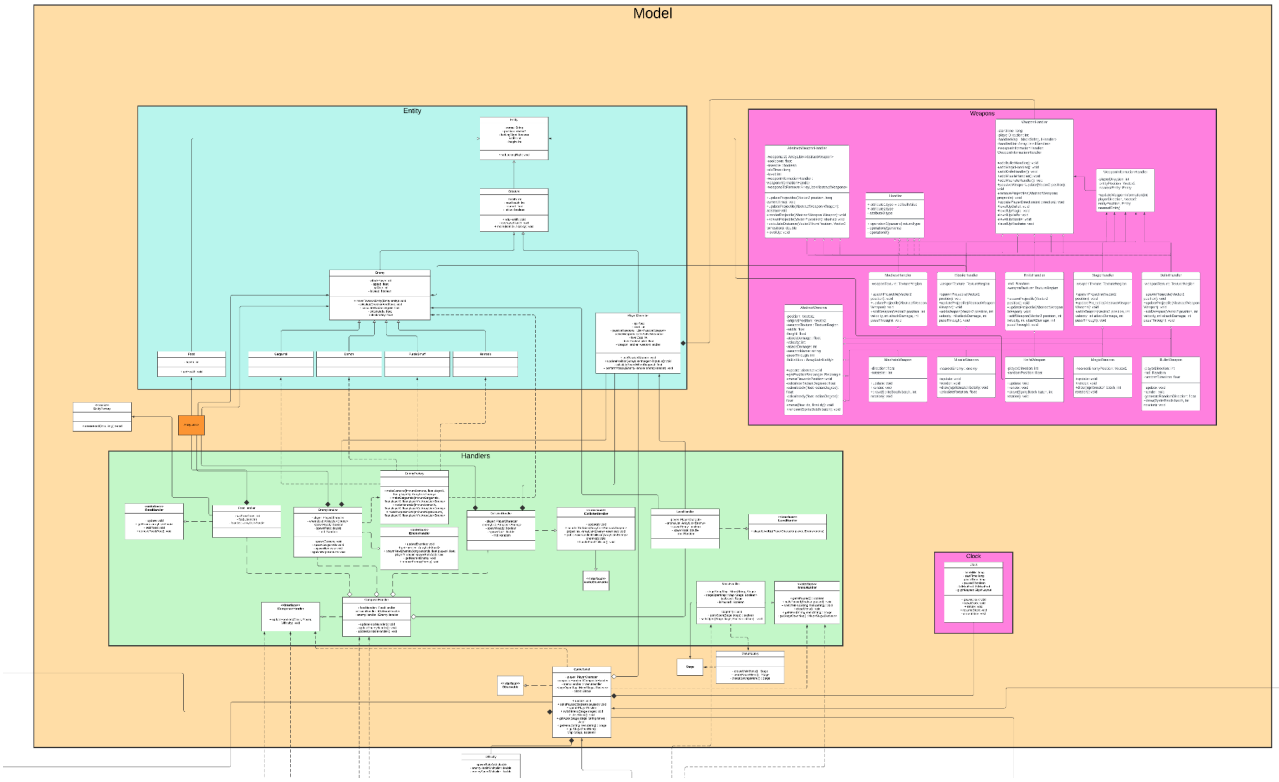
The basic structure of the program is designed using the MVC-design pattern. This means the program structure has been separated into three distinct parts. The model, which handles nearly all of the game’s data and calculations. The View, which handles the output of information to the user. And the controller, which handles the input of information from the user. In our program the model consists of the logic of the game, including the classes of the objects of the program. The view consists of the rendering of the map, menus and other objects. And the controller consists of the input related to buttons in the menus, and the movement of the character using the keyboard.
Use of the MVC-design pattern increases the possibility of reusing certain parts of the program.

3. System design

Here the design of the application is shown using UML-diagrams. We have designed our program according to the MVC design pattern. As such the description has been divided into these three distinct parts, 3.1 being the model, 3.2 the view, and 3.3 the controller. With each part having their packages as subparts. 3.4 and its subparts are reserved for other design patterns used.

3.1 Model

The model is responsible for the game's logic and is designed to be as reusable and extendable as possible. The logic of the model includes the classes for all of the games' entities (such as the player character as well as enemies), the game’s menus as well as the game as a whole. It also includes nearly all of the game's calculations and stored information. As the model is designed to be as



3.1.1 Weapons Module

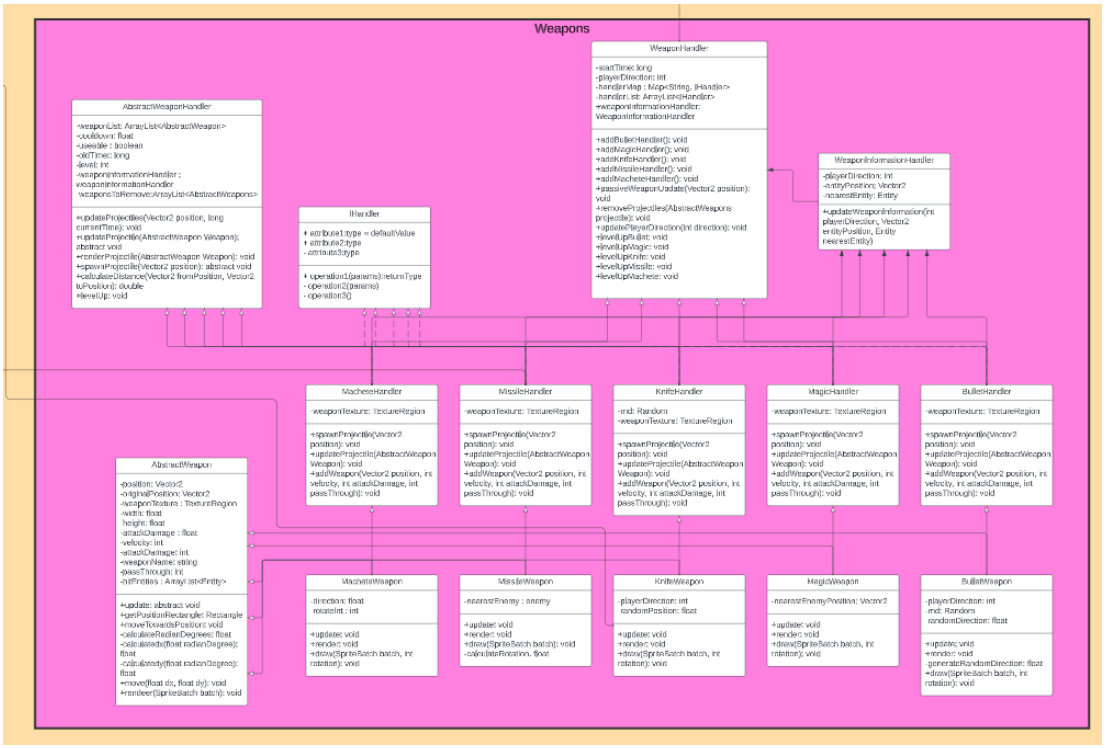
The weapons module is responsible for all things related to weapons. It only has dependencies on the Enemy and Entity classes of the Entity module. It is only depended on by the PlayerCharacter class of the Entity module.

OBS! Please note confusing wording. The "WeaponHandler"-class is not what is referred to when discussing weapon handlers. The ***weapon handlers*** *are the BulletHandler, MagicHandler etc. The* ***WeaponHandler*** *is the class called WeaponHandler*

We have a WeaponHandler which is a sort of facade for the other classes in the Weapons package. The player is given a WeaponHandler from which they can get weapons and fire them. When the WeaponHandler is tasked with giving the player a weapon, for example a "Bullet"-weapon, it runs the function "addBulletHandler()" which adds a BulletHandler to a list of handlers which the WeaponHandler has. The function for firing the weapons will then go through this list of handlers telling them to update themselves. The handlers themselves have a list of the weapons, or a better name for them would be projectiles. When the handler is told to update itself it will check if it's time to deploy a new projectile, time to remove a projectile, and for the remaining projectiles it will call

their update functions, which moves the projectile in a direction. The Handlers for the individual weapons are called “XHandler” for example “BulletHandler” as mentioned before. These handlers share an IHandler, which is mainly used as an interface for the WeaponHandler, as well as a superclass called AbstractWeaponHandler which contains all of the attributes and functions that the handlers share, such as the update function since the logic behind removing, adding and updating projectiles is the same between handlers. The same thing goes for the projectiles themselves, called “XWeapon” or “BulletWeapon” for example, have a superclass called AbstractWeapon which contains all of the attributes and methods which are shared between projectiles, an example of this can be width, height, damage and velocity.

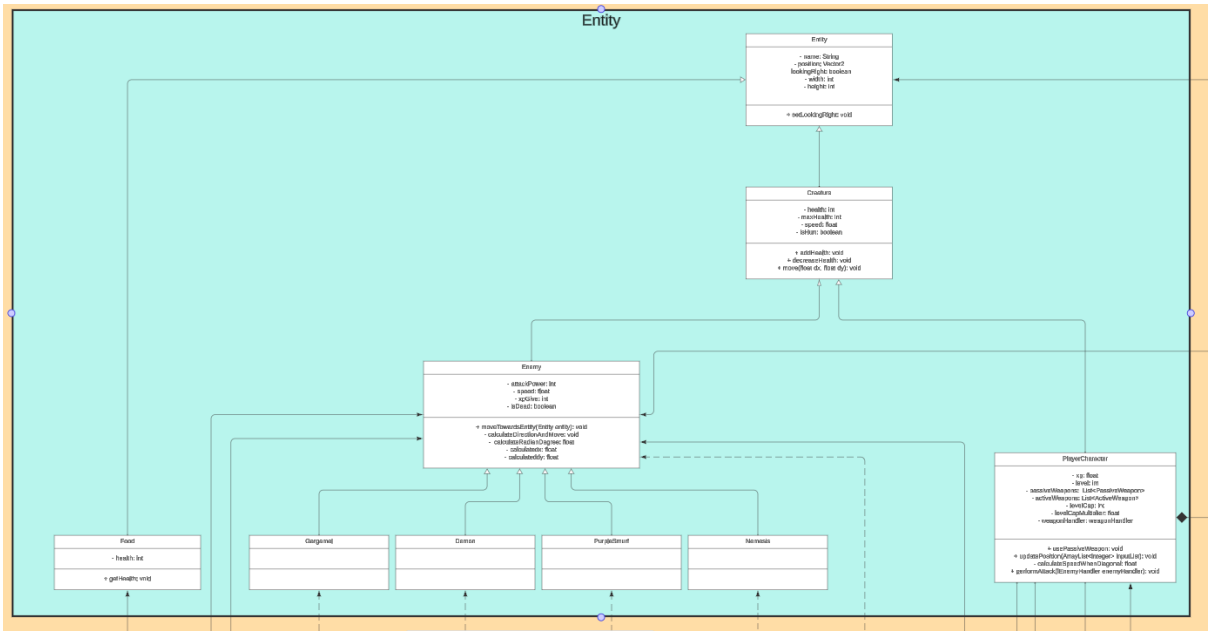
There is also a class for giving specific information to weapon handlers which is playerDirection, nearestEnemy as well as nearestEnemyPosition. This is because some weapons require information from the model to know where they’re supposed to go. PlayerDirection because they want to fire where the player is facing. NearestEnemy because the fire a projectile which follows the nearest enemy. NearestEnemyPosition is for weapons that fire at an enemy, but the projectile doesn’t follow them.



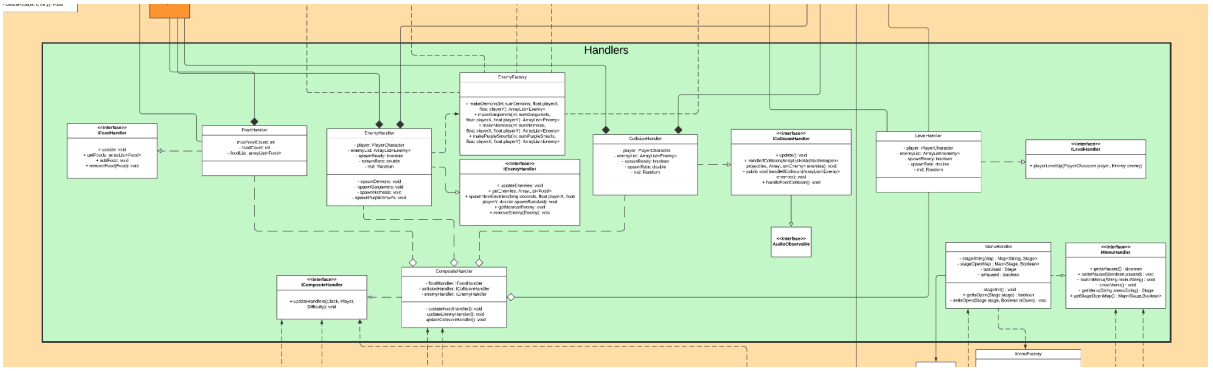
3.1.2 Entity Module

The entity module is responsible for all the models of classes with the abstract supertype Entity. These include the concrete classes PlayerCharacter, Food and different enemies. These classes have in common that they are all “entities” that are rendered onto the map in the game. With the abstract supertype Entity all instances of subtypes could be used as if they were an instance of the class Entity. Likewise all instances of subtypes of the abstract class Enemy can be used as if they were an instance of the Enemy class. Which is what is done in the EnemyHandler class in the Handler Module.

There all the game's enemies are added into a list of enemies, which is later iterated through, following the Liskov Substitution Principle.



3.1.3 Handler Module



The Handler module is responsible for updating logic and data related to different parts of the model.
For example, EnemyHandler is responsible for handling and modifying data related to the enemies of the game and FoodHandler is responsible for handling and modifying data related to Food in the game.

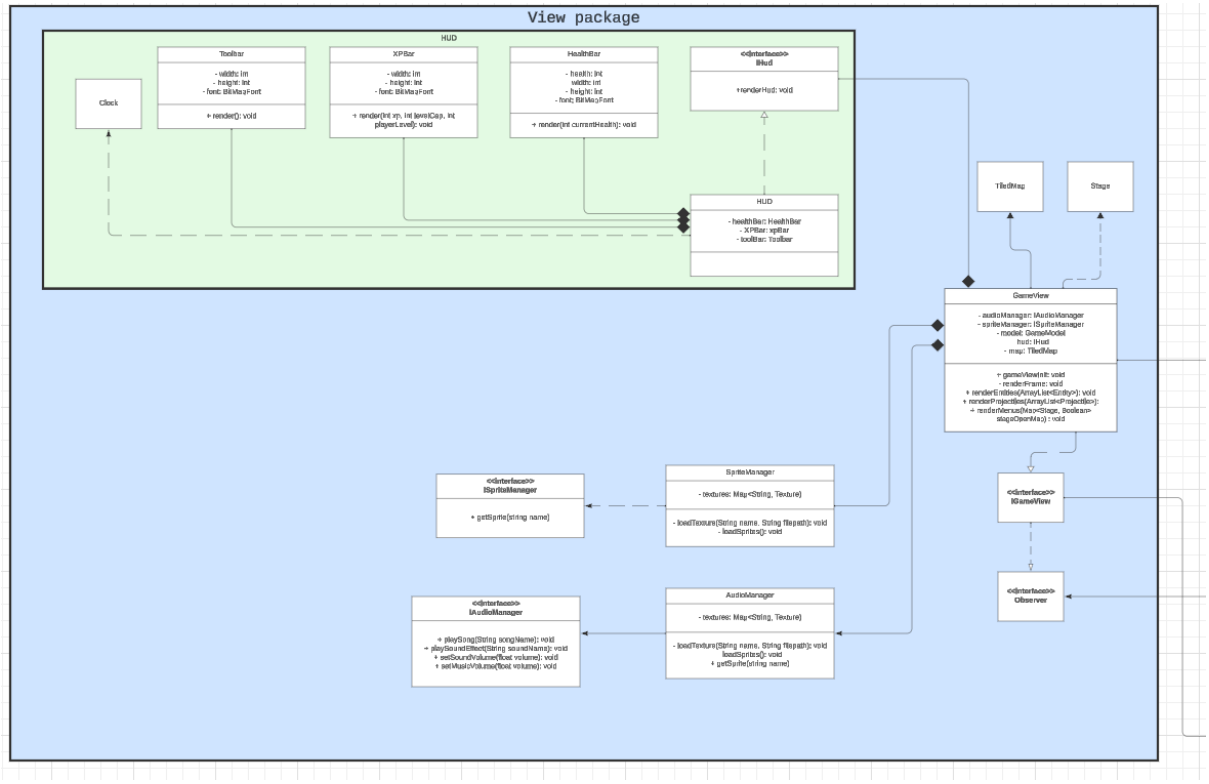
The way the handler module is implemented is useful in many ways. For example it increases separation of concern by separating the handling of different parts of logic into different classes.

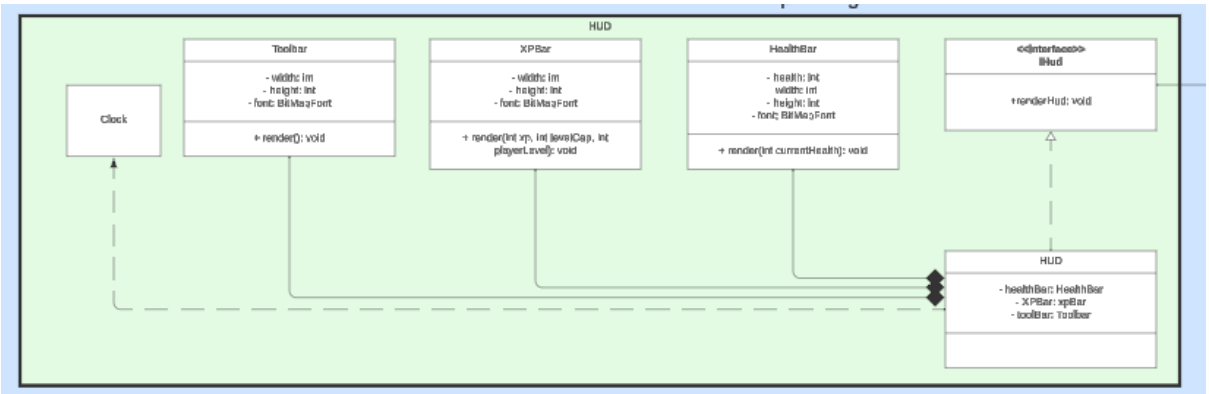
It also follows the single responsibility principle as if these handlers didn't exist and instead their methods and data were stored in the GameModel. The GameModel would have many more reasons to change thus breaking this principle.

3.2 View

The view is responsible for displaying information to the user based on the information it gets from the model. The view does not alter any information inside of the model, it only receives information from it. Information such as the player's, enemies', weapons' and other object's locations as well as the states of objects. When these values change the view is alerted through the use of the observer pattern. It then changes its rendering based on the information it gets from the model. In the program the view gets information from the model through an instance of the model as an attribute of the GameView class.

The view module also contains a HUD which is responsible for rendering HUD related information such as Health, XP, Toolbar and the clock.

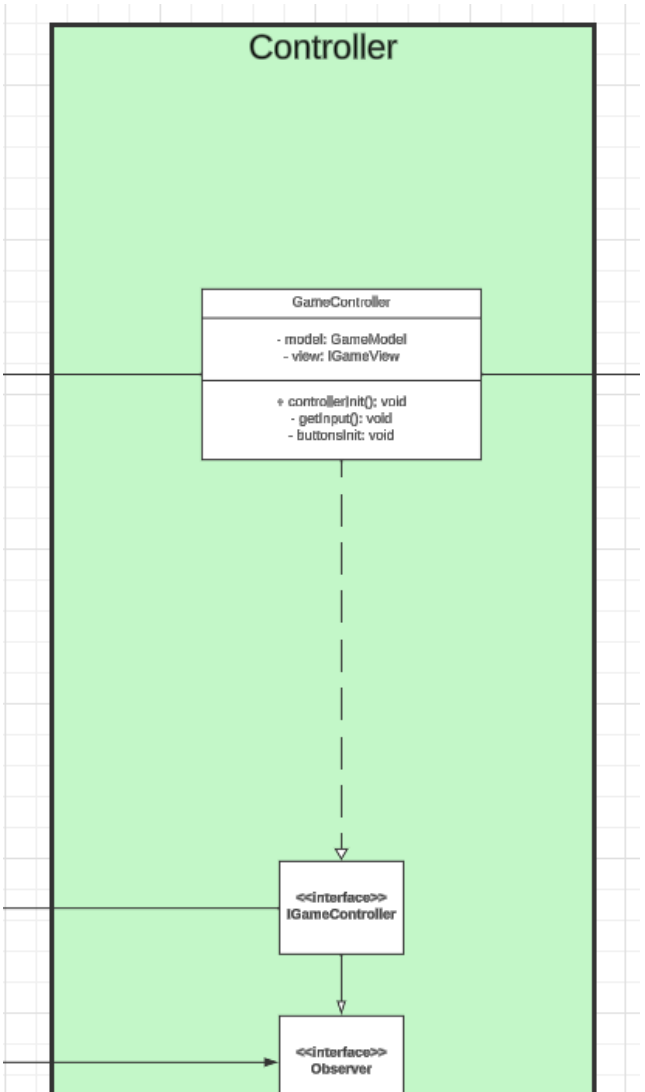




3.3 Controller

The controller is responsible for getting input from the user and directing the model based on this input. When the player presses a button on the screen or on the keyboard, the controller will take this input and based on it make the model calculate a specific function, or otherwise direct it in a specific way. The controller has been designed to contain as little “logic” as possible. Instead having needed calculations,etc. being calculated in the model. In the case of onscreen buttons, these are created in the model with their specific values, rendered in the view and connected to input in the controller.

The controller also implements the Observer interface through implementation of the IGameController interface. It does this in order to know when to listen for input. Everytime the model uses the notifyObservers method, the Controller listens for input.



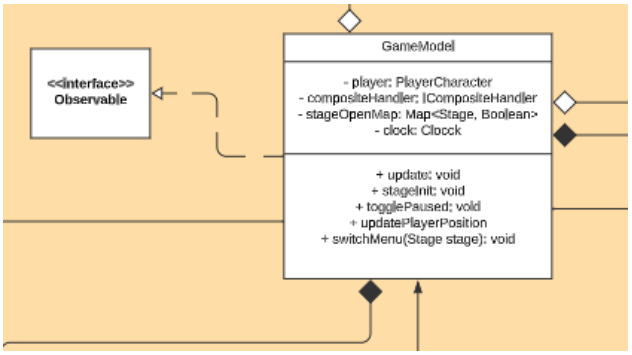
In the program the controller uses an instance of the GameModel class and an instance of the IGameView interface to manipulate the model and sometimes the view.

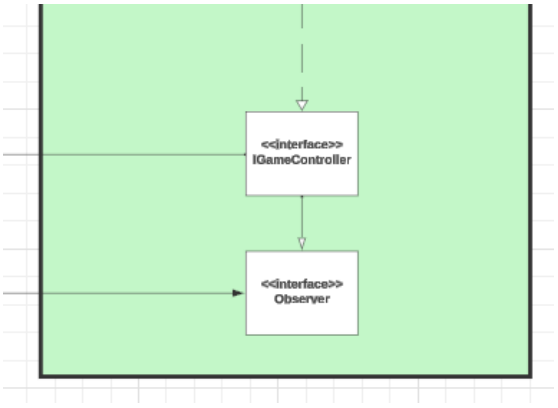
3.4 Other Design Patterns

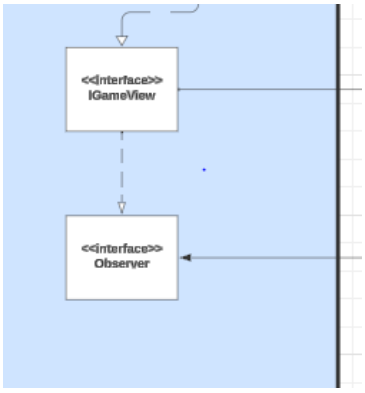
Aside from the MVC - design pattern we have also used many other design patterns in order to follow object oriented design principles.

3.4.1 Observer Pattern

The observer design pattern has been used. This was done in order to make the model be able to notify the view (and controller?) of updates, without having the model be directly dependent on the view. The observer design pattern was used by making an observer interface implemented by the view. The model has a list of observers to which the view is added. This list is continually iterated through and tells each observer to use their update method. This in turn results in the view taking and displaying the necessary information from the model.

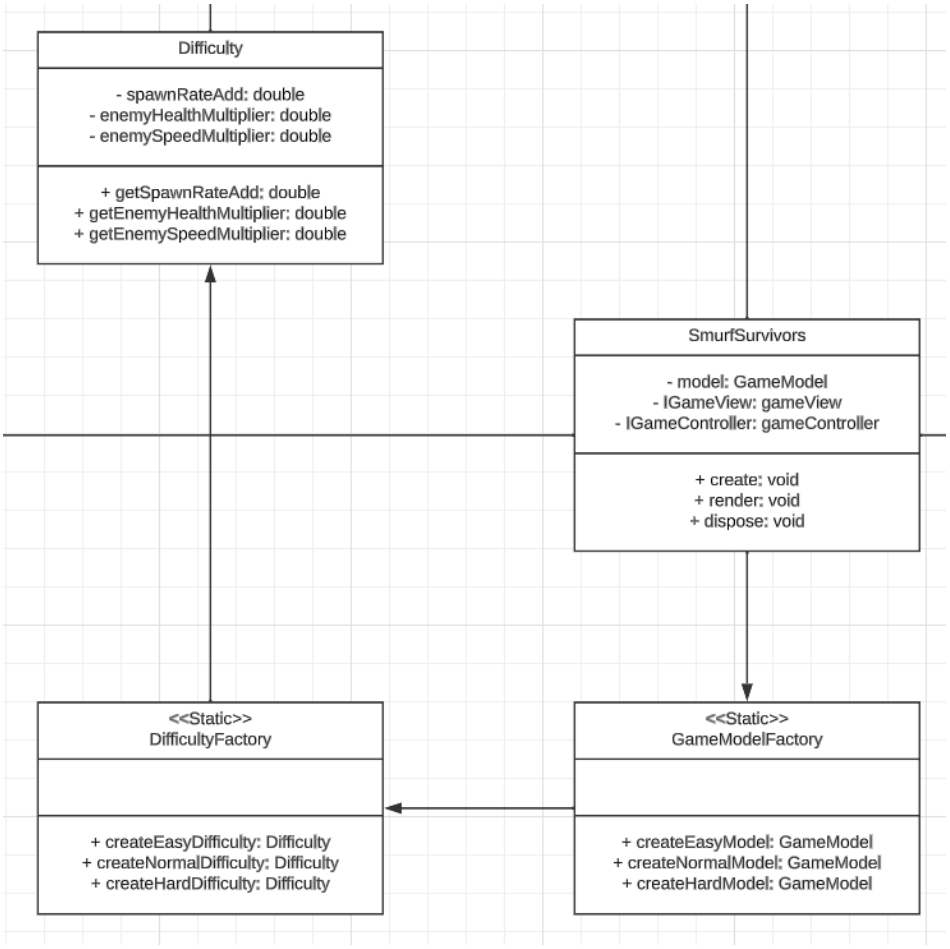


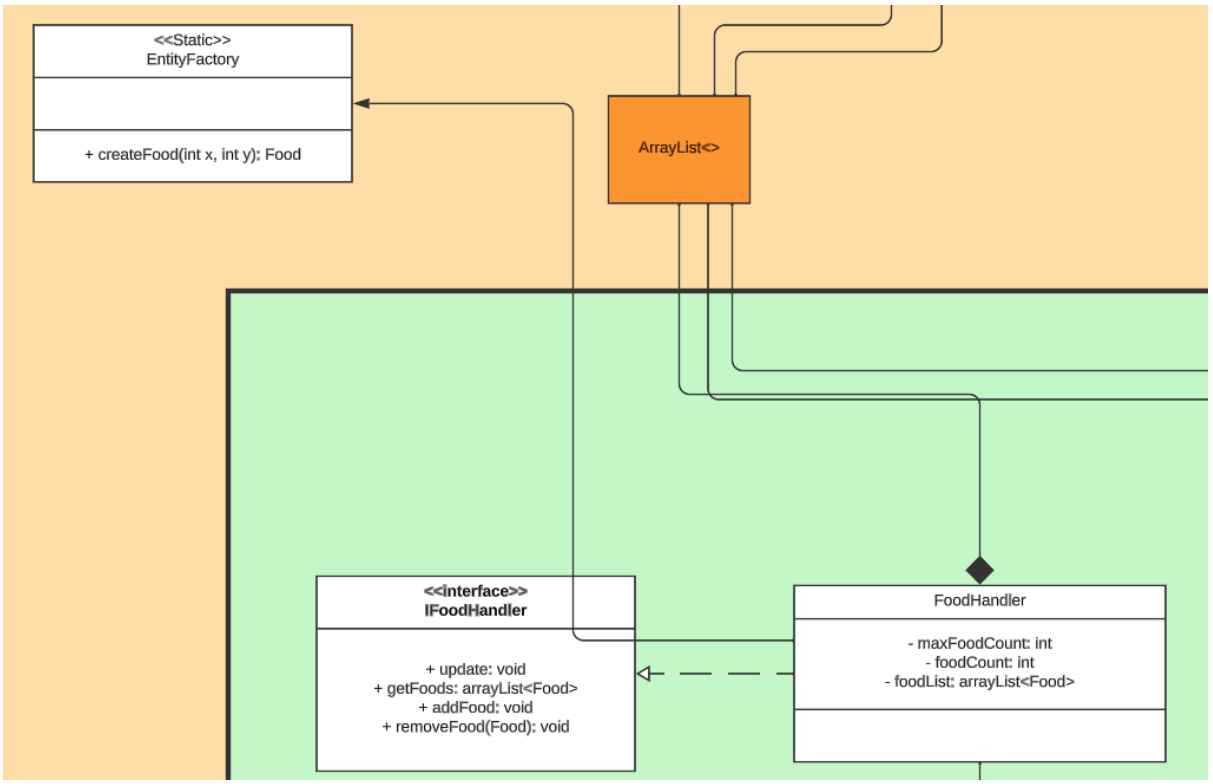


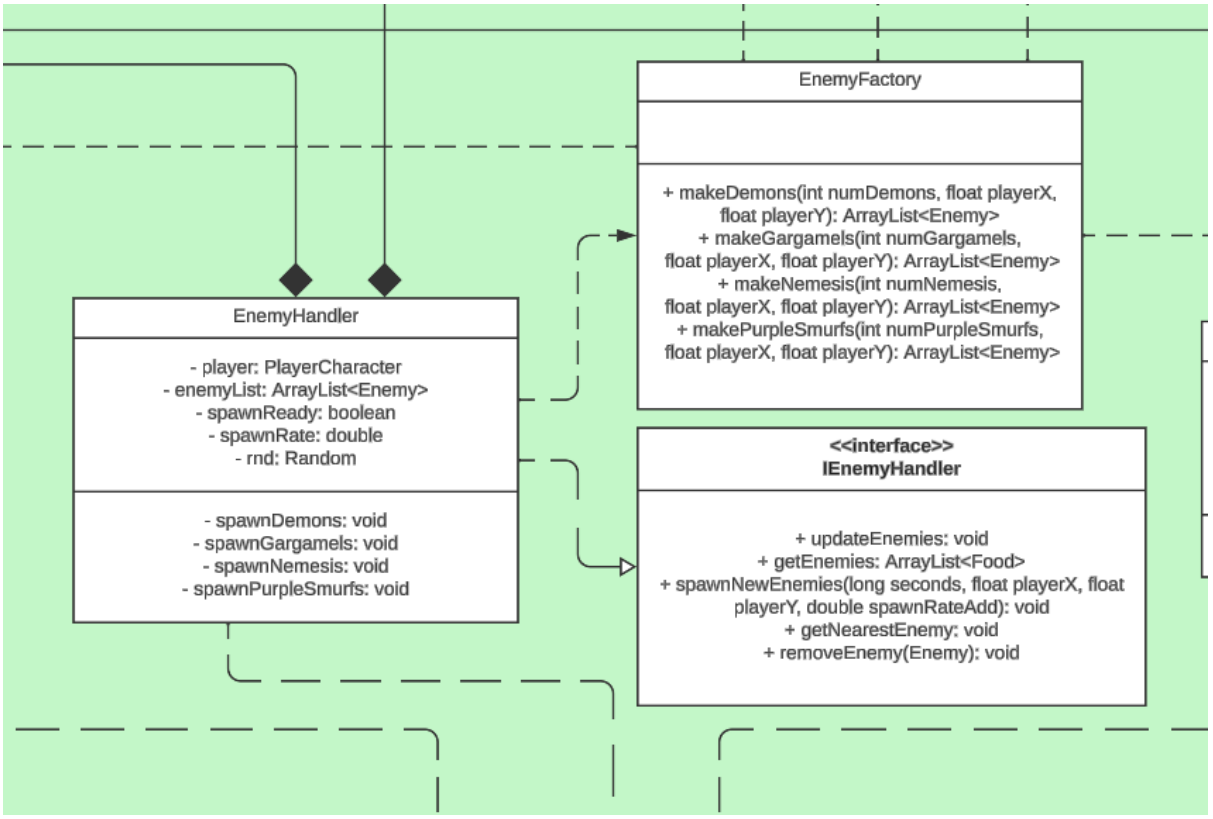


3.4.2 Factory Pattern

The Factory design pattern has been used. Most notably it has been used in the creation of the game's enemies. The factory design pattern is used to hide the internal implementation of an object. This by creating objects and returning them as a common interface. Which is exactly what the program does in the creation of enemies. When creating a specific type of enemy, the factory returns the enemy as an instance of the abstract superclass "Enemy".


Difficulty
- spawnRateAdd: double
- enemyHealthMultiplier: double
- enemySpeedMultiplier: double
+ getSpawnRateAdd: double
+ getEnemyHealthMultiplier: double
+ getEnemySpeedMultiplier: double
SmurfSurvivors
- model: GameModel
- IGameView: gameView
- IGameController: gameController
+ create: void
+ render: void
+ dispose: void
<<Static>>
DifficultyFactory
+ createEasyDifficulty: Difficulty
+ createNormalDifficulty: Difficulty
+ createHardDifficulty: Difficulty
<<Static>>
GameModelFactory
+ createEasyModel: GameModel
+ createNormalModel: GameModel
+ createHardModel: GameModel


<<Static>>
EntityFactory
+ createFood(int x, int y): Food
ArrayList<>
<<interface>>
IFoodHandler
+ update: void
+ getFoods: arrayList<Food>
+ addFood: void
+ removeFood(Food): void
FoodHandler
- maxFoodCount: int
- foodCount: int
- foodList: arrayList<Food>



3.4.3 Composite Pattern

The Composite Pattern is used among the different handlers of the program.
Instead of having the GameModel handle each handler separately within the update() method we have instead created a class named CompositeHandler whose purpose is to both store and update each handler. This is good in a few ways, firstly it greatly reduces the length of the models update method and in turn makes it more readable. Secondly it decreases dependencies between the handlers and the model as instead of many handlers being dependent on a class instead compositeHandler will be dependent and pass this along to each sub handler.

3.4.4 Module Pattern

The module design pattern was used in order to separate the program into modules of related classes and interfaces. The classes of the MVC have been sorted into the modules Model, View and Controller. The classes that are conceptually weapons have been placed into a common module. Same for classes that are conceptually entities.

3.4.5 Dependency Injection Pattern

The dependency injection pattern is used to decrease a class' dependency on specific objects by creating them outside of the class and then "injecting" them into the class. Which is what the program does when creating the player character. In the ModelFactory class the player character is

created with specific parameters and then used as a parameter for the constructor of the GameModel. As such the GameModel isn't dependent on a specific implementation of the PlayerCharacter class.

```
no usages  IsacSnecker
public static GameModel createHardModel() {

    PlayerCharacter player = new PlayerCharacter( health: 100,  x: 16000,  y: 16000,  width: 90,  height: 90,  speed: 5,  direction: 0);
    GameModel model = new GameModel(DifficultyFactory.createHardDifficulty(), player);

    return DependencyInject(model);
}



3 usages  IsacSnecker +1
private static GameModel DependencyInject(GameModel model) {

    // Dependency injection by setter method
    IFoodHandler foodHandler = new FoodHandler( maxFoodCount: 500);
    IEnemyHandler enemyHandler = new EnemyHandler(model);
    ICollisionHandler collisionHandler = new CollisionHandler(enemyHandler,foodHandler, model.getPlayer(), model.gePlayertLevelHandler());
    ICompositeHandler compositeHandler = new CompositeHandler(foodHandler, enemyHandler, collisionHandler);
    model.setCompositeHandler(compositeHandler);

    return model;
}
```

4. Quality

4.1 Improvable parts of the program

1. **Audiomanager**
   The Audiomanager, a part of the view module, handles values related to the game's audio. As such the controller has to call on the view to change values for the game's audio. This is the only thing the instance variable view in the controller is used for. Instead these values could be implemented into the model, removing the controller's dependency on the view.

2. **Stages**
   The Stage class of the libGDX library is used for the construction of menus in the game. This however has led to a sort of blending of the model and the view in the program. This due to the fact that the Stage class both handles certain values and calculations as well as has a render method. In further development of the program the Stage class would be replaced in order to write a render function for menus separate from the menus' model.

3. **Weapons**
   Similar to the menus having render methods defined in the model, the weapons of the program also have render methods defined in the model. Instead the rendering of weapons should be defined in the view, with sprites getable from the SpriteManager class. Furthermore the program's weapon projectiles should be defined as subtypes of the Entity class, since projectiles have the functionality of the Entity class. This would then lead to the projectiles being able to be rendered by the same function as other "entities" in the view.

Many of the weapon classes have very confusing names. The WeaponHandler class and the "XHandler" for example the BulletHandler or MagicHandler are completely separate, however it gets rather confusing discussing them when they're both called handlers. The abstract class called AbstractWeaponHandler isn't even implemented by the WeaponHandler! Instead it is implemented by the "XHandlers". An improvement would be naming the WeaponHandler class something in the realm of WeaponFacade. The same thing goes for the classes which are the weapons projectiles. When a bullet is shot, an instance of the class "BulletWeapon" is created, however why is this class called BulletWeapon when it isn't a weapon but rather a projectile. An improvement would be to call all of the "XWeapon" classes "XProjectile" instead.

Weapons have dependencies on the Entity-class. This could be problematic if the Entity class is rebuilt someway, since if you happen to mess with the information that the weapons rely on, you all of a sudden get problems in the Weapons-package when you're rebuilding entities! A fix for this would be to somehow make homing weapons get updated positions of an enemy whilst not having to know who that enemy is. Perhaps the model-class could keep track of the enemy?

Currently the projectiles have different functions for how they move, and instead of implementing these different move functions through a strategy pattern they're implemented in the weapons superclass AbstractWeapons. This means that weapons share movement functions that aren't used universally, which could be problematic and confusing from a code design perspective. A fix for this is discussed in **4.2.1**.

The way you add a weapon to a player is by running a method in their WeaponHandler called "addXHandler()", for example "addBulletHandler()". This way of adding weapons is a bit problematic since it means for each new weapon you create, you need to create a method in the WeaponHandler for adding said weapon to the armory. Instead you could use a method called "addWeaponHandler(IHandler handler)" where you input the class of the weapon you want to add, for example "addWeaponHandler(new BulletHandler())". This avoids having to add methods in the WeaponHandler when creating new weapons. However it's also more confusing when it comes to adding weapons. Say one person works on the weapons and another person works on the levels. The person working on levels want to give the player a knife whey reach the fifth level. Now since the person working on levels don't know about the internal implementation of weapons, it gets rather confusing for them to use "addWeaponHandler(IHandler handler)" since they most likely have no clue what the parameter "handler" is. However being approached with the "addKnifeHandler()" function is quite a bit more straightforward for the poor soul working on levels. Since there are both benefits and drawbacks with both of the ways, we are torn on what to implement.

The Magic weapon is currently broken. The weapon fires at the nearest enemies position, however if this enemy has moved from their position or is dead the projectile will simply stop at said position. This leaves a lot of magic projectiles floating around in the world. A fix for this would be to improve the algorithm for removing projectiles or changing the way the

projectile moves. If the projectile could aim at an enemies position however keep going beyond said position the problem would be fixed since currently projectiles get removed after moving a certain distance, therefore if the projectile never stops, it will eventually have traveled the distance needed to be removed.

4. **CollisionHandler**
   The CollisionHandler stops the weapons from being expandable. Currently the CollisionHandler only checks if the players projectiles are colliding with the enemies. This means that if we give an enemy weapons their projectiles won't be checked if they collide with the player. A fix for this would be for the CollisionHandler to have a list of WeaponHandlers which it goes through to check if any collision is occurring. For this to work you would also have to implement a “target” in the WeaponHandler. Such that when a WeaponHandler is created you insert a class as the “target”, for example the player would have a WeaponHandler which has the class “Enemy” as their target, whilst enemies would have the class “PlayerCharacter” as their target. The CollisionHandler could then go through all the WeaponHandlers from both players and enemies and check collisions between their respective projectiles and target.

4.2 Further development of the program

4.2.1 Strategy Pattern for enemies and weapons

Both the enemies and the weapons could benefit from a strategy pattern. Currently all of the enemies simply move directly towards the player, say we want to add an enemy which moves a bit, then stops and shoots at the player, we would then need to change that enemies movement function. Same thing goes for the weapons projectiles, some of them move where the player is facing, some move towards enemies, some go in circles. Since some enemies and weapons would like to share movement functions, it would be beneficial to have a class implementing different movement strategies that both enemies and weapons could composite any function they want. This avoids writing code twice and offers an easily understandable way of communicating how the enemy or projectile will move.

4.2.2 Toolbar

It is planned for the toolbar to store unlocked weapons and indicate their level with a label. This would let the player be more easily able to see what weapons they currently have, instead of trying to decipher the projectiles they’re shooting. This has however not yet been implemented.

4.2.3 Upgrades

It is planned to allow the player to choose which weapons to unlock or upgrade during level up. Out current implementation allows for weapons to upgrade in a fixed, hardcoded manner. The idea is to instead allow the player to choose their own upgrades and weapons. Since the upgrades they can choose from would be random, this would lead to the game being a bit less repetitive since the player would probably get new upgrades each time they play. It also allows for the game to feel more customizable and personalized. This has however not been implemented at this moment.

5. References

LibGDX: Library used for rendering the game.
JUnit: Library used for testing methods in codebase.
Mockito: Library used to mock some classes. Required for some JUnit tests.