System design document for Alien Run

Group 28

**Atosa Daneshvar-Minabi, Alexander Gilabert,
Ann Heijkenskjöld and Isak Wideskott**

Objektorienterat Programmeringsprojekt TDA376
Chalmers tekniska högskola

October 24, 2022

Contents

1 Introduction

The purpose of this document is to give an understanding of the overall structure and design of the project Alien Run. It will also explain the reasons behind those design choices.

This project, which is a desktop application, is a side-scroller platformer game where an Alien runs in space while shooting and jumping over obstacles, inspired by the Swedish game *Geometry Dash*. A player runs to the right of the screen at a consistent speed while the user has to jump to avoid obstacles. There are different variations of the player which change the feel of the physics of the world and the speed of the player. According to which mode the user chooses to play there are different variations and combinations of gems and obstacles in the game. The purpose of the game is to survive as long as possible while collecting as many gems as possible.

1.1 Definitions, acronyms, and abbreviations

- Desktop application - A software program that runs stand-alone on the desktop, instead of e.g. from a browser.
- Side-scroller - A side scrolling game, generally in 2D that scrolls from left to right or the other way round.
- Platformer - A game in which a player can run and jump on a platform.
- GUI - Graphical User Interface.
- JUnit - A unit testing framework for Java.
- JavaDoc - A tool used to write standardized documentation for Java projects.
- UML-Diagram - Unified Modeling Language Diagram, a way to visualize a software system.
- MVC - Model View Controller, a software design pattern to structure a program.
- Trello - A web tool that helps visualize and plan projects.
- LibGDX - A Java development framework used to code games.
- Figma - A tool used to design GUIs.
- Sprite - An image that represents a game asset.

2 System architecture

The project is built with an MVC (Model-View-Controller) architecture. All the logic of the game is in the Model package and this contains all the data and the game logic. One would be able to swap this Model out for any View with another library than the one we used, perhaps by the addition of adapters in some cases. Theoretically, one would also be able to play this game completely in the terminal with print messages updating the user of the state of the game. There are no outside dependencies to other libraries in the Model with the exception of the ArrayList class from Java in the Model, which makes this possible.

The GUI part of this project can be found in the View package. The application is built using the framework LibGDX which is used to render and present the Model with the help of different screens, sprites and text to the user.

The Controller is responsible for all the input handling from the user, and communicates this to the Model.

2.1 Application Flow

Because the project is built with the LibGDX framework, the entry point of the application needs to be started via LibGDX main program. It happens in the class DesktopLauncher. From there on the game will initialize its window in the View part via the class Grupp28GDX which extends the frameworks ApplicationAdapter. The starting screen state will show up, the Controller will take input from the user to start the game which happens in the MenuInputHandler and from here on the Model decides what the game will be like and sends signals to View to render what's happening in the Model.

If the player instead decides to click on Instructions in the menu the MenuInputHandler will process this and directly change the state which happens in the View. The third option is if the player clicks Exit which will stop the whole application via LibGDX.

When the game starts the player is taken to the ChooseDifficultyState class in the View, what the user chooses to click here will be the first manipulation of the data in the Model. Firstly the View is updated via the GameStateManager to PlayState according to the Mode chosen, which will determine what the player, obstacles and gems will look like. The Model will be updated via the PlayInputHandler to decide the game mode. From there on the Model has full control of what happens in the game through an update method which updates every 1/60 of a second. The game model is responsible for everything that happens in the game from the physics to the collision detection, and the World class takes care of all of this. The score and state of the game is communicated to the user in the Hud class of the View. When the player touches an obstacle the game is over, a game over stage is shown to the user in the

View via the Hud class while the update method in the Model has stopped. From here on the Controller takes input from the user, if the player clicks on key ”m” on the keyboard the whole process will start over again from the beginning.

3 System design

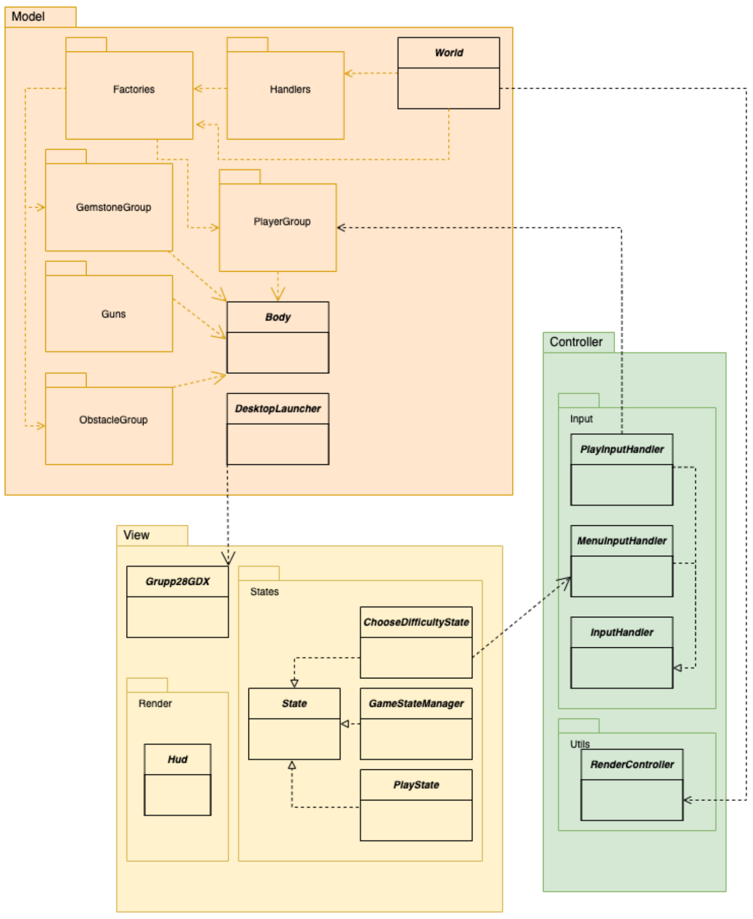


Figure 1: Top level view of the project UML.

In this project the Model View Controller architecture works in a way which lets the model send data to the View, the View renders updates to the User, the User sends input to the Controller which in turn updates the model. It is a simple and classic MVC architechture that has been opted for. The decision to create the project like this is mainly because it makes the model very independent, which lets the developer easily extend the project according to the Open Closed Principle. The classes inside the packages in the Model, View and Controller with interface dependencies between them also creates a high abstraction which further lets the developer easily extend upon the project. The packages inside the packages were created for increased readability.

The Model is created in a way without any libraries or outside dependencies which lets you switch it out for any kind of View. It updates the game through a method which belongs to the Player, this method is then used in the World class which updates the whole game. This information is then sent from World to PlayState in View where all other objects as well as the player will be shown to the user. Because the game has an update method in the model it is possible to build the game to be played entirely without a view. In a previous iteration of the game it used a physics engine which created some dependencies to the logic and hence the Model of the game but now the physics and collision detection is built without a physics engine which makes the Model completely independent and switchable to any kind of GUI.

The game doesn't have a typical game loop because the framework LibGDX that is used is event-based, simply put it updates the game every time when something happens. The model was built in a similar way which updates every time the player moves, and the player has the effects of gravity working upon it all the time. There is a World class that is based around the Players movement. The World moves forward because the player moves forward, the World appears to have gravity because the player has gravity. All other objects just spawn and disappear which the World also takes care of. The dependencies from the Model to View only goes from Player and World in the Model to the PlayState class in View. The View is responsible for all updating of the sprites and the graphics of the objects in the World. This choice was made because the program is made more efficient if sprites disappear from the program when they are out of sight of the screen.

The Controller is also dependent on LibGDX, it uses its InputProcessor class. The controllers only job is to take input and send output which makes this package very thin. It handles Model output through the different handlers. Below you will find a more detailed UML of the whole project. Note that the View and Controller has some dependencies to LibGDX but they are not pictured in the UML. This is also the case with the class DesktopLauncher, this is necessary for the project to work at all because it was built with the LibGDX framework.

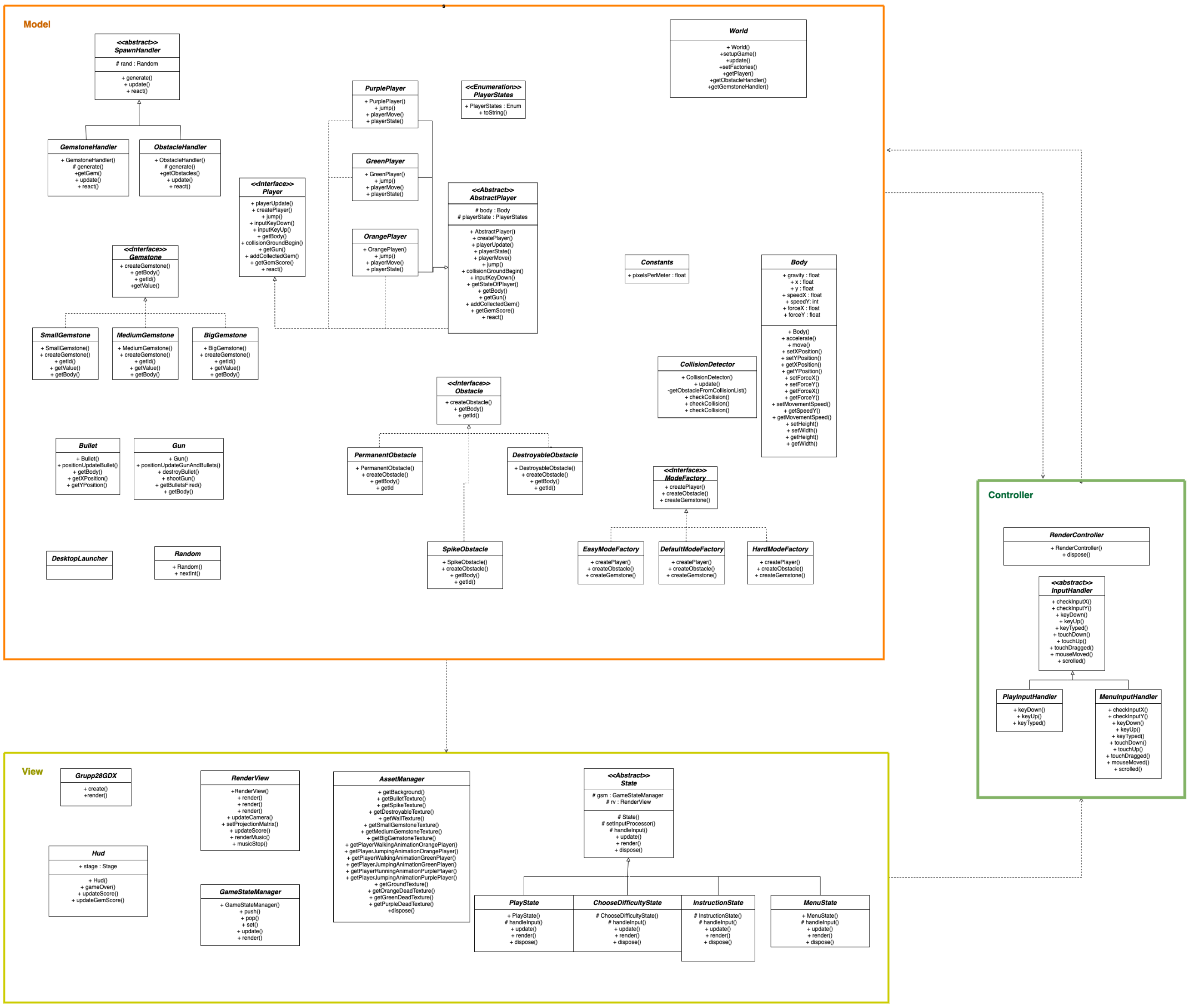


Figure 2: Detailed view of the project UML.

An even more detailed view of the Model looks like this.

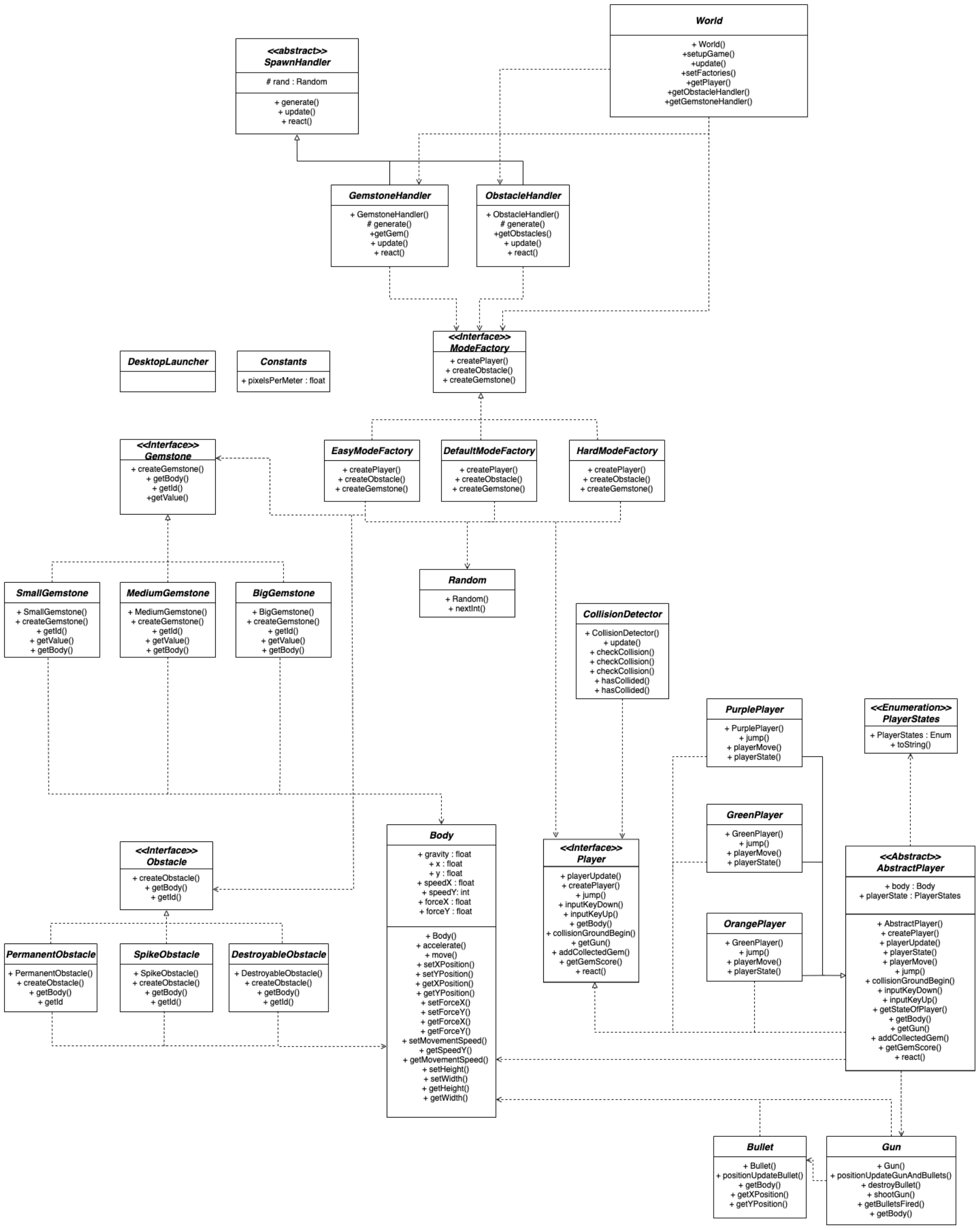


Figure 3: UML of the Model.

The Model was built around the Domain Model and they were both updated as the project progressed, to stay in sync as more features was added to the project. The Domain Model looks like this, by comparison.

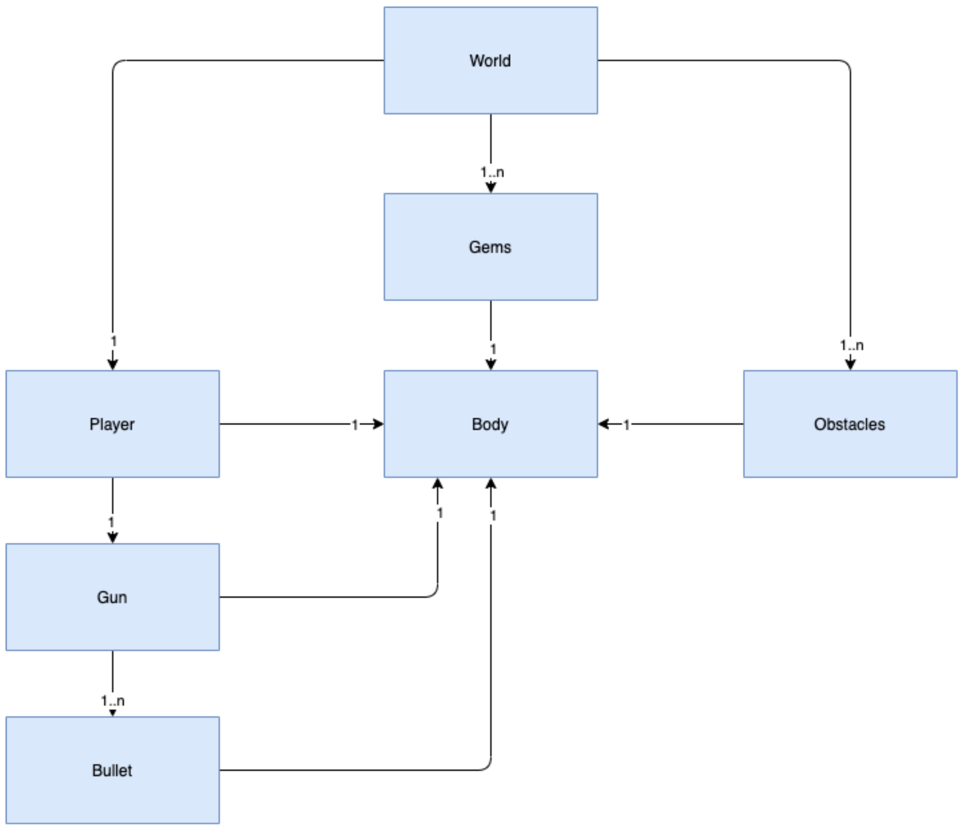


Figure 4: The domain model.

By looking at the two UML-diagrams it is clear that the Design Model follows the Domain Model. All dependencies in the Domain Model can also be found in the Design Model. E.g. all obstacles, players and gemstones have a body. The player has a gun and the gun has a bullet, which both have a body as well.

3.1 Design Patterns

The architectural pattern, as mentioned, is a classical MVC for the whole project. Additional design patterns which were used follows:

- Abstract Factory Pattern
- Template Method
- Observer
- Singleton

Abstract Factory Pattern

At the top of the Design Model UML-Diagram there is a ModeFactory interface. This is the Abstract Factory. Its concrete factories are EasyModeFactory, DefaultModeFactory and HardModeFactory. These are the families of related classes, Easy, Hard and Default. These three different versions can be found in a player instance, obstacle instance or in a gem instance. So e.g. there is a hard gem, default player and easy obstacle. These are all mixed in the game depending on which mode the user chooses to play. The concrete factories can produce all of them. This design pattern was chosen because it abides by Liskovs Substitution Principle, allows for high abstraction and follows the Open Closed Principle. This all makes the code simple to build upon, if a developer wants to make an Extremely Hard player, gem and obstacle or even a whole game mode they can just implement another concrete factory, and their given products of course.

Template Method

At the right of the UML-diagram there is a cluster of different players. These are all subclasses of the abstract class AbstractPlayer. This is a use of the template method, in which the superclass implements all common functionality for the player and the subclasses fill in the gaps for where they differ. This is the only place in the codebase where this pattern is used. The purpose of it was to follow the DRY principle, and it was implemented because there indeed was a lot of code duplication in these classes. The same problem can be found in the clusters of obstacle and gems. The reason the template method was not implemented here was simply because of lack of time. The design pattern was even reluctantly used here because the first choice would have been to use the Strategy Pattern. The reason it was not opted for was again because of lack of time. Why Strategy Pattern was the first choice here was simply because it follows Liskovs Substitution Principle better and the goal of the codebase was to have as high abstraction as possible, together with low coupling and high adhesion.

Observer

A crucial part in managing the logic in the game is the collision detector. This class is made using the observer pattern. the collision detector has subscribers which can be any object. In this case we add the obstacleHandler, gemstoneHandler, gun, player and hud as subscribers and during every call on the update()-method, the collisions between obstacles, bullets, and player is checked. If a collision is detected the involved subscribers will be notified of said collision.

Singleton

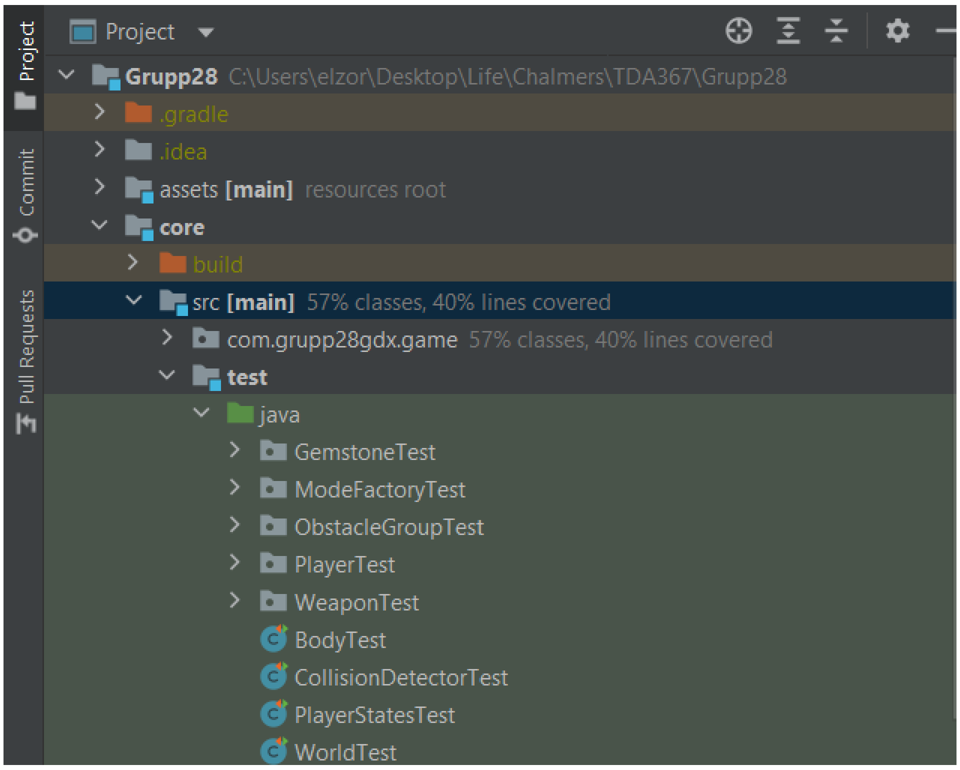
The singleton was was first used to only be able to create a single instance of a player. This was scrapped in a later iteration of the project as it was overused and unnecessary. Now the singleton is used to only create a single instance of the game configuration, and hence the game.

4 Persistent data management

All resources for the project such as pictures and audio are stores in the View in the AssetManager class, which in turn gets them all from a folder in the project called Assets. All classes in the View that uses pictures retrieves them first from that class through getters.

5 Quality

- Testing of the model is done through JUnit. These can be found in the Tests folder of the project. All methods that are public in the Model has been tested and all the tests have passed. The tests can be found below in the project.



```
Project
Grupp28  C:\Users\elzor\Desktop\Life\Chalmers\TDA367\Grupp28
  > .gradle
  > .idea
  > assets [main]  resources root
  v core
    > build
    v src [main]  57% classes, 40% lines covered
      > com.grupp28gdx.game  57% classes, 40% lines covered
      v test
        v java
          > GemstoneTest
          > ModeFactoryTest
          > ObstacleGroupTest
          > PlayerTest
          > WeaponTest
            BodyTest
            CollisionDetectorTest
            PlayerStatesTest
            WorldTest
```

Figure 5: This is the place the tests are placed in the project.

- No continuous integration tool was used for this project, but would've been nice to have.

- There are some design issues with code duplication e.g. in the obstacle and gem classes, these were not handled because of lack of time. As previously mentioned a way to fix these would be to implement the template method as was done in the player classes. This would also create abstract classes in addition to the interfaces present and improve both the readability and abstraction of the Model. There was also problems with creating a JAR-file of the project, which is why there isn't one.

6 References

The following tools and libraries were used in the making of this project:

Tools

- IntelliJ
- Git and Github
- Trello
- Figma
- Draw.io
- Google Drive
- Discord

Libraries

- LibGDX
- JUnit