System design document for Wild West Coffin Escape

David Ferm
Imad Alihodzic
Samuel Karlsson
Anders Olofsson

2021-10-22
Version 3.1

1: Introductions

The purpose of this project is to create a top down arcade-style game, inspired by 80's and 90's shooters. This is due to, in large part, the recent explosion in nostalgia fueled product sales, both of games such as Doom: Eternal (2020), and remakes of retro consoles such as the PlayStation Classic and Mini SNES.
This project aims to capture this lucrative market, and provide both older and younger persons with a fun, easy to play, endless arcade game.

1.1: Definitions, acronyms, and abbreviations

LibGDX: The game-engine library used in the production of this game.
Top-down game: A 2d game played from a bird's eye perspective.
Devs: Developers - makers of the game.
HUD: Heads Up Display. It is used in order to give the player information such as health and ammo.

2 Requirements

This part of the document aims to establish the necessary requirements of the project.

2.1 User Stories

The user stories for this game will be split into player and developer stories respectively, dictated by the wishes of the devs and players respectively. All user stories which have been completed are marked with a "✓", whilst uncompleted ones are marked with an "X".

2.1.1 Player Stories

1. As a player, I want to be able to move, in order to avoid the enemies. ✓

2. As a player, I also want to be able to progress through the game, so that I feel motivated to play. ✓
3. As a player, I want to be able to defeat the enemies, so that I can progress through the game. ✓
4. As a player, I want the enemies to follow me so that it is more difficult and I feel pressure. ✓
5. As a player, I want the game to get progressively more difficult, so that I feel stimulated no matter the skill level. ✓
6. As a player, I want this game to be a challenge, for example by having enemies that I must beat. ✓
7. As a player, I want a health system so that my mistakes have consequences. ✓
8. As a player, I want a map so that I can feel immersed in the game. ✓
9. As a player, I want there to be limitations on where I can move (e.g. walls), so that the game is more of a challenge and I must think about where I move. ✓
10. As a player, I want rewards/feedback so that I feel motivated to continue playing. ✓
11. As a player, I want to know where enemies could come from so that I am able to avoid those places at the start of rounds. ✓
12. As a player, I want a way to aim so that I can decide where the bullets end up. ✓
13. As a player, I want a HUD in order to see essential information about my character. ✓
14. As a player, I want to have limited ammunition so I have to think about how I use it. ✓
15. As a player, I do not want to be able to shoot while reloading so that I have to actively choose when I can reload or not. ✓
16. As a player, I want different weapon types so that there is more variation in the game. ✓
17. As a player, I want to be able to see when enemies are about to hit me so that I have the possibility to avoid it. ✓
18. As a player, I want animations so that the game feels smoother and more immersive. ✓
19. As a player, I want to be able to “sprint” so that I can get out of a sticky situation if need be. ✓
20. As a player, I want power-ups so that the game does not become impossible at a certain point. ✓
21. As a player, I want rewards to move slowly towards me so that they are easier to pick up. ✓
22. As a player, I want to be able to choose which power-ups I get so that I have more autonomy in the game. ✓
23. As a player, I want different types of enemies so that there is more variety to game-play. X
24. As a player, I want multiple different ways of dealing damage to zombies so that I can choose depending on the situation(melee / grenades). X

2.1.2 Dev Stories

1. As a developer, I want a collision system, so that the game functions as most players expect. ✓

2. As a developer, I want the game to be scalable, so that it's playable on a variety of desktop devices. ✓
3. As a developer, I want a path finding algorithm, so that players are motivated to move around the map. X

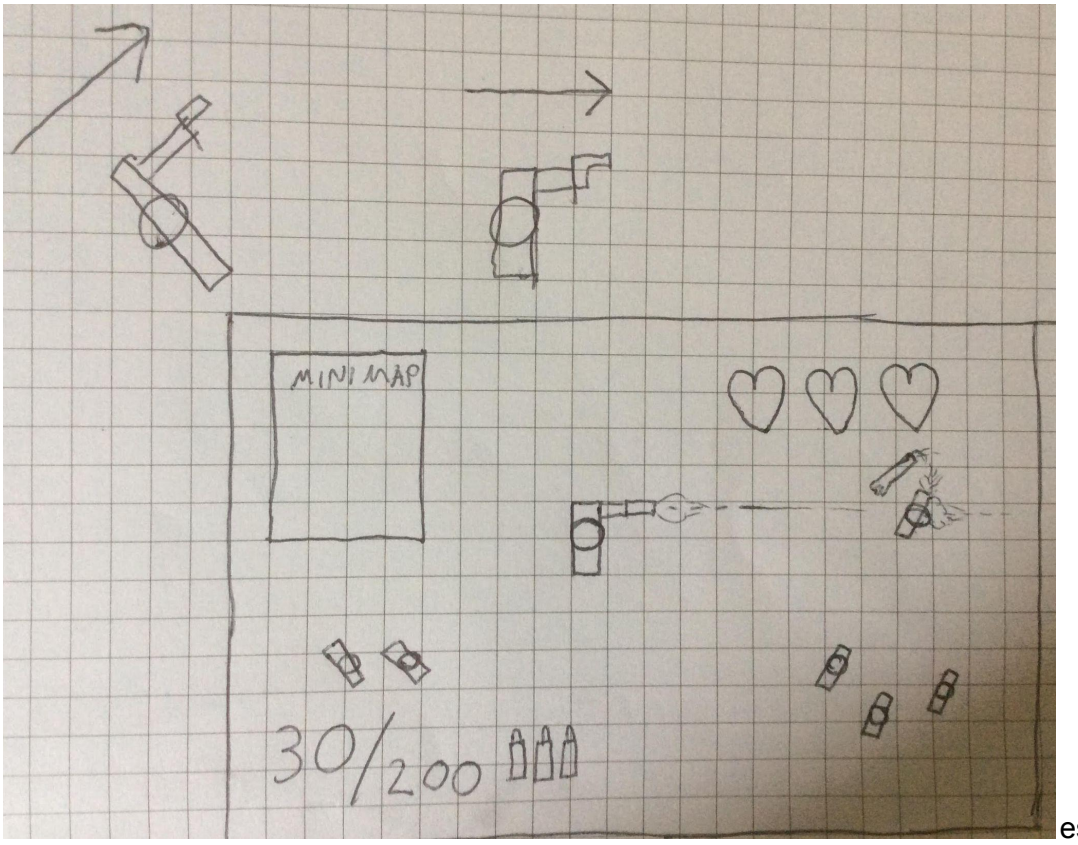
2.2 Definition of done

Games are never finished products. On the contrary, they always have room for improvement, change, and addition. However, to create something which can be played and enjoyed, one must have a point in the production cycle where the word 'done' is not a distant dream, but a definable reality.
Whilst this is something which can be viewed holistically, we will instead focus on specific features, pertaining to the user stories at hand. The definition of done is made up of a list of criteria which is common for all requirements in the project.

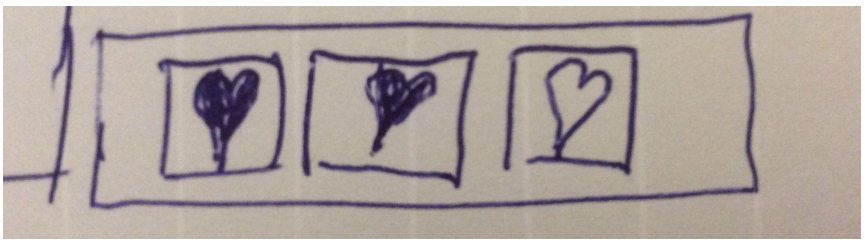
1. The user story is fulfilled to meet what the user wanted.
2. The user story has undergone JUnit tests to make sure that there is no unwanted behaviour.
3. The code written to fulfill the requirements is documented, so that other devs can see what the code does.

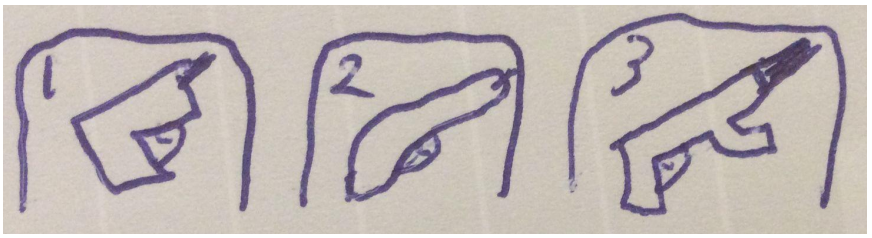
2.3 User interface

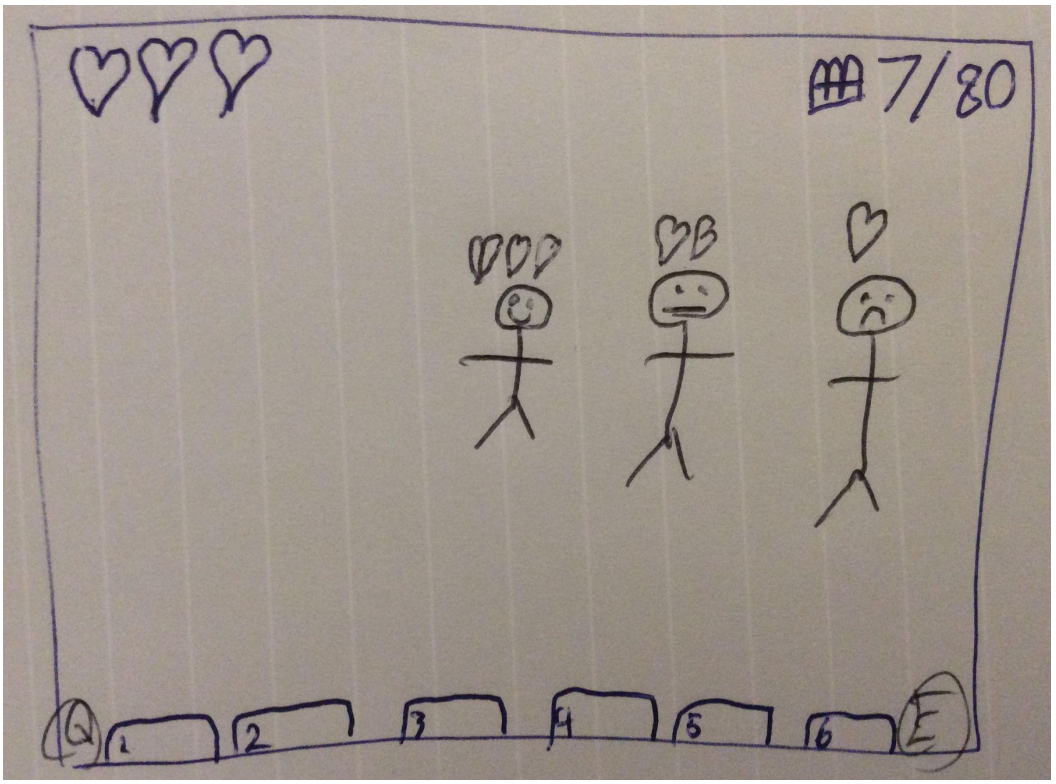
Initial sketch

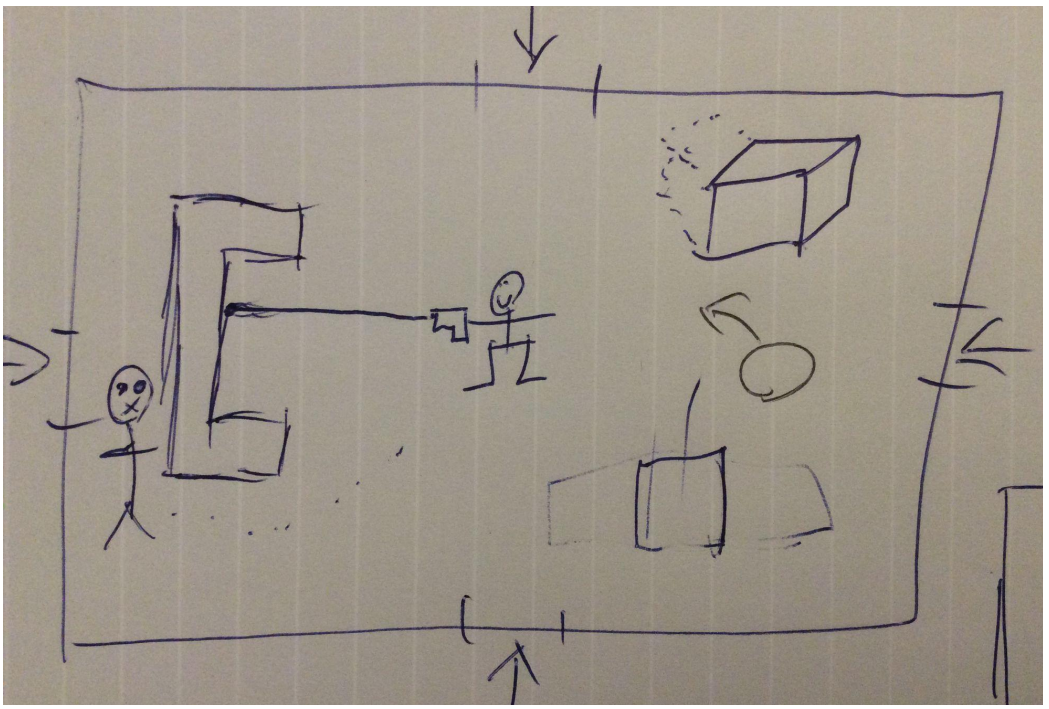


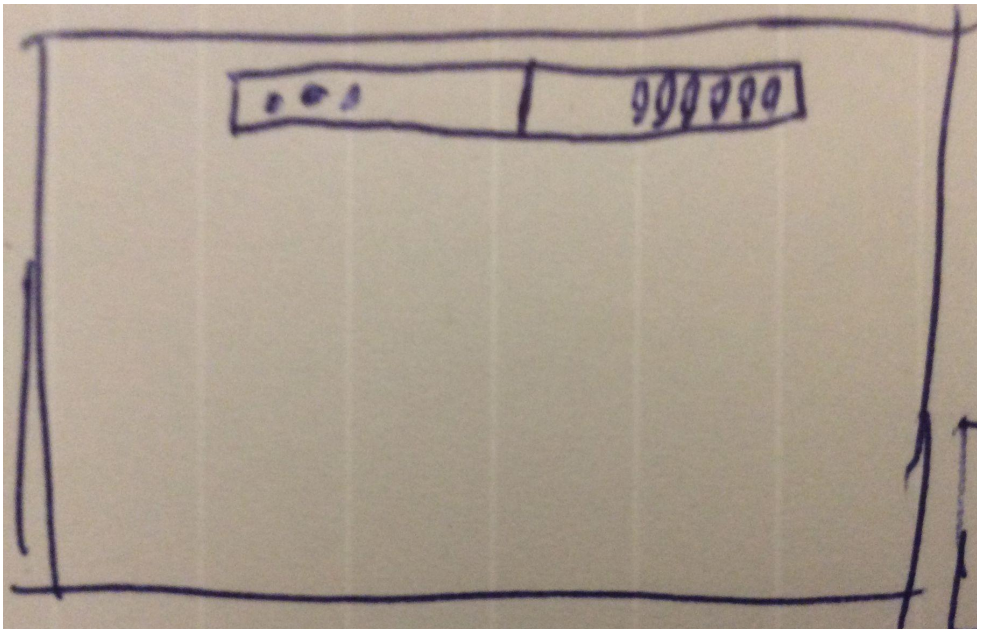
es



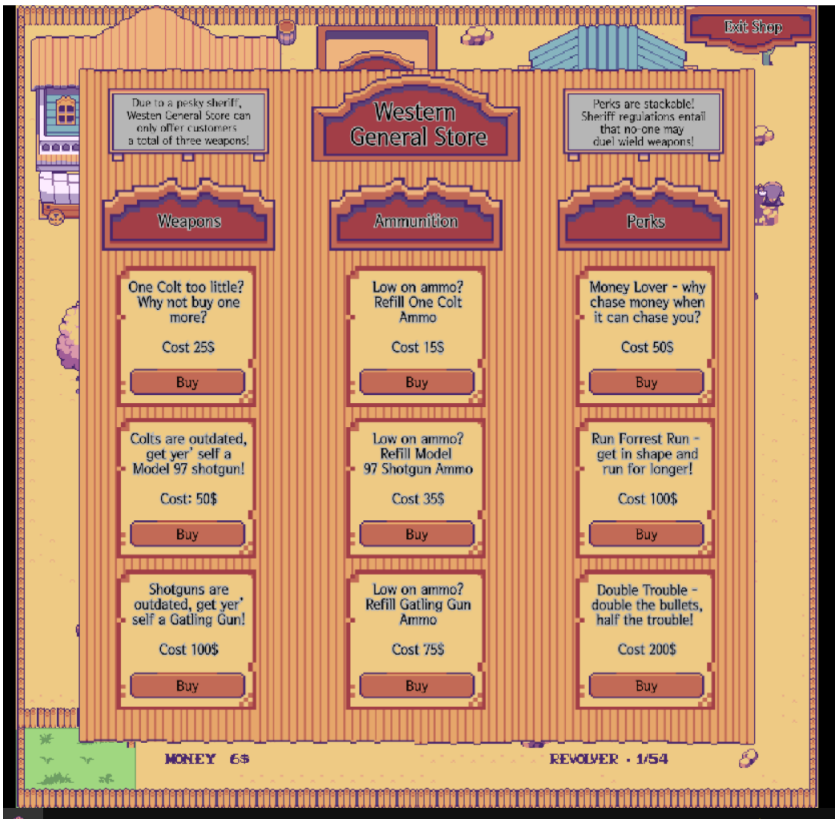

1
2
3


7/80
Q
1
2
3
4
5
6
E



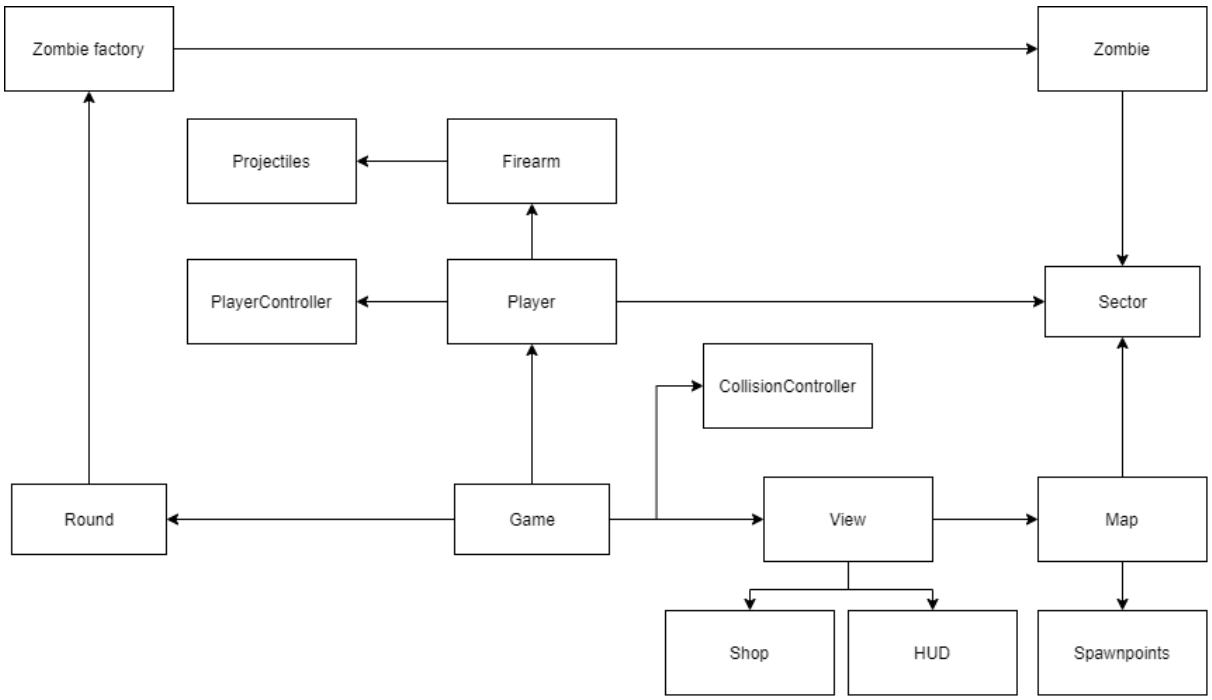


Final result



The final result is quite simple as there are only two interfaces. We have the main screen, which contains a map, the player and the different zombies, and a HUD at the bottom that gives information to the player about their in-game status.
The second image is that of the shop, which is automatically opened when a round has finished. This contains all of the different things one can buy with the money gained by killing enemies, and in the top right of the screen there is an exit button which will continue the game.

3 Domain model



3.1 Class Responsibilities

Spawn-points: Responsible for deciding where zombies spawn in.

View: Handles the output. Renders the program.

Shop: Responsible for the in-game store.

HUD: Responsible for the in-game HUD

Map: Self explanatory. Renders a map.

Firearm: Simplification of the 3 classes responsible for the separate weapons a player has. Firearms fire instances of Projectile (see below).

Projectile: The in game representation of a bullet. It deals damage to Zombies.

Player: Subclass to the superclass Sapien. Responsible for all player-specific code.

PlayerController: Handles input from user and delegates to player character.

Zombie: Same as above, but for zombies.

Zombie factory: Responsible for creating instances of Zombies, grouped within a collection. Classes such as round call this class.

Round: Responsible for round mechanics such as how many zombies spawn in, and when rounds are over/starting.

Game: Responsible for controlling the different aspects of gameplay, which are defined in other classes.

CollisionController: responsible for the collision logic in the game.

Sector: An easy way to divide the map so that everything can be located rather than using coordinates.

Works Cited

libGDX. 2021. Source & Documentation. [online] Available at: <https://libgdx.com/dev/> [Accessed 2 October 2021].

System design document for Wild West Coffin Escape

David Ferm
Imad Alihodzic
Samuel Karlsson
Anders Olofsson

2021-10-22

1: Introductions

The purpose of this project is to create a top down arcade-style game, inspired by 80's and 90's shooters. This is due to, in large part, the recent explosion in nostalgia fueled product sales, both of games such as Doom: Eternal (2020), and remakes of retro consoles such as the PlayStation Classic and Mini SNES. This project aims to capture this lucrative market, and provide both older and younger persons with a fun, easy to play, endless arcade game. This document will explain and clarify the design of the game, and the structure of the code.

1.1: Definitions, acronyms, and abbreviations

LibGDX: The game-engine library used in the production of this game.
Top-down game: A 2D game played from a bird's eye perspective.
Devs: Developers - makers of the game.
HUD: Heads Up Display. It is used in order to give the player information such as remaining health and ammo.
MVC: A design pattern used in OOP which suggests dividing the program logic into model (logic), view (the output) and controller (the input)
OOP: Object oriented programming

2 System Architecture

Program flow:
Main creates -> Model.
Model creates -> Player
Player adds itself to MovableSubject and View
Model creates -> Rounds
Model creates -> View
View creates -> Map
Model loops through the program updating program and rendering.

Loop:

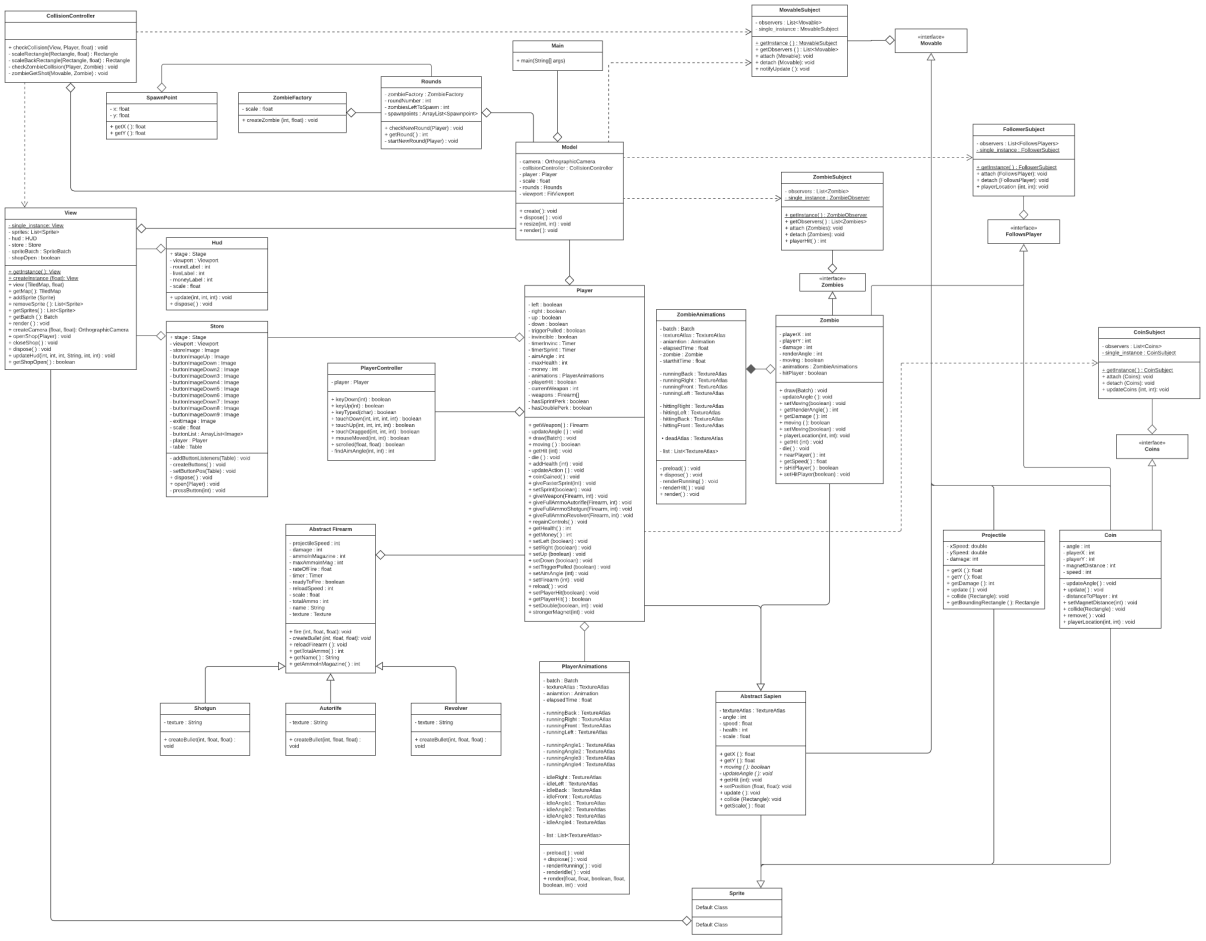
Update the HUD information

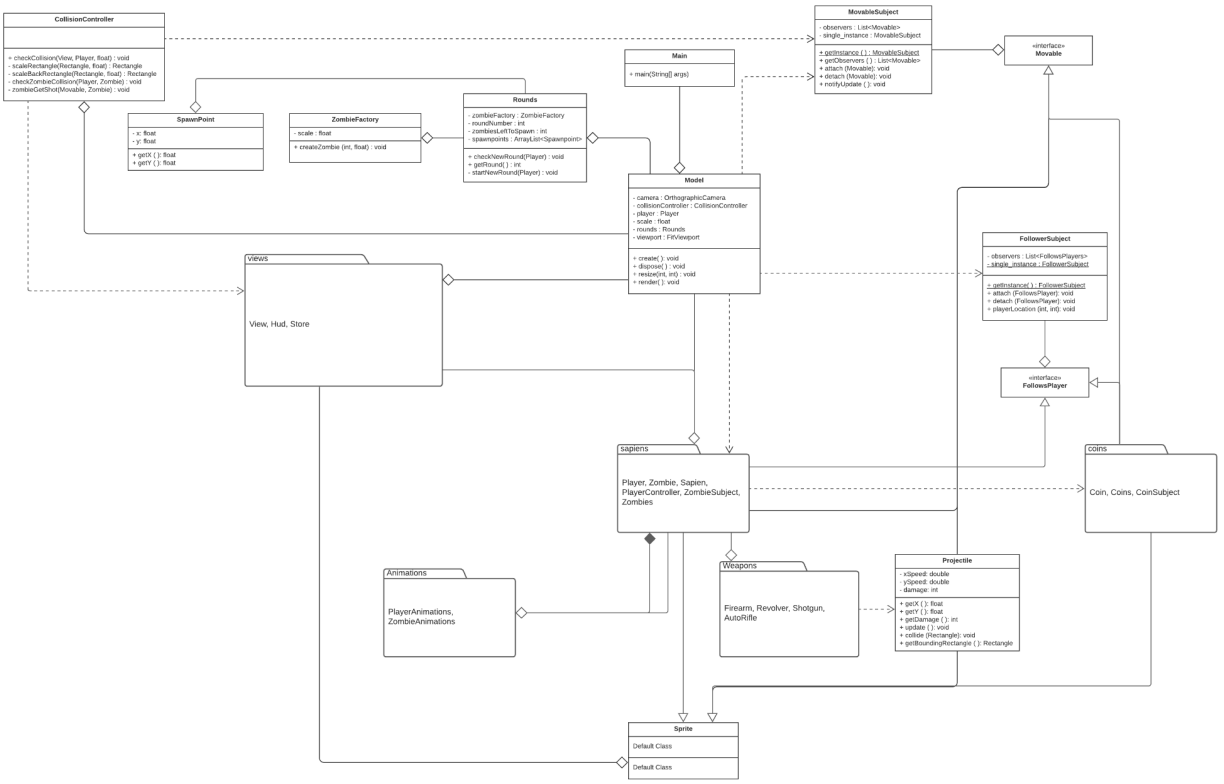
View renders everything
Checks if the shop is open or not
If it isn't:
    Update position of all movable
    Check for collisions
    Give followers the players position
    Detect if the player has been hit
If the previous round has ended or if the shop is already open
    "Open" the shop

If a key is pressed:
    Update movement of the player
    Or change the weapon slot of the player

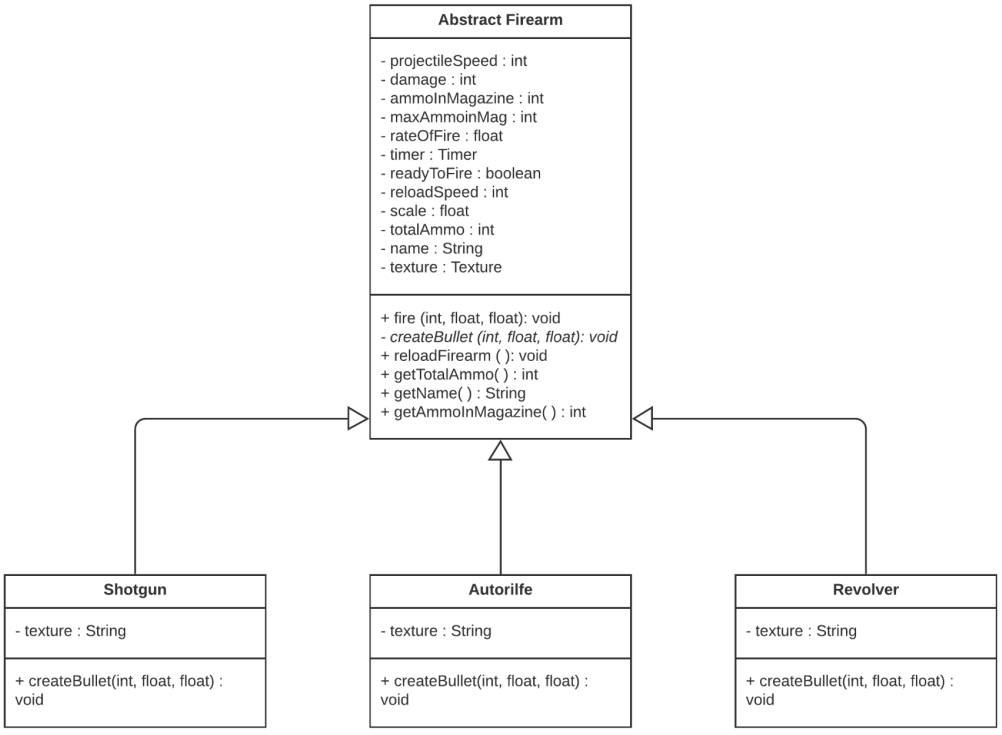
3 System design

https://lucid.app/lucidchart/95d8c238-6aba-492e-9e93-b94a9f4d99bc/edit?viewport_loc=1033%2C-90%2C4195%2C1802%2C0_0&invitationId=inv_ca584bed-bc4e-4511-9de2-8c25b043834d

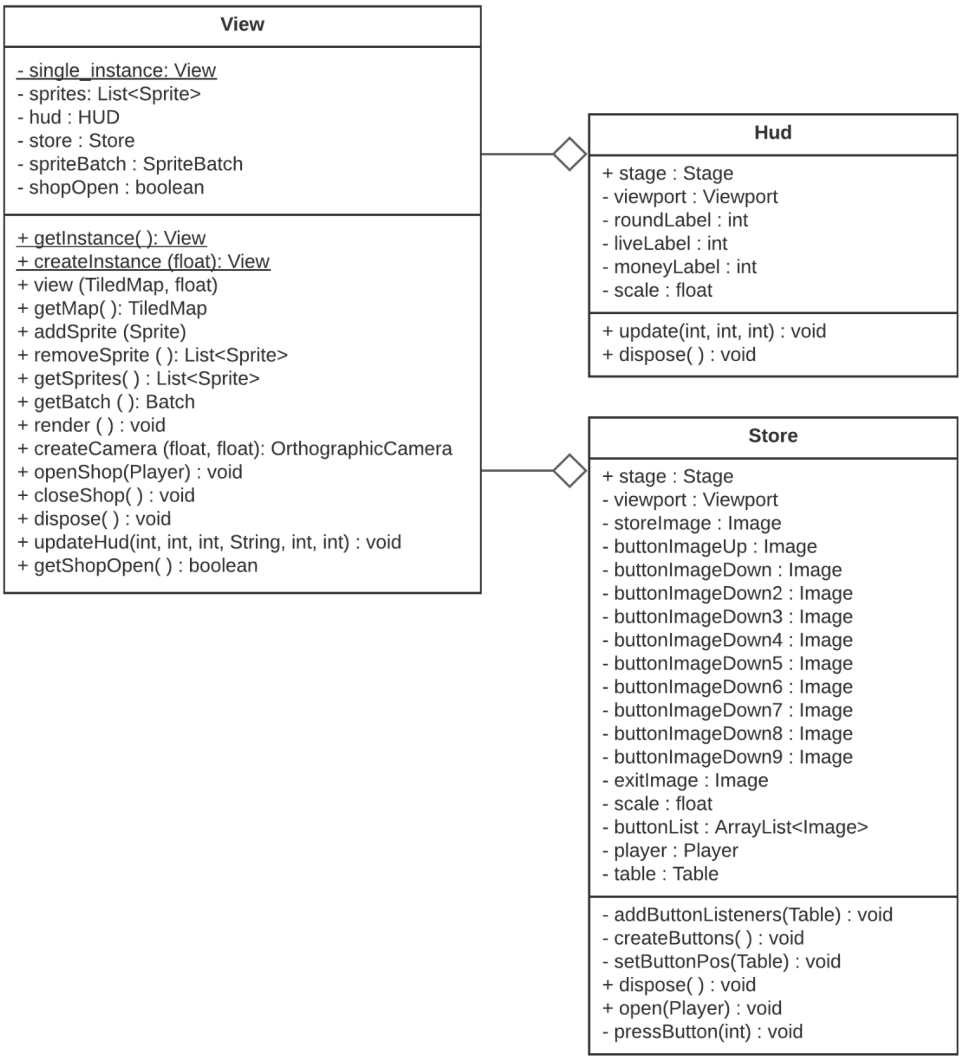




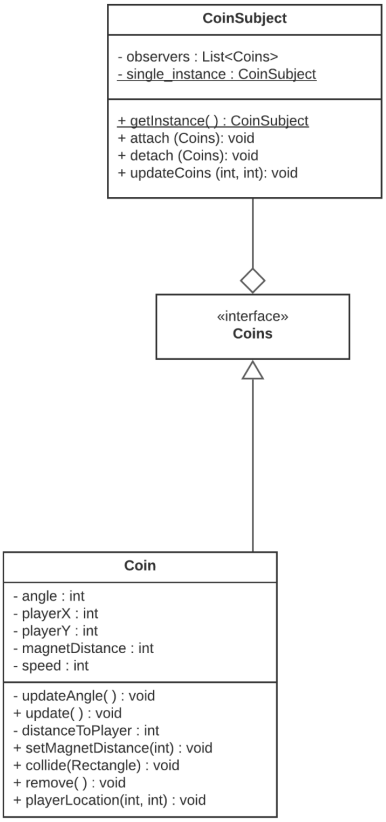
The weapons package



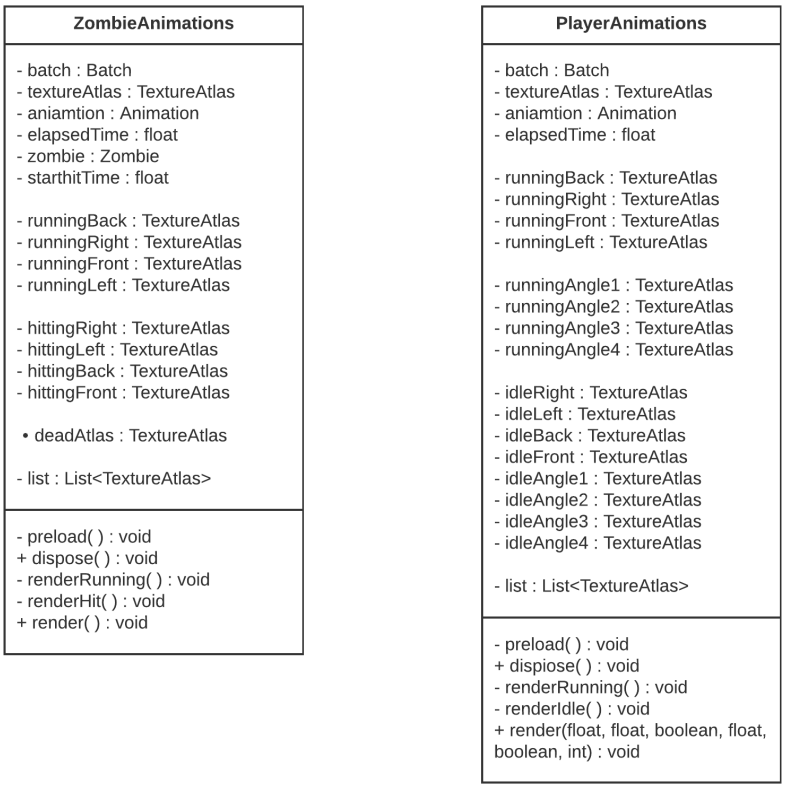
The views package



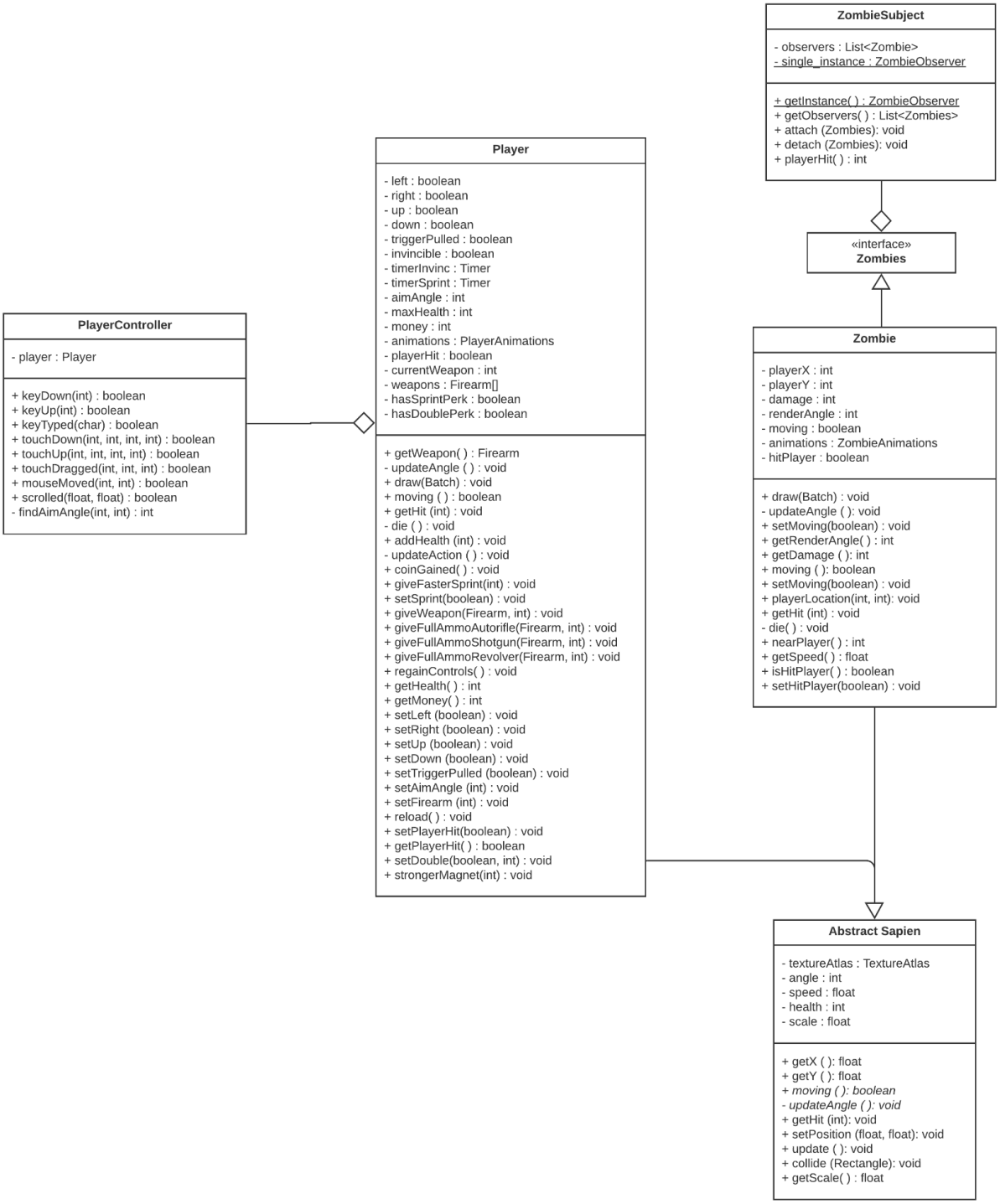
The coins package



The animations package (With more time we would have turned this into an abstraction)



And finally the Sapiens package



This program follows MVC as it contains a ***"Model"*** class, which takes care of the game loop and is the starting point for creating the objects necessary in the program, as well as a View package and a ***"Controller"*** class.

The '***Controller'*** class, found in the *Sapiens* package, is an inputprocessor which delegates to the '***player'*** class.

The *View* package contains three classes, the first of which is called ***"View"***, and which is responsible for rendering the program, as well as creating the map. ***"Hud"***, meanwhile, is responsible for the HUD, which gives the person playing the game information about the

number of lives left, the round number, and so on. The ***"Store"*** class creates the shop UI, and handles the logic behind it.
There is a clear relationship between the domain model and final design model. In the domain model the class game has a **view**, a **player**, a **CollisionController** and **rounds**. This is also done in our final design model.
One noticeable difference between the two models is that in the domain model, the map has "***spawnpoints***", while in the design model, it's the **rounds** which has spawnpoints. In fact both are somewhat correct, as **rounds** gets "**spawnpoints**" from the **map** though the ***View*** class.
Other than that, "**rounds**" has a **zombie factory** which is also represented in the design model. The ***view package*** in the design model is also very similar to the domain model, where "***view***" has a ***store*** and a ***hud***. In the ***Sapiens package*** one can again see the similarities between the domain and design model. The **player** has a **firearm** as well as a **playerController**.

One design pattern which has been heavily applied in our program is the observer pattern. This can be seen with MovableSubject, FollowerSubject and CoinSubject. This is used to affect multiple objects with similar behaviour when something happens. For example, MovableSubject is used to update the movement of all objects, and FollowerSubject is used to give the position of the player to all objects which need to follow the player.
Another pattern which we use is the singleton pattern. This is used alongside the observer pattern to make sure that the list of observers is always updated and there is only one instance of it.
The factory class pattern is also implemented, as evidenced by the ZombieFactory class. The template method pattern can also be seen in the Sapien class where the logic behind updating the angle is different for the subclasses, and so the abstract method updateAngle() is called.
We also use a form of state pattern for firearms. The Player has an array of the abstract class Firearm, and so can change the behaviour depending on what firearm is chosen.

4 Persistent data management

The game assets are kept within an asset folder, which is managed in large part by the LibGDX library.

5 Quality

JUnit tests have been done throughout the code. These can be found in a test folder outside of the source folder. We have also tested the code by playing the game and seeing if any issues arise. More specifically, we have focused on playing the game in such a way as to specifically find glitches/bugs, by doing things a player isn't typically expected to do. Examples include trying to run into buildings, and reloading weapons over and over.

We have also run PMD code quality tests (see assets folder on Github). One thing to note is that whilst PMD generated 1355 'comments' about how the code could be improved, a large

number revolve around j-doc comments, and unused imports, thus making the number of issues with the code over-exaggerated.
In addition, we have also used the inbuilt Intellij tools to run multiple tests. When running Intellij testing for illegal dependencies, the results came back as negative - with no identified illegal dependencies. The same also applies for illegal backwards dependencies,
As for general issues and warnings, this can also be found in a separate HTML document in assets. Note that most of the warnings are about typos in comments.

One issue which we have come across recently is that the shop does not work when the window is resized. The reason we have not looked into this is due to a lack of time. Another issue is that animations have fairly similar code and so a super-class could be made and code extracted. This would mean that instead of Player having a PlayerAnimations and Zombie having a ZombieAnimations, Sapien would have an instance of Animations, which would be the superclass.

Works Cited

libGDX. 2021. Source & Documentation. [online] Available at: <https://libgdx.com/dev/> [Accessed 2 October 2021].

Behövs göras bättre:

Bra att kunna se var man startade ifrån

Lägg gärna till labels som visar hur mycket poäng båda spelare har så man slipper konstant räkna antal rutor man har

När man räknar antal steg som en spelare kan gå så tänker man inte på ifall den andra spelaren skulle hamna i vägen, så man kan ta olagligt många steg för att gå runt den andra spelarens main dancer.

Efter ett visst antal rundor slutar spelet räkna och väntar tills spelet är över.

PlayerTurnSlot enum är helt onödig, finns mycket mer intuitiva lösningar som minskar logiken krävd för att räkna ut allt. Exempelvis räcker det att ta turn number och kolla om det är jämnt eller ojämnt tal. Så får man ut vilkens spelares turn det är.

Ska man kunna draw cards så mycket man vill?

Oklart när spelet är över och vem som har vunnit, spelarna kan fortfarande spela efter att någon har vunnit

Det finns väldigt mycket tips på skärmen, borde vara mer intuitiv, kan va bra att minska det eller ha en separat view för hjälp

Model borde inte ha 500 rader kod, när logiken kan delas upp i olika klasser

Vissa metoder i model är väldigt långa och svårläsliga, t.ex. “moveSelection(int keyCode)”. den kan delas upp i mindre mer förståeliga och testbara metoder.

Vissa fields i model som är public kan bytas ut till private

Inte nödvändigtvis något dåligt, men spelet tar bara slut först när en spelare får ett kort som täcker både den sista vita rutan samtidigt som den täcker den rutan som spelaren tidigare stod på, annars fortsätter main dancer att skapa vita rutor när den går bort och täcker andra vita rutor.

Bra saker:

View och Controller har en instans av Model och det finns inga dependencies som går åt andra hållet. Det är i linje med MVC. Modell delen är även separerad till ett eget package.

Koden är också lättare utbyggbar p.g.a uppdelningen av MVC.

Koden innehåller också mycket felhantering.

- Design patterns
    - Consistent?
        - Model is a God Class
        - The enum dancetype is not used
    - Reusable?
        - Not really as almost all logic is in the model class
        - Controller class is simple enough to be reusable
    - Maintainable?
        - Difficult to maintain with god class and very long methods
    - OCP
        - Not very OCP with hard coded initial deck
        - The cards are hard coded. Would be a lot better to have separate classes for each card type?
    - Design patterns
        - They admit themselves that no patterns are being used
- Documented
    - Yes
- Proper names
    - Good naming in general. Some names are a bit longer than needed, but still conveys the objects purpose. T.ex. moveMainDancerOfCurrentPlayerToIndex
- Modular design
    - Not modular with hardcoded cards
    - Number of players is difficult to increase. Not impossible but a faff.
- Abstractions
    - A few abstractions are used but not enough. Though it is used for players, controller and view.
- Well tested
    - One should maybe not make a constructor only for testing, the tests should instead call the constructor that is actually used in code with the parameters set in these test methods
- Security problems & performance issues
    - Render method in view has nested for loops and does not need to be called each “loop”, as it could be the case that nothing has changed.
    - ^^ We highly recommend having checks within render so that instructions are only executed after a change in program state.
    - Some instance variables are not private
- Easy to understand
    - It has a semblance of MVC. The model is isolated, however it is does not delegate the game logic but does it all itself.
    - Code is difficult to read because of God classes and methods which are very big.
    - Remove commented out code and unnecessary whitespace to make the program more readable.