NTNU
Norwegian University of Science and Technology

Duel of Fates

Implementation

Group 13

Hassel, William Strand

Marleau, William Andreas Krohn

Nguyen, Ken Tan van

Rigatos, Dionysios

Rosset, Adrien

Somorjai, Márk István

Primary Quality Attribute

Modifiability

Secondary Quality Attribute

Usability

COTS

Android SDK, LibGDX, Firebase

1. Introduction

1.1 Description of the Project & this Phase

The educational goal of this project is to “learn to design, evaluate, implement and test a software architecture through game development”. We have applied this by designing and implementing a software architecture for a multiplayer mobile game for Android. This document will therefore serve as the design, implementation and testing phase documentation.

Throughout this phase, the project has been implemented by the selected quality attributes and architecture. Drawing inspiration from "Slay the Spire”, we designed our project to match these ideas. Testing procedures have been implemented to ensure that both functional and quality requirements are met.

1.2 Description of Game Concept

The application will be a 1-versus-1 duel card game inspired by the popular game *Slay the Spire’s* [1] combat system, which served as a reference for both gameplay and artwork. You can choose to be a brave knight, a wise mage or a devious skeleton - each one with its benefits and drawbacks. In the arena, you can choose to save up your mana and perform a chain of attacks in your turn, or consistently make moves to keep your opponent on their toes. Depending on your class, you have certain hidden aces in your sleeve that can turn the tables of the battle.

Figure 1 - Screenshot of the interface of “Slay the Spire” from Mega Crit Game

The arena is brutal - either one or no one will make it out alive. This is why it is important to keep track of creeping lasting effects the opponent has cast on you. When it's a player's turn, they can choose to play any of the cards on their hand they can afford with their current level mana. Multiple cards can be played in a single turn, thus making the management of mana particularly important. The effects of the cards are immediately cast on the opponent. As soon as the round concludes, the opposing side has their passive mana refill and gets a chance to take a shot. The game ends when either player runs out of health.

Figure 2 - Screenshot from a battle in Duel of Fates

2. Design & Implementation Details

As a complex Android Application Game implemented with LibGDX, a variety of tactics and patterns were used to make its development process more reliable and feasible, as well as support selected quality attributes. For an easier understanding of our implementation, we have provided various class diagrams for visualization, as well as an overall package diagram.

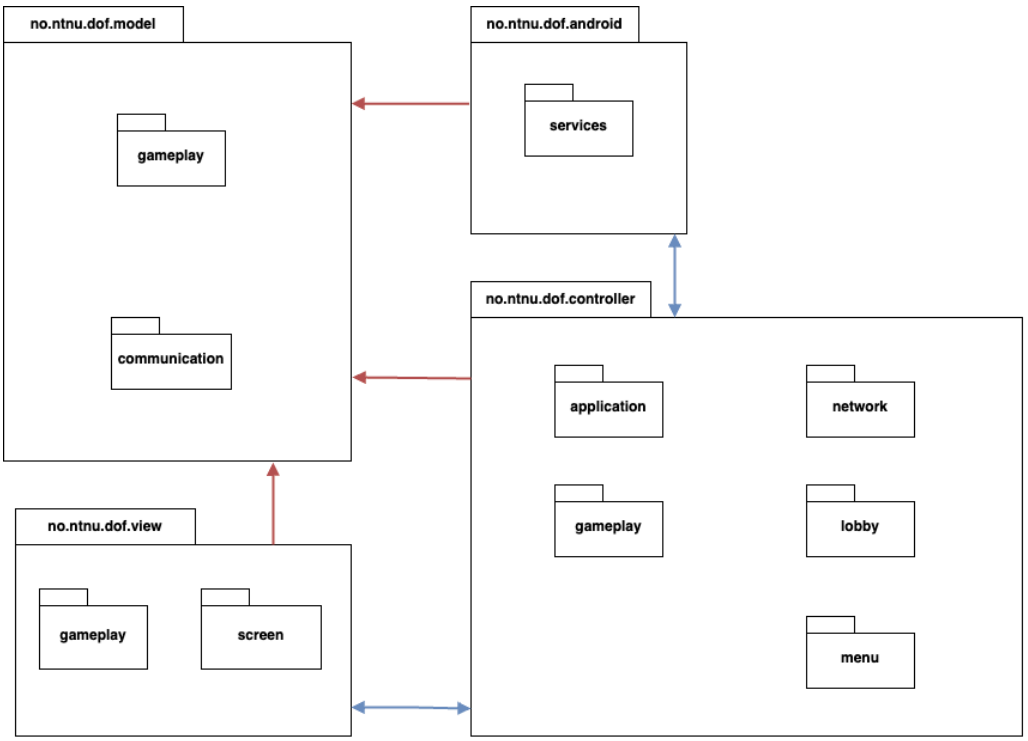

Figure 3 - Application Package Diagram

An overview with quick access to the class diagrams is provided below, as they show a finer view of the project.

Class Diagrams:

- Gameplay Controllers
- Model-View-Controller (Gameplay)
- Gameplay Model
- Application Screens
- Application Controllers
- Firebase Services

- Model Communication
- Inheritance of Gameplay Views

2.1 Tactics

2.1.1 Modifiability Tactics

2.1.1.1 Reduce Coupling

2.1.1.1.1 Intermediaries

In our application, we are utilizing an intermediary class to prepare and navigate to new screens named ScreenController. In this way, we are decreasing direct coupling between the screens and controllers by centralizing the control of such handlers.

2.1.1.1.2 Abstraction of Common Services

Coupling is also reduced through the abstraction of common screen services, such as the ability to toggle the game music ON/OFF or return to the previous screen. This is implemented in the abstract classes BaseScreen and its extension ReturnableScreen - as can be viewed in Figure 7.

2.1.1.1.3 Encapsulation

The interfaces implemented in relation to our networking module hide the details of how the networking methods are actually implemented for each respective platform. This encapsulates code, decreasing the risk of having to change it in numerous places. Encapsulation not only shields the underlying implementation of networking methods but also helps with modularity within our system architecture. Abstracting the complexities of networking operations behind well-defined interfaces makes our codebase more maintainable and scalable. This modular design allows for easier integration of new features and minimizes the ripple effect of modifications.

2.1.1.2 Increase Cohesion

2.1.1.2.1 Split Module

The networking and game modules were split into smaller modules to reduce complexity and increase cohesion. Networking was split into parts representing different areas of

responsibility, whereas logic and objects in the game module such as cards, effects, and players, were split into separate packages.

2.1.1.3 Defer Binding

2.1.1.3.1 Compile-Time Parameterization & Configuration-Time Binding

Since our application contains a lot of static entities (e.g. cards, effects, decks) that have to be initialized at various points something we achieve using a dependency injection framework called Dagger2, which allows for all of the aforementioned and promotes modifiability by having most of the game model object creation in one place - called a Module - where one can modify and adjust them without having to dig deep into the game logic. Modules are easily modified and added so as to provide common services across the application from a single file, where each file has a specific purpose of providing objects to Components. These dependencies are evaluated during compile-time and components are then used to inject these dependencies into the requesting classes automatically at the startup of the application.

2.1.1.3.2 Polymorphism

One of the multiple examples of polymorphism in our application is the PlayerController interface which is implemented by RemotePlayerController and ClickPlayerController, where they serve the same purpose - to communicate the actions taken by a player - in a significantly different way. Polymorphism is very common throughout the application.

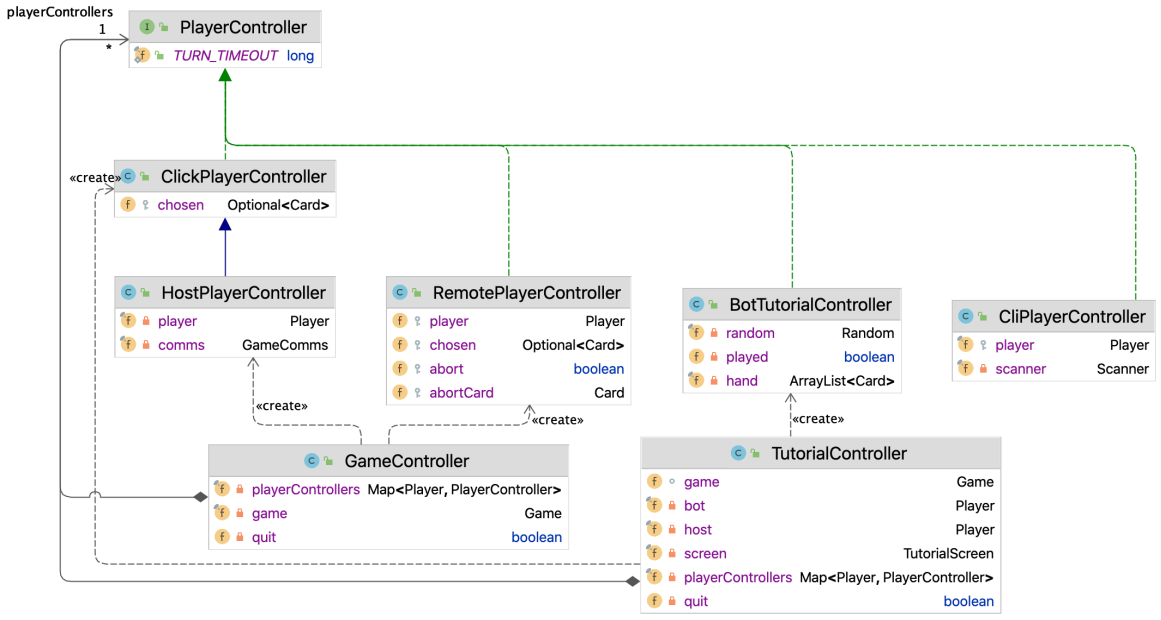

Figure 4 - Class Diagram of the gameplay controllers

In this diagram, we see that the PlayerController interface is implemented by 4 different classes that are used by GameController or TutorialController to get what card a player plays. ClickPlayerController is responsible for handling input from the user through the graphical interface. HostPlayerController extends this with network communication, sending the chosen to the server. RemotePlayerController listens for database updates and converts them to plays during the opponent's turn. CliPlayerController gives the user the possibility to play through the CLI, which we used for testing before the graphical interface was implemented. BotTutorialController is what chooses plays for the bot in the tutorial. Notice how the PlayerController interface allows the integration of graphical, CLI, bot and networking implementations, without any necessary changes to the rest of the application.

2.1.1.3.3 Resource Files

All of our graphical assets are dynamically loaded as image objects (usually in a PNG format) from the resources folder when required by the application. They are then stored in memory and presented when required, and ultimately disposed of when they are no longer needed.

2.1.1.3.4 Shared Repositories

Firebase is used as a shared repository among users (and subsequently players when in-game) for sharing active lobbies, game match histories, exchanging cards during a fight and so on.

2.1.2 Usability Tactics

2.1.2.1 Support User Initiative

2.1.2.1.1 Undo

In order to allow for easy navigation between screens, we implement a "back" button on every ReturnableScreen, which promptly returns you to the previous screen you were viewing. Some screens however are not returnable in this classic fashion. Exceptions are GameScreen and TutorialScreen, which are considered in-game screens, where we allow the user to exit by a "White Flag" button which forfeits the match. Additionally, if you are in the Main Menu, there is no prior screen and you can only "Log Out" of the application using the respective button.

2.1.2.1.2 Cancel

We implement “Cancel” in order to avoid phantom lobbies in the LobbiesScreen. A lobby Host may exit the Lobby Screen by clicking the “Delete Lobby” button, which acts both as a return button as well as a “garbage” collector for the lobby instance from Firebase.

2.1.2.1.3 Pause/Resume

A screen should only be active when a user is browsing it. We do this by utilizing LibGDX’s stack functionality, we utilize push and pop screen calls to manage screen flow. When transitioning to a new screen, the current stack gets paused as the new screen gets pushed. This approach prevents unnecessary resource consumption and allows for efficient utilization of system resources. As a result, users can smoothly navigate through different screens without experiencing delays.

2.2 Architectural Patterns

2.2.1 Model-View-Controller (MVC)

To follow the Model-View-Controller (MVC) architecture pattern, we organize our code into three primary packages, attempting to minimize dependencies among them.

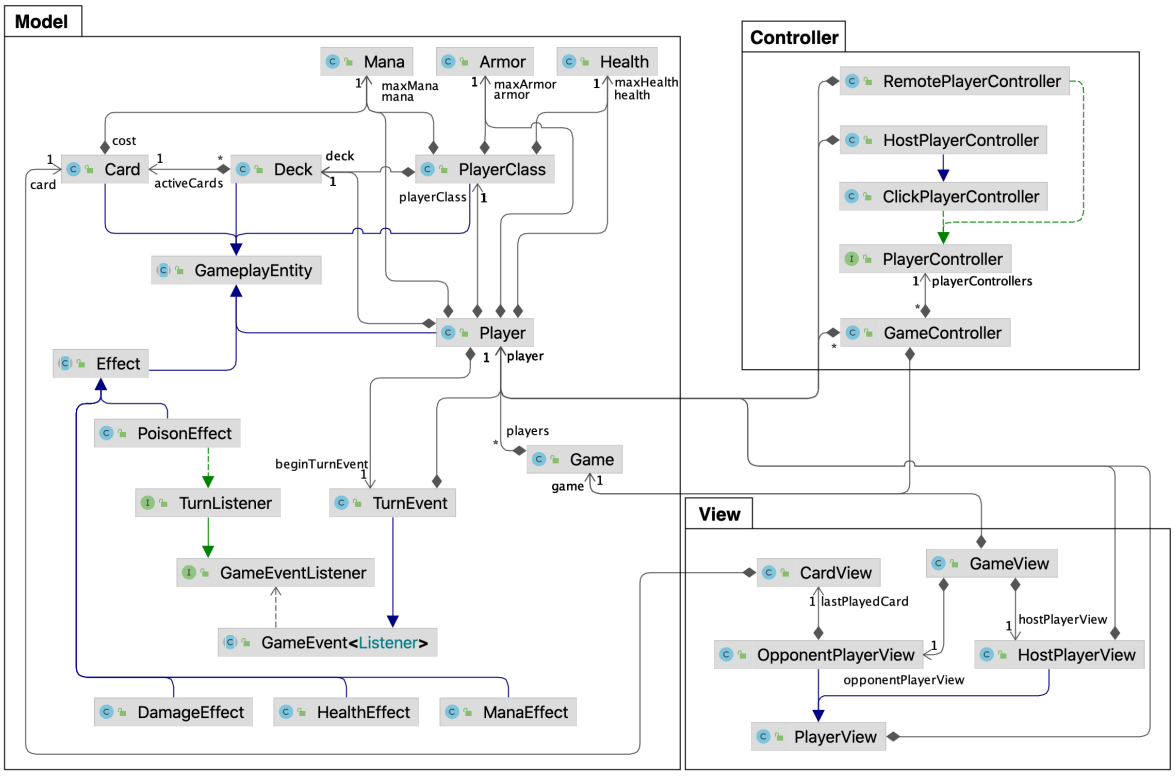

Figure 5 - Diagram of main classes used in a game.

Within this diagram, the three distinct packages are clearly identified, with attributes/fields being excluded for brevity and clarity. Let's explore a scenario to illustrate their interaction, focusing on the actions triggered when a user plays a card. Initially, the “CardView” class, which implements the “ClickPlayerListener”, detects user clicks to determine which card is played. For the opponent, the “RemotePlayerController” manages the recognition of the played card. These controllers then communicate the played card to the “GameController”, which in turn updates the game model. This update subsequently propagates changes across other dependent models.

In the subsequent frame rendered by LibGDX, the views fetch updated data from the models to refresh the display. This interaction between components exemplifies the benefits of the MVC architecture in our system.

In the following three diagrams, we explore an expanded view of the gameplay model, screens, and controllers.

2.2.1.1 Gameplay Model

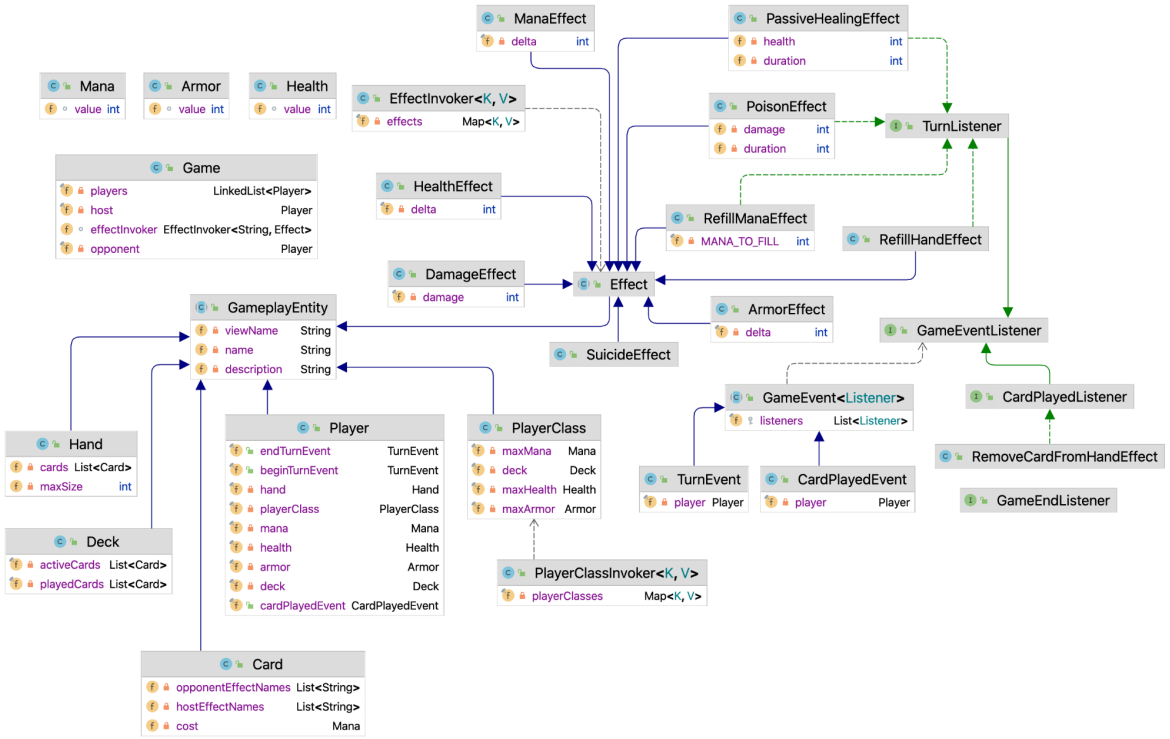

Figure 6 - Class Diagram of Gameplay Model

For our Gameplay Model, most of the game entities inherit from a common “GameplayEntity” abstract class which contains common attributes. Each unique entity then can implement its own behavior or be an abstract class itself, with “Effect” being in the spotlight as it contains a few implementations under it. Additionally, implementations of “Effect” may freely implement listener interfaces so as to notify relevant Controllers when specific actions (i.e. apply method of a recurring effect) are performed. The Game class is responsible for orchestrating the gameplay, keeping track of the state of the game and applying the effects of cards and finalizing turns.

2.2.1.2 Screens

Figure 7 shows our screens within the view component. LoginScreen is the screen you are shown when loading the application. It exclusively handles login/signup operations and does not extend a BaseScreen like other screens do. BaseScreen renders the background and also introduces the sound button visible on most screens. ReturnableScreen extends the BaseScreen, adding only a back button for the pages needing it. MenuScreen and LobbyScreen do not need back buttons, as they are replaced by “Log Out” and “Exit Lobby”/ “Delete Lobby” buttons respectively.

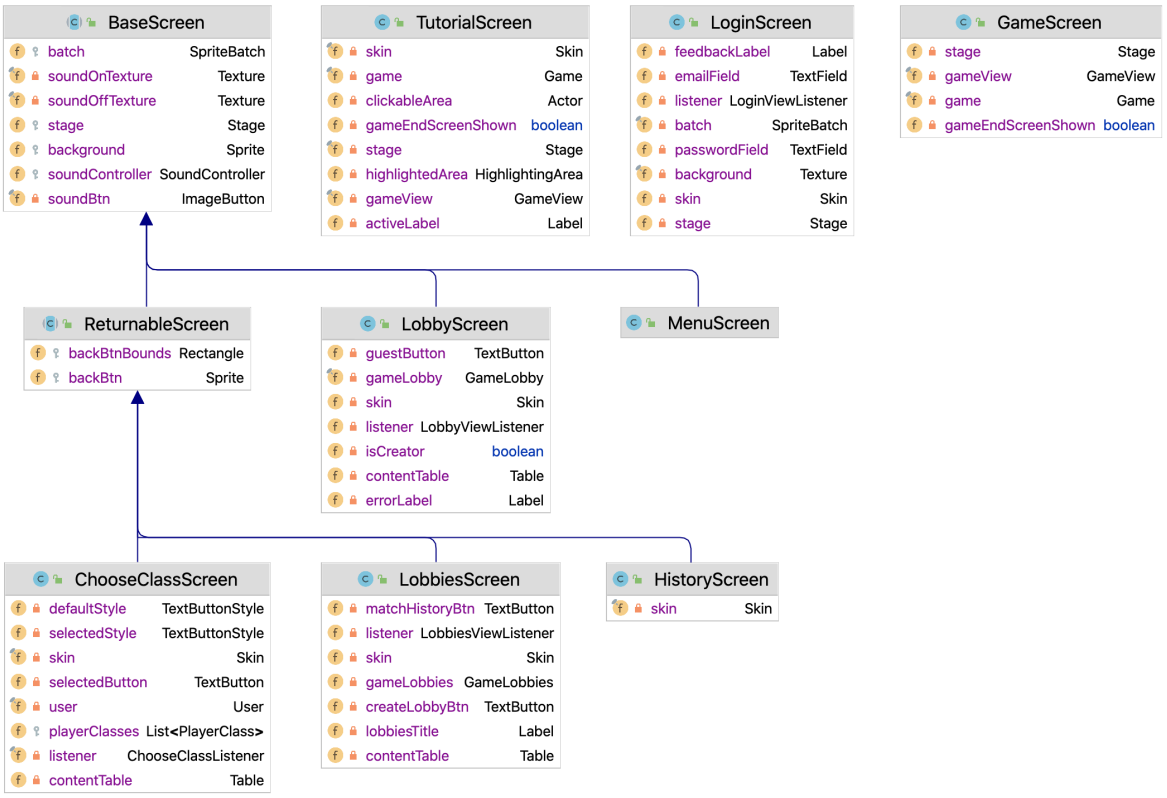

Figure 7 - Inheritance Diagram of Screens

Each Screen is connected to a dedicated controller, except for the MenuScreen, which only redirects to other screens through the ScreenController. The controllers are illustrated in the next section.

2.2.1.3 Controllers

In Figure 8, are each controller classes, named with a prefix referencing the respective screen it is responsible for communicating with. The DuelOfFates class is the application controller which is accessible by the ScreenController to send appropriate information to the other controllers. For example, when the ScreenController creates the GameLobbyController it also passes the currentUser object. For visual purposes and to save space, the service interfaces and the ServiceLocator are included in this diagram, which will be explored in the next section.

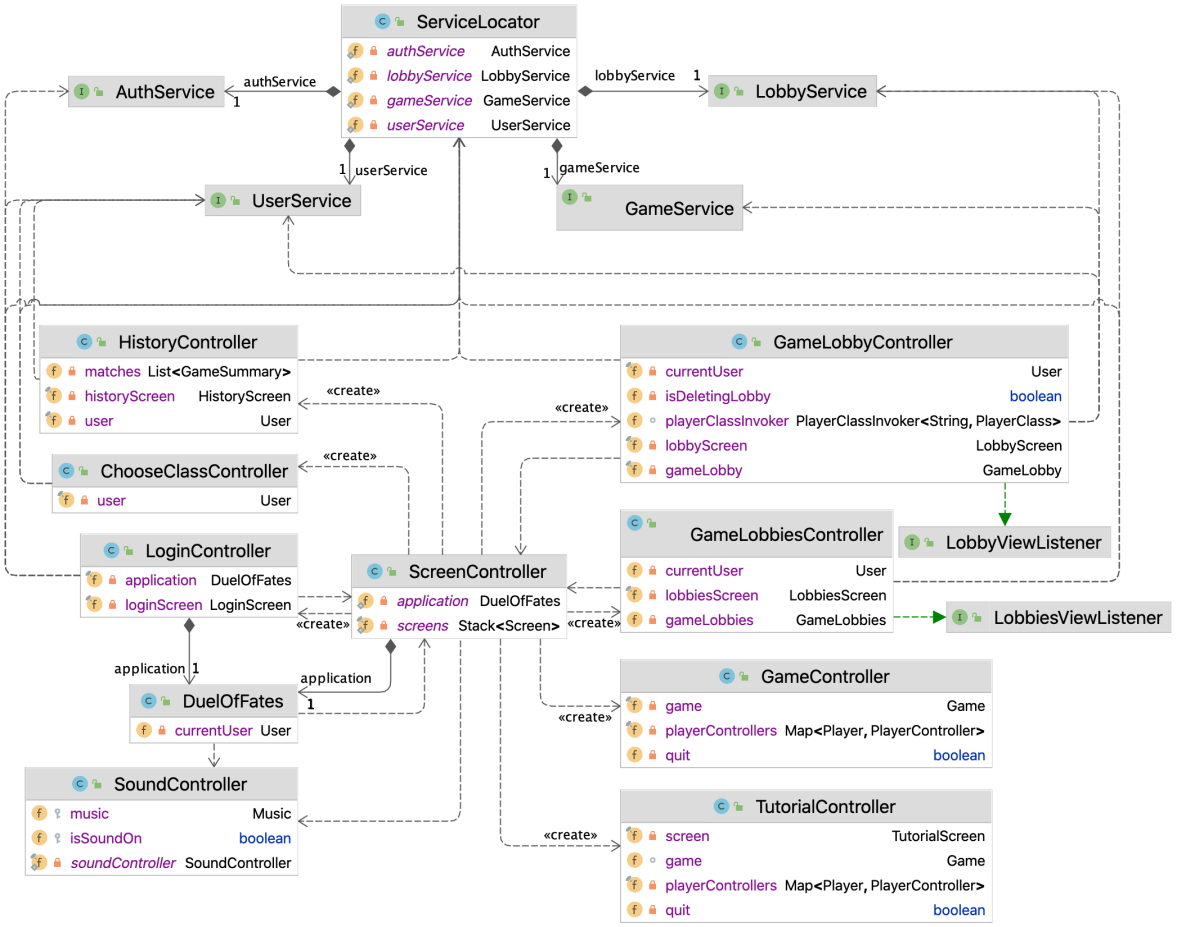

Figure 8 - Class Diagram of the controllers and ServiceLocator

2.2.2 Service-Oriented Architecture (SOA)

Through utilizing a service-oriented architecture and Firebase as Backend-as-a-Service, we've managed to enhance modifiability and modularity. In our application, Firebase is used for a variety of functionalities from authentication, storing user information across sessions, to communication between the clients during the gameplay itself. Notably, *Firebase Authentication* and the *Firebase Realtime Database* services were used.

Communication to these services was implemented through a series of interfaces and platform-specific code, following LibGDX best practices for enabling and modifying service functionality across different devices [2]. The communication to these Firebase services was split up into the following interfaces: AuthService, GameService, LobbyService, and UserService - as shown in Figure 9. Platform-specific code is implemented in their respective folders. Full functionality was implemented for both desktop and Android.

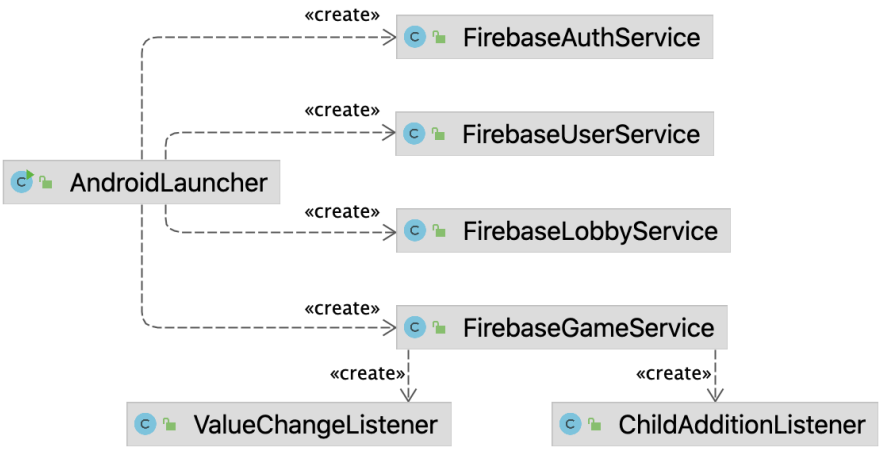

Figure 9 - Class Diagram of classes related to Firebase

2.2.2.1 Firebase Services

2.2.2.1.1 AuthService

This interface is responsible for communicating with Firebase Authentication for operations such as logging in, logging out and signing up.

2.2.2.1.2 UserService

UserService is responsible for storing and fetching information from a specific user across sessions. This involves adding the user object to the Firebase Realtime Database upon signing up, fetching user info, as well as uploading and fetching game summary data.

2.2.2.1.3 LobbyService

Responsible for creating/deleting lobbies, fetching available lobbies, joining and leaving them. Communicates with the Firebase Realtime Database and utilizes listeners to be able to notify relevant controllers of updates from the database in real-time. For example, if a guest joins a lobby, LobbyService will notify the host client, allowing the screens to access the updated information and show it.

2.2.2.1.4 GameService

Interface for gameplay communication. It is responsible for updating the game state in the Realtime Database (i.e. uploading a card that was played) as well as for listening for updates.

2.2.2.1.5 ServiceLocator

To access these services across the application, ServiceLocator is implemented. As the application is initialized, the respective platform-specific services are instantiated in this class. With getters on its static fields, controllers across the application are able to leverage the services needed through the respective interfaces.

In Figure 8, we can see how the ServiceLocator is implemented with four distinct services and controllers. The controllers handle user interactions and determine which services are needed. They then request these services from the ServiceLocator. The services themselves are responsible for retrieving and managing their data.

2.2.3 Entity Component System (ECS)

We implement attributes (Mana, Armor, Health) as a component that can be reused. They are used for the Player's in-game (live) values, as well as in the PlayerClass as initialization values. Mana is also reused as a component of the card entity for the cost.

2.2.5 Publish-Subscribe

We implement the Publish-Subscribe pattern with events: CardPlayedEvent and TurnEvent. These are the channels through which each player (publisher) can notify its subscribers about certain events happening to it: what card the player played, or whether its turn has begun/ended. We utilize these notifications for cards with longer-lasting effects, e.g. PoisonEffect which deals damage over several turns.

2.2.6 Shared-Data

Our utilization of Firebase follows the Shared-Data pattern as we used it to store information about available lobbies and the players they contain, active game state and user match history, some of which can be accessed by all user-stakeholders (i.e. users looking to enter a lobby) or shared by two players in an active game (cards and effects that were played).

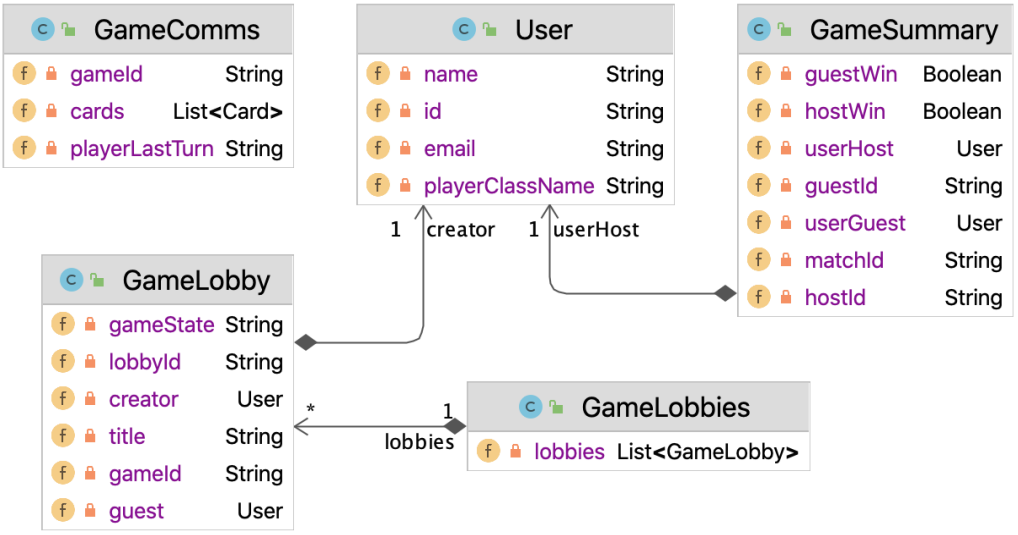

Figure 10 - Class Diagram of model communication package

All GameLobby, GameLobbies, GameSummary and User classes shown in Figure 10 contain information fetched from Firebase, while the GameComms class acts as the communication object during gameplay: played cards get pushed to its list of cards, while its playerLastTurn attribute communicates the end of a player's turn.

2.3 Design Patterns

2.3.1 Creational Patterns

2.3.1.1 Singleton

SoundController is a prime example of a Singleton. It contains the Music object which, for auditory consistency, is common across all of the screens, and for usability, remains

muted/unmuted if a user toggles it at any point. Additionally, we avoid memory leaks as we always fetch the same active instance instead of continuously creating a new one.

2.3.1.2 Builder

Builder is the most used creational pattern in our application, especially in gameplay entities (i.e. effects, cards, decks, playerclasses) as it provides a versatile way of creating objects with multiple, and sometimes optional, attributes in a clear and concise manner. We use Lombok's annotations to make classes Builder-initializable.

2.3.2 Structural Patterns

2.3.2.1 Facade

The Facade pattern is implemented throughout the service interfaces, hiding the complexity of methods and cross-platform functionality. The entry point of the Facade is the ServiceLocator class, where these services can be accessed.

2.3.2.2 Adapter

ChildEventListener and ValueEventListener are interfaces through which Firebase can notify us about changes in the database, such as a card being played. We use the classes ChildAdditionListener and ValueChangeListener as adapters, to transform these data updates during gameplay to plays made by the opponent, which can be passed onto a RemotePlayerController through its PlayListener interface.

2.3.2.3 Composite

We make extensive use of LibGDX Actor's tree for everything related to display. Our GameScreen contains a stage that will contain the different Actors we wish to display, allowing each group to have its own coordinate system. This implementation also allows us to display other things that aren't Actors in classes such as HealthBarView. Here we draw rectangles independent of actors, while at the same time drawing an actor Label in the same class. Figure 11 below is a class diagram of the view component that shows the diligent inheritance used in conjunction with the Actor class.

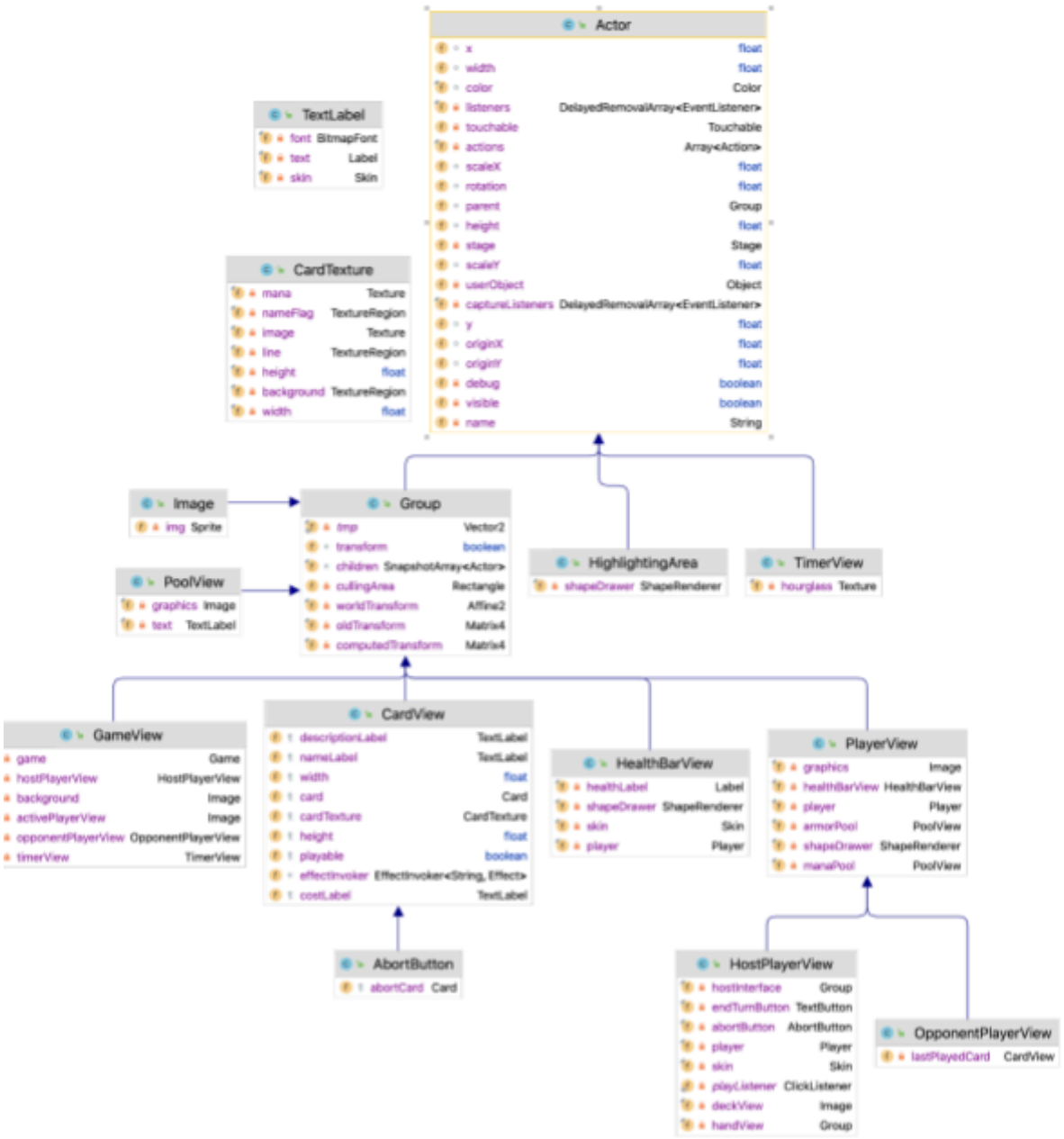

Figure 11 - Diagram of inheritance of gameplay views

2.3.3 Behavioral Patterns

2.3.3.1 Observer

A realization of Observer is the TurnListener and CardPlayedListener interfaces that other classes can implement so as to perform an action when an event occurs. For example, TurnListener is used when an Effect is recurring for additional turns. CardPlayedListener is

implemented in the class that removes cards from the hand when they are played, and on the timer that gets restarted each time a card is played. In each instance, the listeners act as observers of the corresponding TurnEvent and CardPlayedEvent.

2.3.3.2 Template Method

The template method was especially useful when arranging the hierarchy of screens with common render/show methods. One prime example is the BaseScreen along with its children (i.e. ReturnableScreen), where a common background is present.

2.3.3.3 Strategy

To exchange information about cards from one player to another through Firebase, we have to serialize the effects they contain. As such, when a card is played, we serialize all the names of the effects it contains and load them to Firebase as strings. On the receiving end, we fetch those effect names, and to reconstruct the original effect, we use an EffectInvoker object which, given an effects name, returns the effect object so its apply() method is performed on the receiving Player.

3. User Manual

3.1 Requirements

To play this game, you'll need to require one of the following: Android Studio, an Android device with a minimum Android SDK version of 24 (Android 7.0), or Android Studio with an emulator that connects to Firebase. It is important to note that this game is a 1-versus-1, so you'll also need another player to play against.

3.2 Installation

There are two ways to install and play the game, depending on what you are looking for. You can either install the APK directly to your device, as a user would, or run it like a developer through Android Studio. The repository is publicly available here.

3.2.1 APK

1. Clone the repository.
2. Transfer the APK file to your compatible mobile device.
3. Install the application.
4. Enjoy the game!

3.2.2 Project Source

1. Clone the repository.
2. Import the [project folder](./DuelOfFates/) in Android Studio.
3. Run the project on your device or emulator.
4. Enjoy the game!

3.3 Game Functionality

3.3.1 Login screen

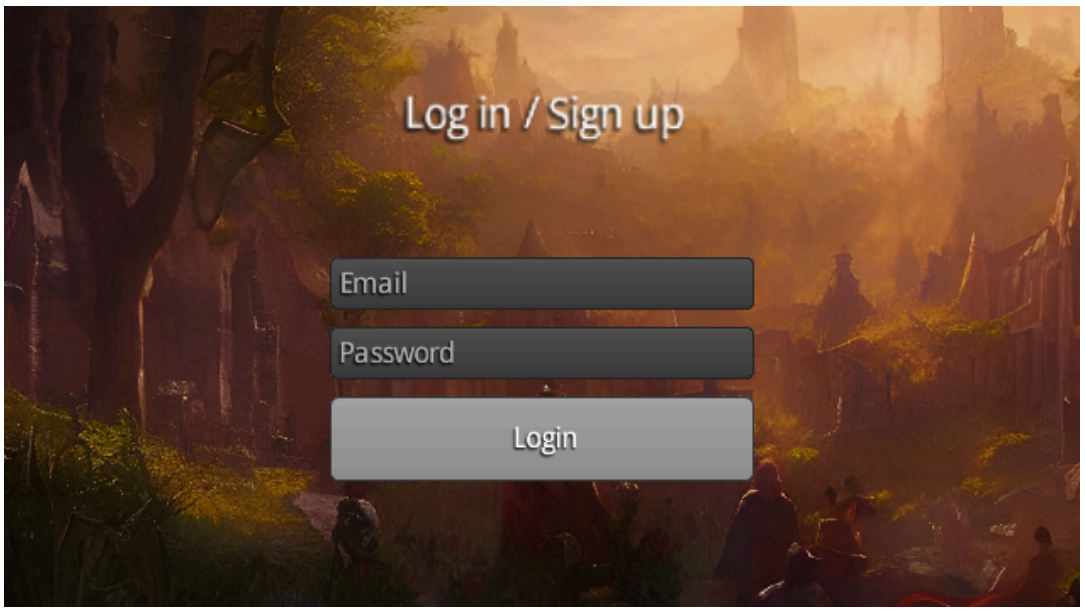

Figure 12 - Login screen

To play this game, you must first log in with your email and password. If you don't have valid login details, you'll need to sign up using your email and password.

3.3.2 Main Menu

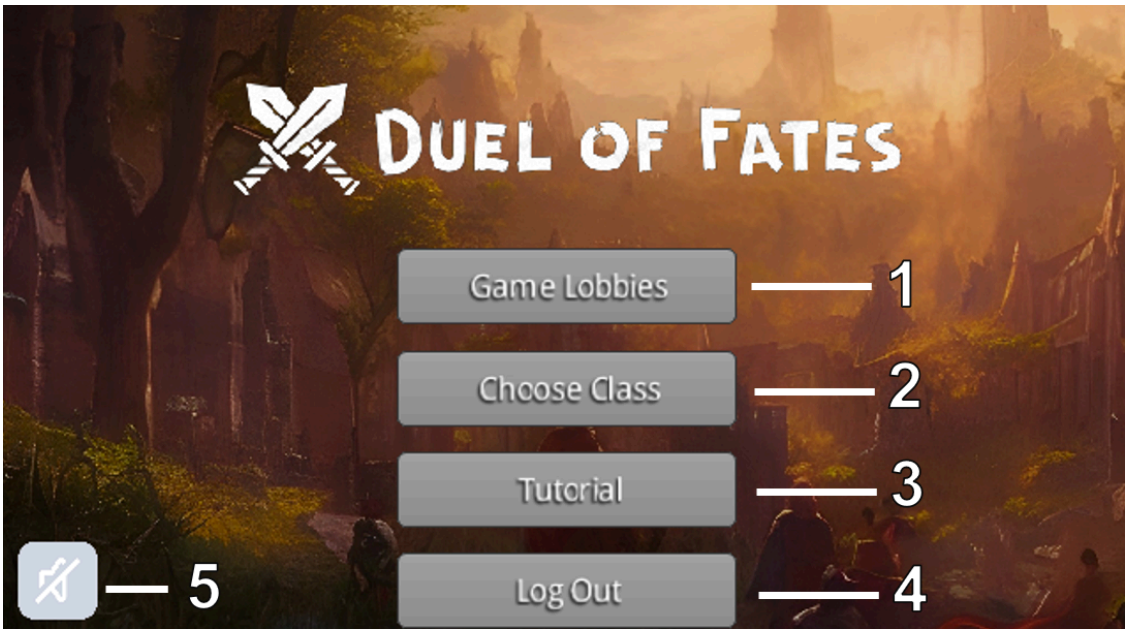

Figure 13 - Main menu

In the main menu, you have different options to choose from. You can navigate to the game lobbies (1) or choose a class (2), with “Knight” automatically chosen initially. You have the option to access the tutorial (3) to learn about the game mechanics, and if desired, you can also mute the game music (5). Additionally, there's a choice to log out (4).

3.3.3 Game Lobbies Menu

Figure 14 - Game lobbies menu

In the game lobbies menu, you can either create (2) or join other available game lobbies (1). You can join a game lobby and play a game against another player. There is also an option where you can see your previous match results in Match history (3).

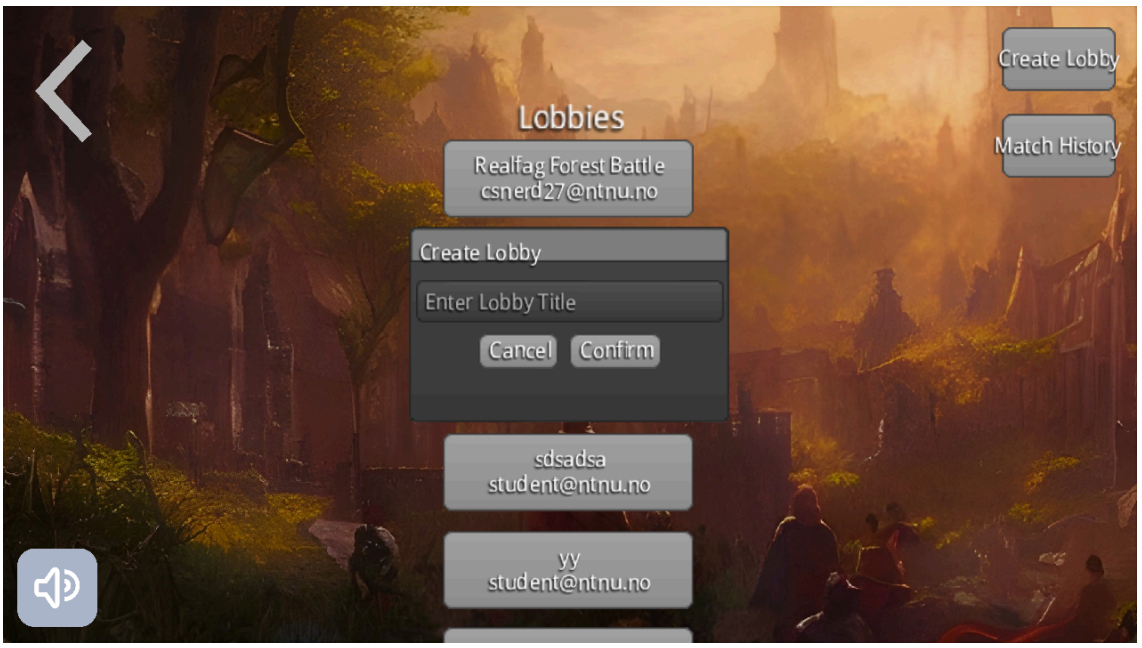

Figure 15 - Create Lobby window

If you decide to create a lobby, you can click on the “Create lobby”-button and a window will pop up giving you the option to enter a lobby name and create the lobby.

3.3.4 Game Lobby

Figure 16 - Inside a game lobby: Guest’s view

In a game lobby, you can see if the lobby is available. With this information, you can either join (1) or exit the lobby (2), returning to the game lobbies menu. If you opt to join, the game will start when the host starts the game.

Figure 17 - Inside a game lobby: Host's view

This is the view from the host, where you can either start the game (1) or delete the lobby (2).

3.3.5 Choose a class

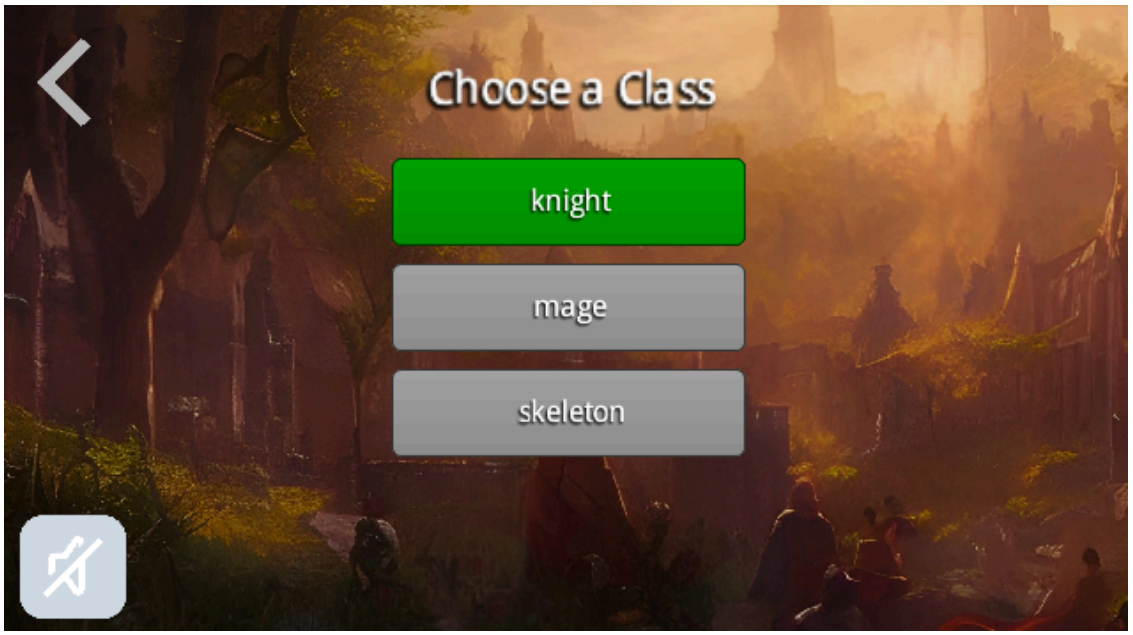

Figure 18 - Choose a Class screen

You can select a different class from the main menu. If you're interested in trying out another class, you can do so here. When you first access this menu, you'll notice three classes available: Knight (initially selected), Mage and Skeleton. You have the option to switch to another class if you prefer.

3.3.6 Gameplay

Figure 19 - Gameplay screen

Once the game lobby consists of two players, the game will start when the host starts the game. Upon entering the game, you'll play with your selected character and your character will have mana (3), health and armor (4). It is your turn when the green arrow (2) is over your character. You will also possess a deck of cards (6), with three available in your hand. The cards cost mana (7), and each card has different effects (8), whether offensive or defensive. You can choose which card to play, and subsequently, you'll receive a new card from your deck to maintain three in hand consistently. The main goal is to reduce the opponent's armor and health points to zero to win. In addition, a timer (9) will end your turn if you haven't finished it yourself (5).

Figure 20 - Game Summary

When the game ends, you will get a window showing whether you won or lost, and the option to exit the game and return to the game lobbies screen.

3.3.7 Match History

Figure 21 - Match History

Within the match history, you can see your past match results. It will display the players' names, identify who the host and guest were, and indicate the winner and loser among the two players.

3.3.8 Tutorial

Figure 22 - Tutorial screen

The tutorial provides basic information on how the gameplay functions, where you're against a tutorial bot. You will get more helpful instructions on how the game works by clicking on the screen. You can leave the tutorial by aborting the match, or play out the game where you will either win or lose.

4. Test report

This section consists of test results for both functional and quality requirements. The requirements maintain their original identity as specified in the Requirements document.

4.1 Functional Requirements

This subsection lists test results for related functional requirements.

FR1: The user must be able to navigate the application with touch gestures.	
Executor:	Dionysios Rigatos
Date:	17/04/24
Time used:	30 seconds
Evaluation:	Success
Comment:	Apart from inputting the credentials during log-in/sign-up, the application can be exclusively navigated using touch gestures.

FR2: The user must be able to join a game match online with another user.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	2 minutes
Evaluation:	Success
Comment:	After finding an available lobby, it was a matter of simply joining it and waiting for the host to start the game. After that, there was a redirection to the game screen where the turn started immediately as soon as both players were loaded in the game.

FR3: The user should be able to choose the player class they wish to play with before a game match starts.	
Executor:	Dionysios Rigatos
Date:	17/04/24

Time used:	1 minute
Evaluation:	Success
Comment:	Upon logging in and being redirected to the Main Menu, a “Class Selection” menu was immediately available where the player classes were displayed and easily selectable.

FR4: A match should occur between two players, each player taking turns in an alternating order.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	8 minutes
Evaluation:	Success
Comment:	After successfully transitioning to the Game from a lobby, it was one of the player’s turns. They were able to play their cards and end their turn, which then allowed the opposing player to play. This alteration occurred until the game concluded.

FR4.1: A player’s turn should automatically end after a defined amount of time.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	2 minutes
Evaluation:	Success
Comment:	Each player has a specific amount of time to play their cards so as to keep the game flow going. As a test, one player idled until their timer run out - and the turn was indeed finalized, with the opposing player starting their turn.

FR4.2: A player should be able to end their turn manually.	
Executor:	Dionysios Rigatos, Márk Somorjai

Date:	17/04/24
Time used:	30 seconds
Evaluation:	Success
Comment:	Upon completing a turn prematurely (before the timer runs out), the player clicked on the “End Turn” button which finalized the round and gave the opposing player their turn.

FR5: A player should have health, defense and mana, as well as cards in their hand.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	1 minute
Evaluation:	Success
Comment:	Upon entering a game or the tutorial, it is immediately apparent that each player has a character sprite that is surrounded by a health bar (and numerical value), an armor icon (and numerical value) and a mana icon (and numerical value). An array of playable cards is available for clicking in their hand (bottom center of the screen).

FR6: The user should be able to view their and the opponent’s available resources (health, armor, mana) when in a match.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	1 minute
Evaluation:	Success
Comment:	Upon entering a game or the tutorial, it is immediately apparent that each player has the opponent player is surrounded by a health bar (and numerical value), an armor icon (and numerical value) and a mana icon (and numerical value).

FR7: The game should deal cards to each player from their decks at the beginning

of the game.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	1 minute
Evaluation:	Success
Comment:	Upon entering a game or the tutorial, it is immediately apparent that each player has the opponent player is surrounded by a health bar (and numerical value), an armor icon (and numerical value) and a mana icon (and numerical value).

FR8: A player should only be dealt cards from their chosen class's deck.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	5 minutes
Evaluation:	Success
Comment:	After a few rounds of gameplay, it was apparent that similar cards were re-dealt at hand refills, all matching the player class' deck.

FR9: During their turn, a player should be able to play all of the cards in their hand.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	2 minutes
Evaluation:	Success
Comment:	If enough mana points are gathered, a player is indeed able to select and play all the available cards in their hand - with the effects being immediately applied on the opponent.

FR9.1: When a card is played during a game, the system should remove the card from the player's hand and add it to a "played card" pile.

Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	3 minutes
Evaluation:	Success
Comment:	As soon as a card is played in a game, it is immediately visible in the stockpile on the bottom-corner of the screen.

FR9.2: When a card is played during a game, the system should modify the player's health, defense and/or mana according to the card's effects.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	3 minutes
Evaluation:	Success
Comment:	As soon as a card is played in a game, if the effects it contains are applicable to ourselves, they are immediately visible on our players health bar/armor/mana.

FR9.3: When a card is played during a game, the system should modify the opponent's health, defense and/or mana according to the card's effects.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	3 minutes
Evaluation:	Success
Comment:	As soon as a card is played in a game, if the effects it contains are applicable to the opponent, they are immediately visible on their player's health bar/armor/mana.

FR9.4: A player should only be able to play a card if it doesn't reduce their mana below 0.

Executor:	Dionysios Rigatos, Márk Somorjai
Date:	17/04/24
Time used:	2 minutes
Evaluation:	Success
Comment:	When it's a player's turn and their hand is visible to them, the cards that are playable (can be afforded with the current mana, and the card's mana cost) are highlighted and click-able. Cards that are not available are lowered and cannot be clicked (played).

FR10: The system should fill up the upcoming player's hand to the maximum allowed amount of cards from their deck at the beginning of their turn.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	18/04/24
Time used:	4 minutes
Evaluation:	Success
Comment:	At each start of a player's turn in a game, irregardless of how many cards were played, the hand was indeed re-filled from the deck - up to 3 cards (current maximum).

FR11: If the player's deck runs out of cards, the system should re-shuffle the player's played card pile and repurpose it as a drawing pile.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	18/04/24
Time used:	7 minutes
Evaluation:	Success
Comment:	Indeed, after going through a deck of cards (currently 16 cards, which took a few turns), cards that were initially in the deck but had already been played were back in the rotation.

FR12: The system should refill the player's mana by a certain amount of points at

the beginning of their turn.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	18/04/24
Time used:	3 minutes
Evaluation:	Success
Comment:	After the first turn concludes, and the opponent concludes as well, when the turn returns to the initial player their mana is refilled by 3 mana points.

FR13: If the player has no more health, they should lose the game.	
Executor:	Dionysios Rigatos, Márk Somorjai
Date:	18/04/24
Time used:	7 minutes
Evaluation:	Success
Comment:	Upon the completion of a lot of turns, and as soon a player's health was reduced to 0, the game indeed concluded with “You Lost” message for that player.

FR14: The user should be able to turn on/off sound and/or music.	
Executor:	Dionysios Rigatos
Date:	18/04/24
Time used:	30 seconds
Evaluation:	Success
Comment:	Immediately upon logging in, a sound icon is visible in the bottom left. Clicking it stopped the music immediately.

FR15: The user should be able to forfeit the game.	
Executor:	Dionysios Rigatos
Date:	18/04/24

Time used:	4 minutes
Evaluation:	Success
Comment:	During a game match, it is possible to forfeit a match and exit the game using the white flag button at the top right, which immediately makes the player lose.

FR16: The user should be able to view their game history and the outcome.	
Executor:	Dionysios Rigatos
Date:	18/04/24
Time used:	2 minutes
Evaluation:	Success
Comment:	Upon entering the Lobbies Menu, a Match History button is available in the top right - listing all the previous games as well as their outcome.

FR17: The user should be able to initiate a tutorial game as many times as they wish, from the main menu.	
Executor:	Dionysios Rigatos
Date:	18/04/24
Time used:	8 minutes
Evaluation:	Success
Comment:	Immediately in the main menu, a Tutorial button is available. Upon clicking it, the tutorial is initiated. The tutorial can either be completed or forfeited (ended prematurely), and then the user is redirected to the main menu. Then, it can be re-initiated at any time as it is still accessible.

FR18: The user should be able to enter and exit a lobby created by other users, or create and delete their own lobby.	
Executor:	Dionysios Rigatos
Date:	18/04/24

Time used:	5 minutes
Evaluation:	Success
Comment:	Upon entering the lobbies menu, the user is able to join available lobbies, and able to exit them before the game starts. Likewise, if the user creates a lobby, they can either delete it (and subsequently exit) or start the game when a guest user has also joined the lobby.

4.2 Quality Requirements

This subsection lists test results for related quality requirements.

M1: Add new effect	
Executor:	Dionysios Rigatos
Date:	19/04/2024
Environment:	Design time
Stimuli:	Add new card effects to the game
Expected response measure:	Within 30 minutes
Observed response measure:	10-25 minutes
Evaluation:	Success
Comment:	Adding a new effect was as simple as creating its class and overriding the apply() method with the desired functionality. Depending on how sophisticated the effect is, this can range from 5 to 20 minutes. Finally, in order to make the effect usable, it has to be added to the EffectModule, which simply requires the creation of a dagger Provides method that initializes the effect and its attributes using a builder, then adds the effect to the EffectInvoker map - all of which takes approximately 5 minutes. The effect is then ready to be used in cards, and the game may call it through the invoker if it is ever applied. The process seems to be as simple - creation and then initialization in one place makes the effect immediately available to the whole application.

M2: Modify effect	
Executor:	Dionysios Rigatos
Date:	19/04/2024
Environment:	Design time
Stimuli:	Change the attributes of an existing effect in the game
Expected response measure:	Within 10 minutes
Observed response measure:	5-10 minutes
Evaluation:	Success
Comment:	Modifying existing effects attributes/values is as simple as editing the builder attributes. Changing the apply method itself is a bit more tricky and might require more time if something sophisticated is to be implemented - however this is not related to the architectural choices rather the actual complexity that would exist irregardless. Builder initializations in dagger modules make adjustments extremely easy, and the common interface for the apply method makes the whole process straightforward.

M3: Add new card	
Executor:	Dionysios Rigatos
Date:	19/04/2024
Environment:	Design time
Stimuli:	Add new card to the game
Expected response measure:	Within 30 minutes
Observed response measure:	10-15 minutes
Evaluation:	Success

Comment:	Adding a new card is a matter of initializing its Provider method with the CardBuilder with the correct effects as well as giving it a cost and name. Adding it to decks requires simply adding it to the respective deck list. The process seems to be as simple - creation and then initialization in one place makes the effect immediately available to the whole application. If one wishes to make cards without using pre-existing effects, then the time cost should be accumulated with M1/M2's.

M4: Modify card	
Executor:	Dionysios Rigatos
Date:	19/04/2024
Environment:	Design time
Stimuli:	Change the properties of an existing card in the game
Expected response measure:	Within 15 minutes
Observed response measure:	5-10 minutes
Evaluation:	Success
Comment:	Modifying existing card attributes/values is as simple as editing the builder attributes in the provide methods. If, for a card, you wish to add new non-existing effects as well, this accumulates the time cost with M1.

M5: Add new deck	
Executor:	Dionysios Rigatos
Date:	19/04/2024
Environment:	Design time

Stimuli: Expected response measure: Observed response measure: Evaluation: Comment:	Add a new deck of cards to the game Within 45 minutes 10-20 minutes Success Adding a new deck is only a matter of creating a new Provider method and giving it a name, as well as filling the new deck with the desired cards, if pre-existing cards are used. If you wish to create new cards for the new deck, the time cost accumulates with M3.

M6: Modify deck	
Executor:	Dionysios Rigatos
Date:	19/04/2024
Environment:	Design time
Stimuli:	Change the content of an existing deck
Expected response measure:	Within 20 minutes
Observed response measure:	10 minutes
Evaluation:	Success
Comment:	Modifying a deck is a matter of changing the cards that are added to it (a deck is a list) in the provider method in its module.

M7: Add new PlayerClass	
Executor:	Dionysios Rigatos
Date:	19/04/2024
Environment:	Design time
Stimuli:	Add a new Player Class to the game
Expected response measure:	Within 1 hour
Observed response measure:	10-40 minutes
Evaluation:	Success
Comment:	Adding a new Player Class is a matter of defining its special attributes and assigning a deck to it as well as its textures. Depending on what will be re-used from decks/cards/effects, this can be done as quickly as within 10 minutes.

M8: Modify PlayerClass	
Executor:	Dionysios Rigatos
Date:	19/04/2024
Environment:	Design time

Stimuli: Expected response measure: Observed response measure: Evaluation: Comment:	Change the attributes of an existing PlayerClass Within 30 minutes 5-20 minutes Success Modifying an existing player class is a matter of adjusting its values in the initialization builder in its respective provider method in its module. Adding a new existing deck is also a matter of changing the initialization. Creating a new player class with entirely new decks, cards and effects, accumulates the previous costs.

M9: Modify existing graphics	
Executor: Date: Environment: Stimuli: Expected response measure: Observed response measure: Evaluation: Comment:	Adrien Rosset 19/04/2024 Design time Modify the appearance of a card or a player class Within 1 hour 5 min, 1 hour Success 5 minutes for just changing the image display for an entity 1 hour to make more complex changes, like adding an image or combination of images to an element.

U1: Learning the game	
Executor: Date:	Herman Vellene (External User Test) 19.04.2024

Environment:	Runtime
Stimuli:	Play the tutorial for the first time
Expected response measure:	Completes the tutorial within 10 minutes
Observed response measure:	4 minutes
Evaluation:	Success
Comment:	The executor was highly motivated to defeat the opponent, which is the longest part of the tutorial

U2: Enter a game	
Executor:	William Marleau, Dionysios Rigatos
Date:	19.04.2024
Environment:	Runtime
Stimuli:	Enter a game
Expected response measure:	Enter a game within 2 minutes
Observed response measure:	Users created, joined, and started the game in 30 seconds
Evaluation:	Success
Comment:	Assumes a guest joins the lobby one has created

U3: Modify game settings	
Executor:	William Marleau
Date:	19.04.2024
Environment:	Runtime
Stimuli:	Modify a game setting (e.g. Toggling the music ON/OFF)
Expected response measure:	Within 10 seconds, the user has toggled the setting they wish and the change has been applied
Observed response measure:	User toggled the sound setting in 3 seconds and the change was applied instantaneously
Evaluation:	Success
Comment:	

5. Relationship with Architecture

5.1 Architectural Patterns

5.1.1 Model-View Controller (MVC)

Even though we tried to stick to the model view controller architecture thoroughly, we were forced to deviate from it at some points. For example, the TutorialScreen and TutorialController have a stronger cohesion, because we need a way for the controller to indicate when to go from one part of the tutorial to the other. This could not have been discovered earlier because it’s a consequence of our interactive tutorial which had not been exposed before in the project.

5.1.2 Entity Component System (ECS)

We develop the game with an Entity component system architecture but we don't really make full use of it. Statistics such as Health and Armor aren’t really reused for active entities. And the Effect list for cards is reused because we can add multiple effects to the same card but are not reused in other entities.

5.1.3 Shared-Data

Though we utilize a shared-data pattern with a common state for game lobbies through the Firebase Realtime Database, the degree to which we use it for gameplay is limited. During gameplay, only the state of whose turn it is, as well as the particular cards played is stored in a shared-data pattern. In order to leverage this pattern to its full extent, we could have stored the state of the entire game model itself in the database, as a shared source of truth. We decided not to do that in order to use the minimal amount of data necessary for correct communication.

5.2 Views

5.2.1 Physical View

While the deployment diagram of our architecture illustrates communication specifically between Android devices - through Firebase - we have also implemented cross-platform functionality for desktop. Early on in the project, due to limitations in equipment, we found it

slow to properly test our code using Android emulators. To save time in the long-run we started implementing Firebase functionality for desktop. This suited well with our architecture, utilizing interfaces for services and LibGDX's native setup for multi-platform implementation. With Firebase's easy integration across platforms and serving as a source of Shared-Data, we ended up with implementing full functionality for cross-platform gameplay across desktop and Android.

6. Problems, issues and points learned

This project was challenging. In preparation for the project, we had to understand how to make good architectural decisions and implement this for a game application, which should consist of a relatively complex software architecture. In addition, we had to learn to develop in Android Studio using the LibGDX framework.

6.1 Memory Disposal

We learned that memory disposal is an important task and that it needs to be done very carefully even for little elements of the game especially for graphics elements that don't get deleted by garbage collectors or else the game will end up crashing due to lack of memory.

6.2 Group workload distribution

We encountered a significant issue during the development process. Even though the group members come from a somewhat similar educational background, we have different skills and experiences. Therefore, it was hard for everyone to contribute in the same way, resulting in some group members having huge workloads. Furthermore, it was difficult to coordinate as a group when people were unavailable and unreliable, resulting in less overall effort from these people.

To address these issues, we could have approached the project differently. Since the group members were unfamiliar with each other from the start, it would be beneficial to have a skills assessment, to align with each other's strengths and expertise. With this assessment, it could have improved the delegation of tasks. We could also have better communication by increasing the number of our meetings, thereby facilitating more regular updates, which could have resulted in a better workload distribution throughout the whole project. However,

due to our different schedules and commitments to other courses, scheduling additional meetings proved challenging.

As the project progressed, we divided the group in pairs in order to focus on different aspects of the implementation in groups of two. This helped address some knowledge gaps among group members. Delegating the work this way helped us conduct quick communication within the pairs, and made it easier for all members to keep track of what was being worked on.

6.3 Project Management

The early stages of our project lacked clear ground rules for project management. Our development tasks were distributed with set deadlines. However, some of the tasks were interdependent, hindering the process. In addition, we adopted a spontaneous approach by underestimating the workload, resulting in messy code and necessary refactoring cycles. If we used another project management we could have laid down essential rules and a clearer project structure. Reflecting on the challenges faced, establishing a clear project structure from the beginning was difficult, since it wasn’t clear what our game concept and structure would look like. Our approach was very time-consuming, and it was not ideal for this group project. Concerning this problem, we should have used another project management method to find a clearer vision of how to realize our game and plan accordingly.

7. Individual Contributions

7.1 Implementation Report

Member	**Part of the document**	**Approximate Hours**
Adrien Rosset	Design and Implementation: Model-View-Controller, Tactics Relationship with Architecture Quality Requirement : M9	7h
William Marleau	Structure, Design and Implementation: Service-Oriented Architecture, Tactics, Patterns Test Report	7h
Ken Nguyen	Introduction, Structure, User Manual, Patterns	7h
William Hassel	Patterns, Problems, issues and points	7h
Dionysios Rigatos	Introduction, Design & Implementation Details, User Manual, Test Report, Relationship w/ Architecture	9h
Márk Somorjai	Class diagrams, Design and Implementation: Tactics and Patterns	9h

7.2 Project Contributions

Contributions to the project excluding contributions to the project reports, as they are listed in their respective sections.

Member	**Contribution**
Adrien Rosset	Worked mainly on the majority of the view package (all of the view.gameplay package) as well as screens (screen package). Implemented the tutorial throughout with the respective controllers, logic, views and setup. Added main assets as well as their handling. Finally, main contributions to the game video.
William Marleau	Started with login functionality and implementing connections with Firebase. Made the structure for service interfaces. Implemented controllers and model-view-controller logic for most application related screens (i.e. login/menu/lobbies). This involved listeners for fetching lobbies and lobby info in addition to adapting the lobby screen based on being a host or guest. Furthermore, I implemented the screen and functionality for choosing classes. Implemented most functions for communicating with Firebase on the application (not gameplay) level.
Ken Nguyen	Worked on the view of game history, showing the previous matches of the current user and how it should be displayed.
William Hassel	Worked on the user interface. Set up libGDX assets. Stylized and scaled screens, ensuring proper rendering on Android and Desktop.
Dionysios Rigatos	Initially worked on the Controller and Model parts of the application, implementing the main game logic entities (i.e. all of model.gameplay, some of model.communication classes) as well as the actual gameplay (all of controller.gameplay classes excluding PlayerController). Implemented Dagger Modules/Components throughout (di

	packages), as well as handled initialization of all Game Entities (effects, cards, decks, playerclasses). Contributed to the serialization of objects for Firebase compatibility. Added minor functionalities in FirebaseServices when it comes to lobbies and their state update. Implemented Sound throughout the screens in coordination with other team members. On the View part, contributions to the handling of the screens throughout (their orchestration for viewing/disposal with ScreenController). Memory leak management, major debugging. Finally, contributions to the presentation video.
Márk Somorjai	Designed and implemented the gameplay Model and Controllers, including all events and their listeners, the way effects work, all player controllers, the protocol design and Firebase implementation of communication between clients during the game and the MVC architecture. Created the timer and abort parts of the game view and connected the whole of the game view to the game logic. Worked on sound and its button working properly across screens. Patched a bunch of memory leaks across screens and fixed bugs. Implemented desktop Firebase services. Worked on the integration of different screens in the application, such as initializing and starting games on the lobby-to-game transition or returning and uploading the game summary on the game-to-lobbies transition. Implemented the persistence of user's selected classes in Firebase.

8. References

[1] Slay the Spire by Mega Crit Game on Steam, https://store.steampowered.com/app/646570/Slay_the_Spire/

[2] LibGDX. 2024. *Interfacing with platform specific code*. Retrieved from https://libgdx.com/wiki/app/interfacing-with-platform-specific-code