NTNU
Norwegian University of Science and Technology

Duel of Fates

Architecture

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 Architectural Phase Description

This document acts as the architectural specification of the project. We build upon our architecture from the requirements document - giving structure to the project. The Architectural Significant Requirements and the Architectural Drivers will define the architecture. Its specification and documentation are achieved through a variety of architectural patterns and tactics - described using the 4+1 Model and modeled with UML.

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. Architectural Drivers / Architecturally Significant Requirements (ASRs)

2.1 Functional

2.1.1 Turn-Based Multiplayer

The primary focus of the functional requirements revolves around enabling users to engage in a turn-based multiplayer game online. Achieving this requires implementing communication between clients. For this, in addition to other purposes, Firebase is set to act as the intermediary.

2.1.2 Different Cards

The player will have the option to select from a variety of cards for both offensive and defensive actions. Consequently, we will ensure the capability to incorporate new cards and adjust card interactions within the architecture.

2.1.3 Predefined Classes

The player should be able to choose between a selection of different predefined classes. Each class will feature its unique combination of cards, providing players with diverse gameplay experiences based on their chosen class.

2.1.4 Resource Management

The player will initially start with a predetermined amount of health, armor and mana. Successfully managing these resources will be crucial for victory, as reducing the opponent's health to zero signifies winning.

2.2 Quality

2.2.1 Modifiability

Modifiability is the primary quality attribute for our game, and is emphasized in our architecture. The architecture should facilitate effortless addition or removal of cards, features and adjustments.

2.2.2 Usability

As a secondary quality attribute, we have chosen usability. The game should prioritize easy comprehension and gameplay. For this, we plan to implement a tutorial. Additionally, the game should give feedback on the players’ choices and actions.

2.3 Business

The main objective is to learn through designing a well-thought-out architecture and develop an engaging game. Given the time constraints, the goal is to make a complete game that fulfills the requirements. This implies that the architecture should be efficient to ensure timely completion. Another important objective is to meet all assignment requirements, as well as

the deadlines for implementation and documentation. Therefore, our initial priority will be to establish essential functionality, with a commitment to ensuring a high degree of modifiability for effortless addition of new content. The team members are excited about collaborating on the development of the game and learning more about Android development, LibGDX and Firebase.

3. Stakeholders and Concerns

The following table describes the main stakeholders and their interests in the project.

Stakeholder	**Concern**
Developer	The code should be well-written making it easy for cooperation and task delegation. Prioritizing good modifiability is important, as it facilitates effortless changes or additions to the source code.
Course Staff	The code should be implemented for readability and be thoroughly documented. The game should also meet all the requirements and have a well-designed architecture. The documentation should be of high quality, demonstrating a clear understanding of the selected architectural patterns and our decisions. The game should also be easy to set up, to avoid unnecessary problems.
End-Users	The game should be easy and interesting to play. The user has an option to complete a tutorial before starting to play. The game should be intuitive to play and should give feedback on the user's actions. End-user is interested in the architectural logical view, which provides information on game features and functionalities.
ATAM evaluators	The architectural documentation should be written in a clear and comprehensible manner. The evaluators will require essential information, thus it is crucial to ensure that no important details are overlooked.

Table 1 - Stakeholders and Concerns

4. Architectural Viewpoints

View	Purpose	Stakeholders	UML Model(s)
Logical View	Exhibit features of the system in a static way.	Developers, Project Evaluators, End-User	Class Diagram
Process View	Exhibit program flow for use case extraction.	Developers, Project Evaluators	Activity Diagram, Sequence Diagram, Communication Diagram
Development View	Ease development by showing the technical components of the system as well as their interactions.	Developers, Project Evaluators	Package Diagram
Physical View	Exhibit the mapping between software and hardware/networking components of the system.	Developers, Project Evaluators	Deployment Diagram

Table 2 - Overview of Architectural Viewpoints

5. Architectural Tactics

5.1 Modifiability Tactics

To effectively or efficiently modify the system without introducing defects or degrading existing quality, we can apply a set of known tactics. The purpose of a *modifiability tactic* is to control the complexity, time, and cost of making changes to a system [3]. In our implementation of *Duel of Fates*, we seek to utilize tactics that reduce coupling, increase

cohesion, and defer binding. Each of these is estimated to reduce the cost of future modification of the system.

5.1.1 Reduce Coupling

Coupling is how much a module is directly connected to another one. When two modules are tightly coupled it is difficult to modify or replace either of them as issues would be delegated to the partner module. Ideally, we want to reduce coupling between our modules.

5.1.1.1 Intermediaries

To avoid direct coupling between screens we seek to utilize an intermediary which can handle the intricacies of preparing resources and navigating to a new screen. Avoiding direct coupling in this way will allow us to centralize potential modifications, reducing the time, cost, and complexity of modifications compared to if all screens contained the functionality themselves.

5.1.1.2 Abstraction of Common Services

By abstracting common services, we seek to get more consistency when handling infrastructural issues. This can be achieved by using abstract classes and as in any mobile application, a lot of entities and components contain common attributes when abstracted (i.e. IDs) and thus allow for consistency when utilizing them in modules. Most importantly, in a well-defined hierarchy, changes on specific levels only affect relevant inheritors - thus allowing for module and system-level modifiability.

5.1.1.3 Encapsulation

Encapsulation refers to the bundling of related data and attributes into a unit while restricting direct external access to its resources. Interfaces excel at providing that as they allow for modules to communicate using a specification - like an API - thus hiding the implementation away from other modules while still allowing them to access certain methods. This will be especially helpful when creating services and controllers - allowing for easy exchange between different types throughout our implementation.

5.1.2 Increase Cohesion

Cohesion refers to how things in a module are coherent to be together. By having high cohesion, a module limits communication with other modules and it also allows to centralize everything that is related to a certain point and so limits the number of changes needed.

5.1.2.1 Split Module

Split module is a tactic where a larger module is split into smaller more manageable and independent units. A key aspect of our code where we seek to implement this is networking, where communication to the database is split into two or more finer modules. Furthermore, we seek to utilize this tactic on the game logic itself, which in its totality, is estimated to be relatively complex.

5.1.3 Defer Binding

Deferring binding supports modifiability by bounding decisions at specific times throughout the pipeline - specifically by minimizing the number of modules that require changing after implementing a new feature or modification. We aim to achieve this by utilizing the following tactics.

5.1.3.1 Compile-Time Parameterization & Configuration-Time Binding

Configuration-time binding refers to deferring the initialization of objects on startup time. Compile-time parameterization refers to binding values during compile time - such as creating a dependency graph. This way objects are created in a reusable manner and the dependencies can be evaluated to avoid cycles and runtime initialization issues, which will be especially useful in scenarios where we instantiate objects in a specific way (i.e. cards, effects etc.) or injecting dependencies (i.e. strategy invokers).

5.1.3.2 Polymorphism

Polymorphism is a tactic where values are bound at runtime. It allows late binding of method calls, allowing for flexibility in program execution. Moreover, polymorphism lets us define a single operation with different implementations in subclasses. The decision on which implementation to execute is made at runtime based on the object’s type. Due to the variety of game entities that share functionality (i.e. attributes), its certain polymorphism will come in handy.

5.1.3.3 Resource Files

Using resource files can further enhance modifiability by extracting configurations or assets into separate, easily accessible and replaceable entry points. In our case, we could utilize it to make replacing in-game graphics - such as player sprites or the background - as easy as swapping a PNG file with another one of the same name. It could also be helpful with specifying cards and decks in resource files so that they can be configured by non-technical people without having to understand the code.

5.1.3.4 Shared Repositories

Shared repositories refer to the blackboard pattern - where information can be stored and accessed by authorized stakeholders when needed. This can be achieved through the use of Firebase's Realtime Database, which offers flexible data synchronization capabilities, making it easier to modify and update data in real time across multiple clients such as active lobbies - which will always stored in the respective table and fetched by users when they wish to find an available lobby.

5.2 Usability Tactics

"Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support that the system provides." [4]. The tactics can be split into support user initiative and support system initiative. Usability testing is crucial for ensuring that systems support user actions and provide a smooth user experience. It assesses how easily users can complete tasks and the level of system support. Since our main quality attribute is modifiability, we can use usability testing to identify and fix issues with relative ease, to ensure the system meets user needs and delivers a friendly experience. We plan to use the following tactics.

5.2.1 Support User Initiative

5.2.1.1 Undo

Being able to reverse actions is extremely important in a usable UI as misclicks can occur and one might change one's mind. Our target is for an easy-to-use GUI that enables users to take initiative within the system when navigating through screens. One should be also able to log out of the game and forfeit matches.

5.2.1.2 Cancel

In applications where users can create entities and resources, it is important to be able to cancel their creation - freeing resources - as well as notify any other users who are reliant on them. We seek to use this tactic on matchmaking lobbies which may contain two users simultaneously and be deleted at any time by the host. In this case, the guest user should be informed of the deletion and redirected to the appropriate screen. Additionally, the lobby resources should be freed as they are obsolete and inaccessible.

5.2.1.3 Pause/Resume

This tactic implies that the user should be able to pause and resume long-running operations. By having these options available, resources can be temporarily freed so that they can be reallocated to other tasks. In our case, it can be implemented in screen flow where the navigation of screens happens. When the user goes to another screen, the stack of screens present should be paused to prevent unnecessary resource consumption.

5.2.2 Support System Initiative

5.2.2.1 Maintain User Model

For this tactic, the system should maintain “the user’s knowledge of the system, the user’s behavior in terms of expected response time, and other aspects specific to a user or a class of users.” [2]. For our project, the system should be able to save the user’s chosen class selection, ensuring that even after logging out and logging back on potentially a different device, the latest class selection is preserved. Saving the class preference will enhance the user experience since the user wouldn’t need to change to their preferred class every time they log in to play.

6. Architectural & Design Patterns

6.1 Architectural Patterns

6.1.1 Service-Oriented Architecture (SOA)

We seek to integrate a Service-Oriented Architecture pattern to enhance modifiability and modularity while also easing development complexity. SOA is a pattern that focuses on

discrete services rather than monolithic design. Such a service is usually recognized by being self-contained, a black-box for its consumers, and representing a repeatable business activity. By utilizing Firebase's Backend as a Service (BaaS) solution, we seek to leverage loosely coupled services, such as authentication, game state hosting, and user data persistence.

6.1.2 Model-View-Controller (MVC)

Model-View-Controller is a commonly used architectural pattern, as it promotes modularity and makes it easier to maintain and manage the codebase. In our application, it will provide a way to separate the business logic from the view, while being mediated by a controller. An example of this would be a player's client receiving an update from Firebase's Realtime Database, the controller sending this update to the model, whose change of state is reflected in the view.

6.1.3 Entity Component System (ECS)

The Entity Component System (ECS) is an architectural pattern that decouples logic and data by organizing gameplay and system functionality into entities and components. Implementing ECS in our game will involve defining entities for game objects like players and cards, where the entities have common components representing attributes such as mana cost, damage, and health effects.

6.1.4 Publish-Subscribe

The Publish-Subscribe pattern allows objects to subscribe to information updates from other objects, without the former *subscriber* objects having to depend on the latter *publisher* object. This can be done through an intermediate channel/event to which subscribers register themselves as listeners, and to which the publisher can provide updates for distribution. This pattern could be useful for notifying each client of changes in the game state. For example, when a player plays a card, the game server could publish the played card through an event to inform all subscribers, so that they can update their views and game states accordingly.

6.1.5 Shared-Data

As communication and state updates between clients in our multiplayer game will be pivotal, we seek to utilize a Shared-Data pattern by using Firebase's Realtime Database. The pattern revolves around the exchange of data between multiple accessors and at least one shared-data store [5]. Utilizing this pattern will enable us to establish a common state for lobbies and gameplay, where shared data can be used to update each client accordingly.

6.2 Design Patterns

6.2.1 Creational Patterns

6.2.1.1 Singleton

To ensure consistency in the application's state, we seek to utilize the Singleton pattern, ensuring that a class has only one instance and provides a global point of access to it. Among the classes, we seek to utilize this pattern on management objects that are static throughout the application, such as a controller for sound.

6.2.1.2 Builder

The Builder pattern will allow us to produce different types of objects and representations of an object by utilizing the same construction code. This can be particularly useful for creating game entities with different components.

6.2.2 Structural Patterns

6.2.2.1 Facade

The Facade pattern is a simplified interface to a complex subsystem, which will allow us to improve usability and readability in the codebase by hiding the subsystem's complexity. Through implementing simple methods such as for starting a game, playing a card and ending a turn, we will abstract away the complexities of database updates, state synchronization, and game logic processing.

6.2.2.2 Adapter

The Adapter pattern can be used to hide the discrepancies between different interfaces to the interface's user. It can be especially useful when using other libraries with predefined interfaces that we need to adapt to our code. It is most likely that we will utilize this pattern to adapt network communication to our game updates, by adapting our generic message-sending methods to Firebase, or to be able to register our update listeners to Firebase's events through adapters.

6.2.2.3 Composite

The composite pattern is a way to structure data into trees that are not necessarily binary, commonly used in UI frameworks to allow the creation of element groups. It can also help with the z-order of elements on the screen, as well as with input handling. Since our chosen graphics library LibGDX also makes use of this pattern with its actors and actor groups [6], we aim to implement some of our own individual and group objects in rendering, for example for players - in the former case - and the view of card hands in the latter one.

6.2.3 Behavioral Patterns

6.2.3.1 Observer

The Observer pattern allows objects to subscribe to event notifications from other objects, promoting modularity and decoupling. The pattern will be useful for notifying each client of changes in the game state. For example, when a player ends their turn, the game server would notify all subscribed clients, ensuring they update their views and game states accordingly.

6.2.3.2 Template Method

The Template Method pattern allows for code reusability by overriding methods that encapsulate common behavior that has to be enriched by subclasses. We will likely be using it with screens, as they usually show common features (i.e. background, buttons) and become more specific depending on the application's progress.

6.2.3.3 Strategy

The Strategy pattern helps developers manage how certain objects perform their tasks when required to do so, by defining a common interface that can be implemented by all the actors. In the case of a mobile game, this can be especially useful when different game entities

perform the same type of action differently. One example would be how an array of different effects can be applied on the target player - each being unique, but all being applied in the same way.

7. Architectural Views

The 4+1 Architectural View Model is a methodology for describing software architectures in a systematic way. This model provides a structured approach for analyzing, designing, and documenting software systems. It consists of four main views that capture the functional and non-functional requirements of the system, along with a fifth view that integrates the other four views [2]. The four main views within the 4+1 are the logical view, the process view, the physical view, and the development view.

7.1 Logical View

7.1.1 Model-View-Controller Class Diagram

The logical view describes the system’s functionality from a user's perspective and focuses on the system's components and their interactions. Figure 3 shows a class diagram displaying a simplified frame of the model-view-controller architecture. The reason for not displaying every class we can think of is that for one, there are simply too many classes for one diagram to go into detail, but also because at this point we cannot be sure exactly how every detail is going to be implemented - that will be available in the actual implementation document.

The model package contains classes that will implement the core gameplay. The view package is responsible for displaying the state of the model, while the controller package handles inputs and network communication.

The classes in gray are from our chosen graphical library, libGDX. Inheriting these Actor and Group classes will allow us to utilize libGDX’s composite pattern to help us with rendering and input handling.

The view and controller packages depend on each other, both depend on the model package, while the model itself does not depend on either of the others.

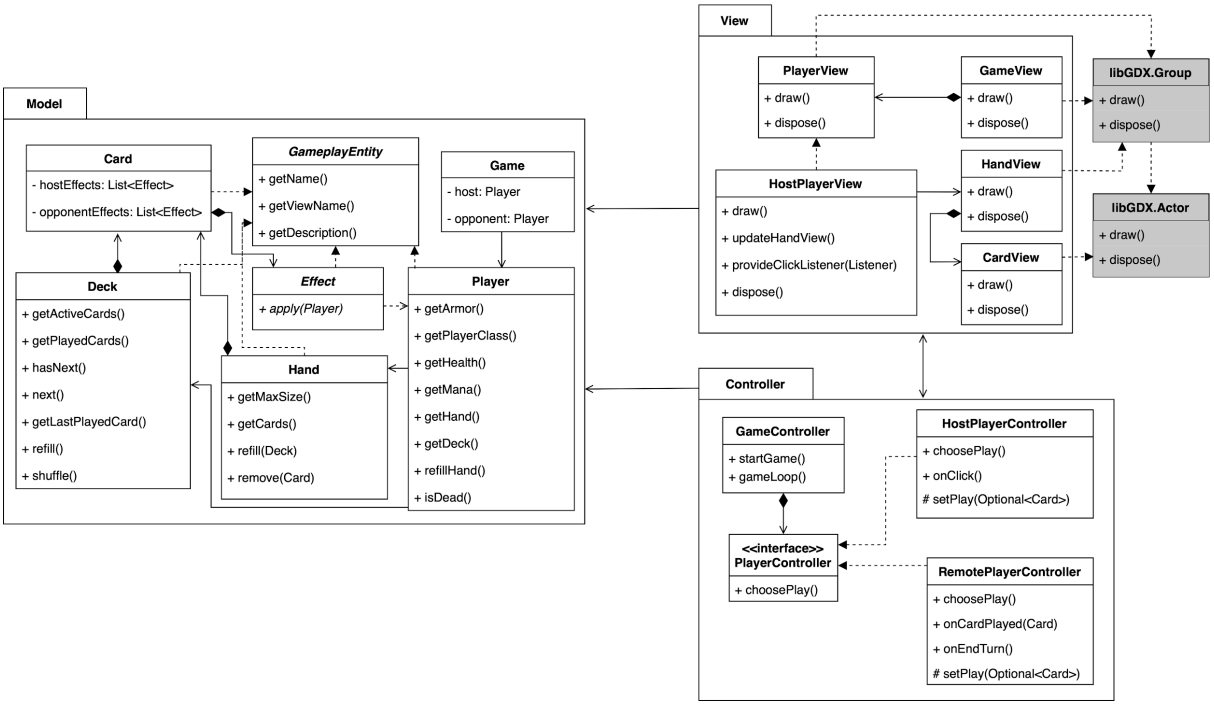

Figure 3 - Logical View Class Diagram

7.1.2 Entity-Component System Class Diagram

For the ECS diagram, the relationships between Entities and the Components that make them are highlighted, as well as the Systems that handle the aforementioned Entities. In our case, ECS will come in handy for handling game entities such as Effect, Card, Deck, PlayerClass and Player - which will then be handled in their controllers, as we also follow MVC. The colors in the arrows do not serve any special role other than making them more distinctive.

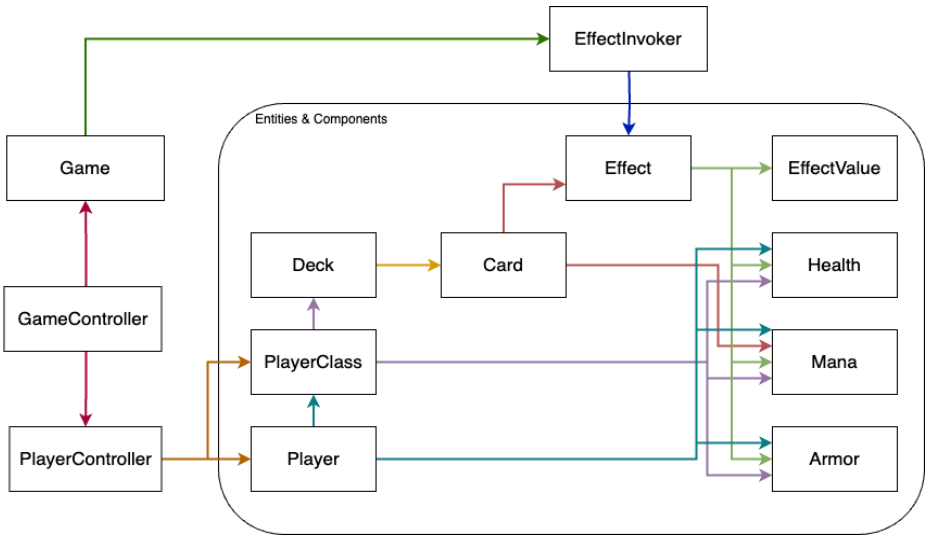

Figure 4 - ECS Class Diagram

7.1.3 Publisher-Subscriber Class Diagram

For the Publisher-Subscriber diagram, we show how the Player (Publisher) notifies all of its Listeners (Subscribers) through the GameEvents channel to have a platform for recurring events throughout the game. This way, we will be able to implement lasting effects (e.g. a PoisonEffect), with the effect also acting as an Observer of the corresponding TurnEvent.

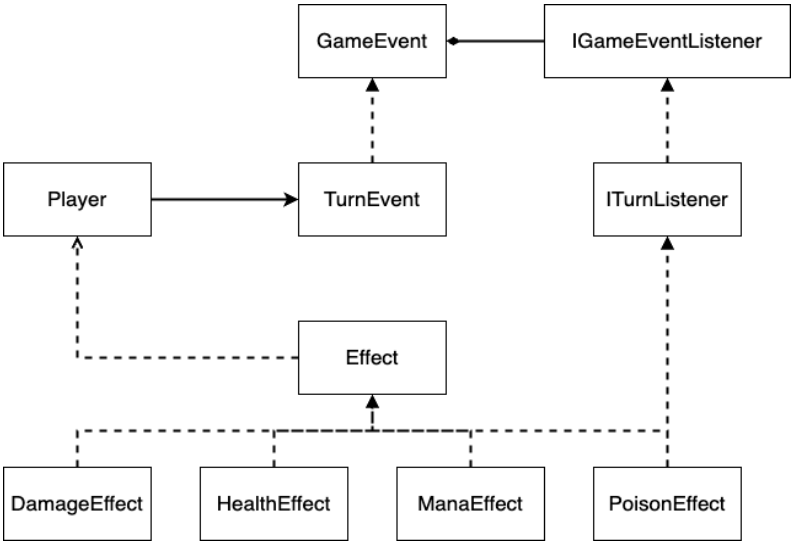

Figure 5 - Publish-Subscribe Class Diagram

Additionally, the materialization of the Strategy pattern is also shown: by the power of polymorphism, the different subclasses of Effect will act as a strategy with their apply method on a Player object, because depending on what effects a played card has, different apply implementations will be executed. All of these will allow us to make effects completely modifiable, because what an effect does to a player can be customized in its apply method, and with different GameEvents, they could also be triggered at various points during the game.

7.1.4 Services Class Diagram

The application will depend on many services, from authentication through lobby creation to game communication. We're going to be using Firebase for these services, but in order to make the application independent of the service provider, we utilize the Facade pattern to separate the interface from the implementation. As an advantage, if the provider had to be replaced at some point, it could be done just by switching out the Firebase implementation to a different one, without making any changes to the application.

The planned services, their methods and responses can be found on the communication diagram in Section 7.2.3, which shows the service interfaces. The utilization of Facade with

the Firebase implementation of services is presented by the example of GameService in Figure 6. The GameService interface defines the necessary method for game creation and communication during the game, while the FirebaseGameService will be an implementation of it.

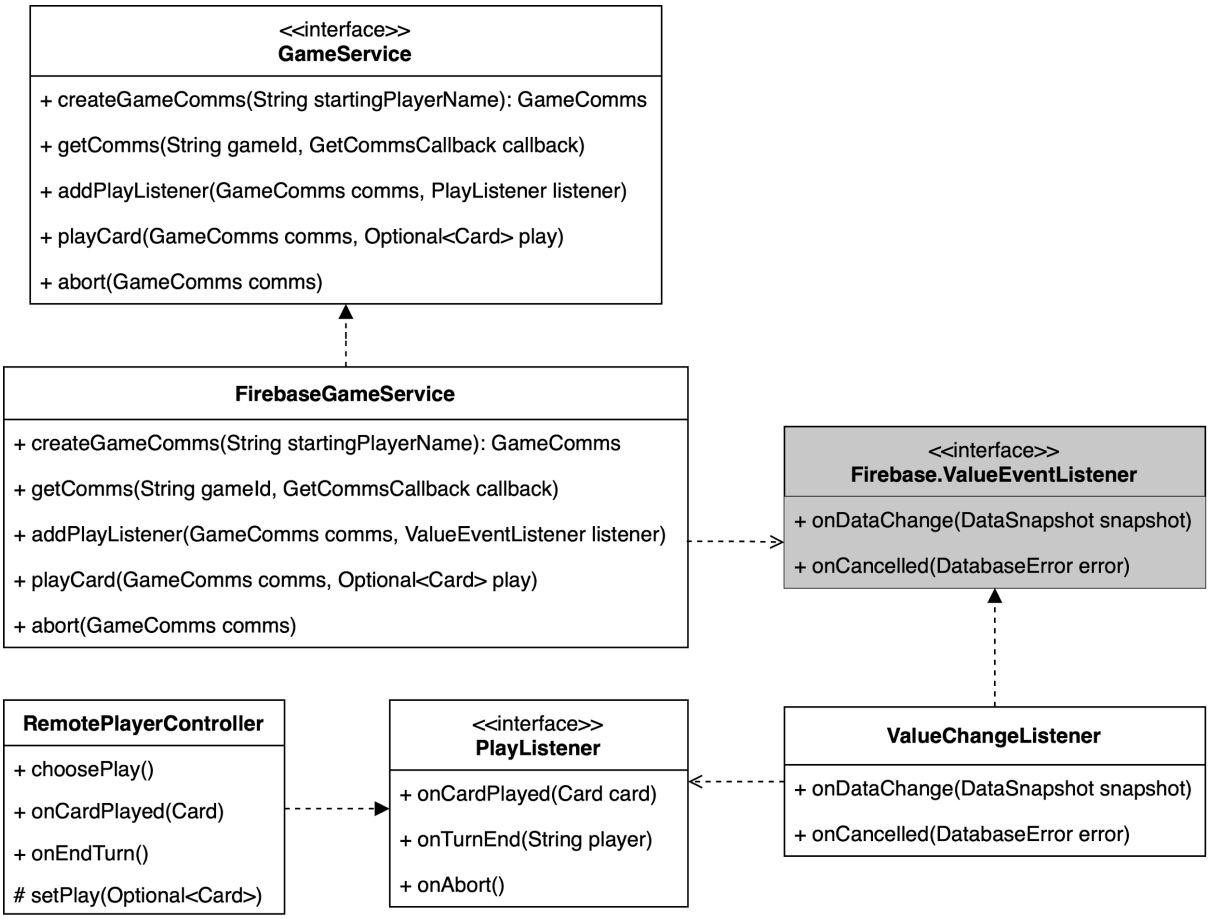

Figure 6 - Services Class Diagram

Additionally, our planned use of Adapters can also be seen in Figure 6. Firebase has specific interfaces (e.g. ValueEventListener) that can be used to register listeners to updates coming from the database. We will have to utilize a class like ValueChangeListener to adapt the update coming from Firebase with raw data to plays that can be passed onto a RemotePlayerController through its PlayListener interface.

7.2 Process View

7.2.1 Sequence Diagram

The sequence diagram in Figure 7 depicts the main game flow. The GameController will orchestrate the loop by first asking the upcoming player's PlayerController for a play, and then applying the provided play to the game state. A play can either be a card or the player ending their turn. The loop will continue until the game is over, i.e., when at least one of the players dies. Since Figure 7 focuses on the game loop, the user and interaction and communication are represented on separate sequence diagrams, namely Figures 8 and 9. These sequence diagrams show how the PlayerControllers get the selected play, as well as how they communicate with the server. The full sequence is broken up into two parts here for legibility: one for getting the play from the user and pushing it to the server, and one for receiving a play from the server. The sequence as a whole with 2 clients can be seen in the Appendix in Figure 14.

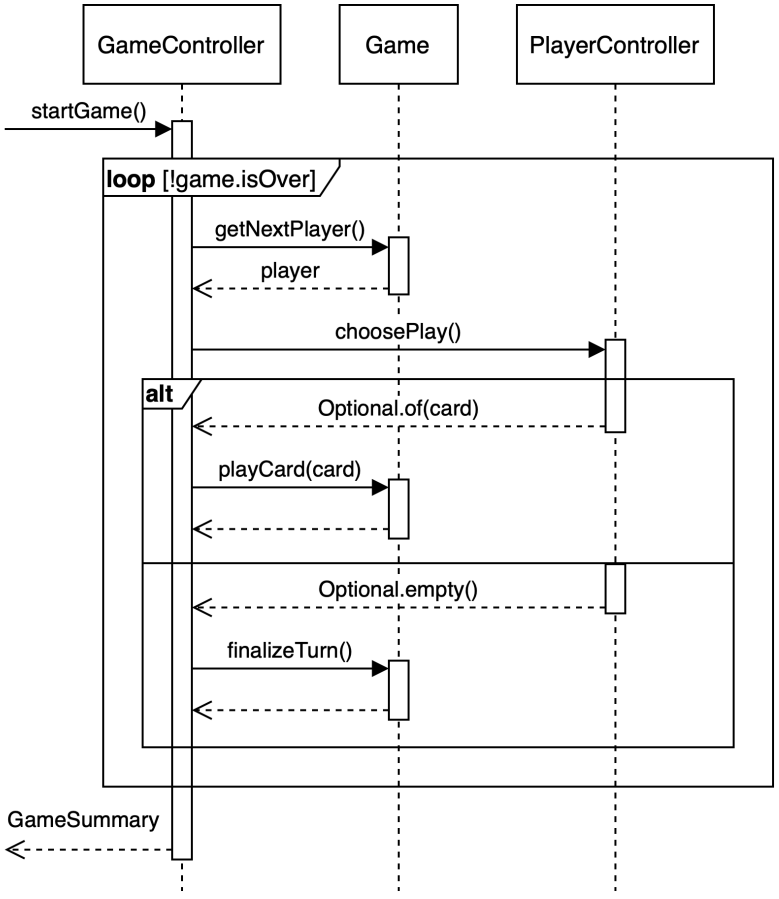

Figure 7 - Game Loop Sequence Diagram

In Figure 8, the planned flow of information from the user touching a certain play to the server being updated is presented. The sequence expands upon Figure 7's third message, where the PlayerController is asked for a play. HostPlayerController will be responsible for getting a play from the user through the graphical interface and pushing the selected play to the server. After being asked for a play, it will wait for a user input by being registered as a ClickListener of the CardViews and the button for ending the player's turn. Once it receives the play from a click on the corresponding graphic, it will push the play to the server using GameService, which in our case will be implemented as communication to Firebase.

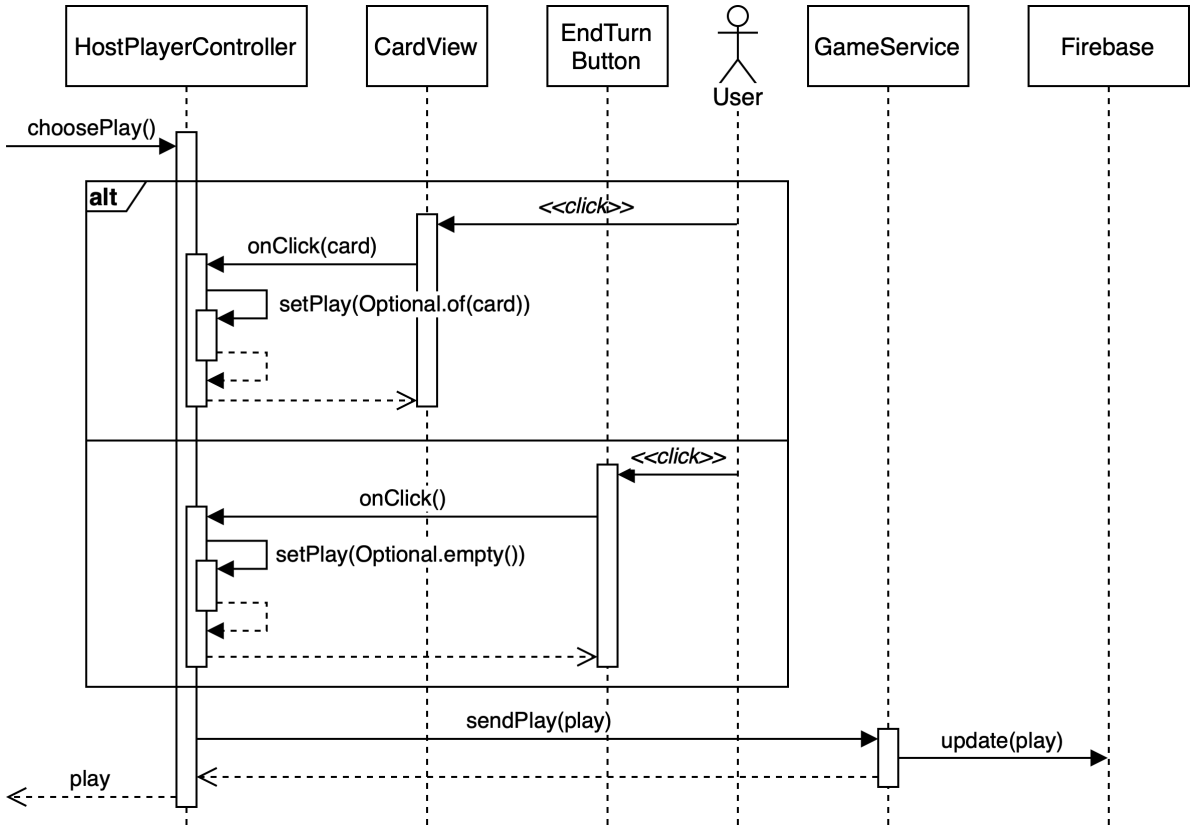

Figure 8 - Host Plays Sequence Diagram

Figure 9 continues the sequence from the other client's standpoint, it shows how the updated play in the server will get through to the other client's PlayerController. This sequence also expands upon Figure 7's third message, where the PlayerController is asked for a play. RemotePlayerController will be responsible for listening to the server's updates and converting them to a play of the opponent. After being asked for a play, it will wait for a server update by being registered as a listener to the server's value change events. In our case, this update will come from Firebase. Once it receives the play from the server's update, it will be converted to either a played card or the end of the player's turn.

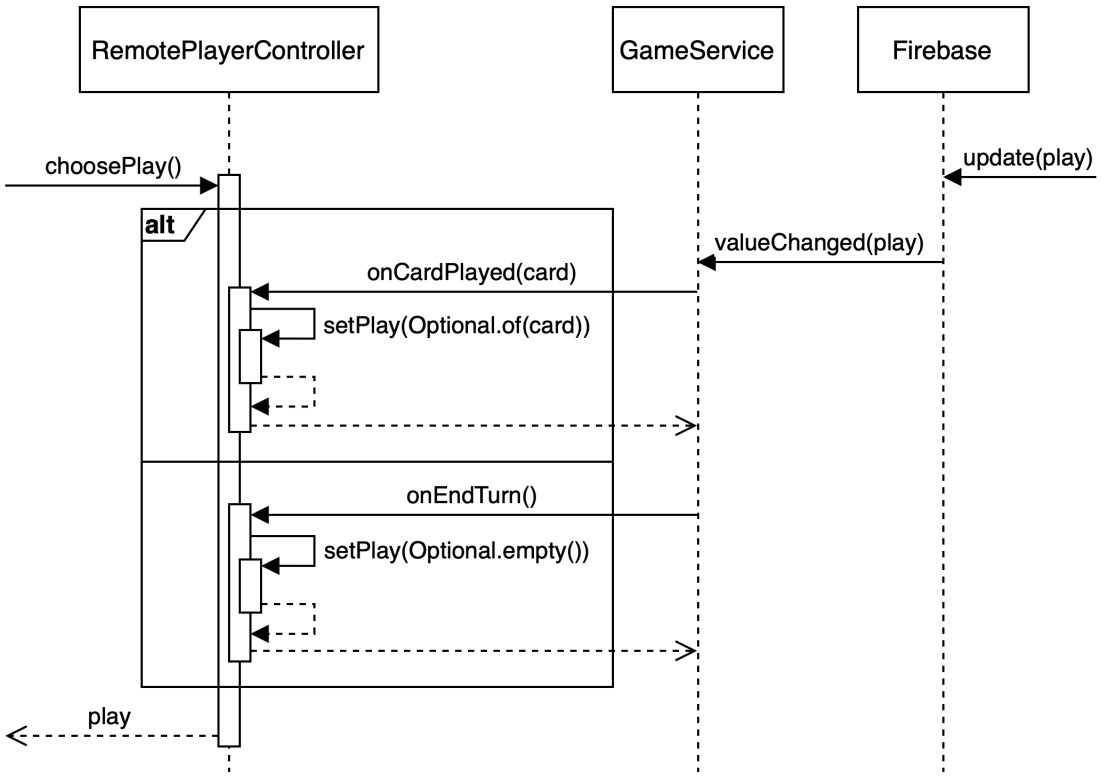

Figure 9 - Remote Opponent Plays Sequence Diagram

7.2.2 Activity Diagram

The activity diagram in Figure 10 describes the runtime control flow of the application. More specifically, it displays how to traverse between different states/screens. Upon launch, the user is prompted to log in using their credentials. If they are valid, the user is transferred to the Main Menu where they may perform a variety of actions - if not, they can choose to Sign Up. The user can enter Class Selection, initiate the Tutorial, Log Out, quit the game or view the available Game Lobbies - where all of these actions lead back to the Main Menu should the user choose to. Once the user is in a Lobbies Screen, they can pick an existing Lobby and Join Lobby, Create Lobby and create their own or view their Game History. If a lobby contains two users, the Game may be started by the host. Then both users enter the Game where they can interact with their opponent in a turn-based environment. After the game has run its course or either player forfeited, it will conclude and the user is immediately transferred to the Game Result screen where a summary of the game is displayed, and the user is prompted to return to the Main Menu.

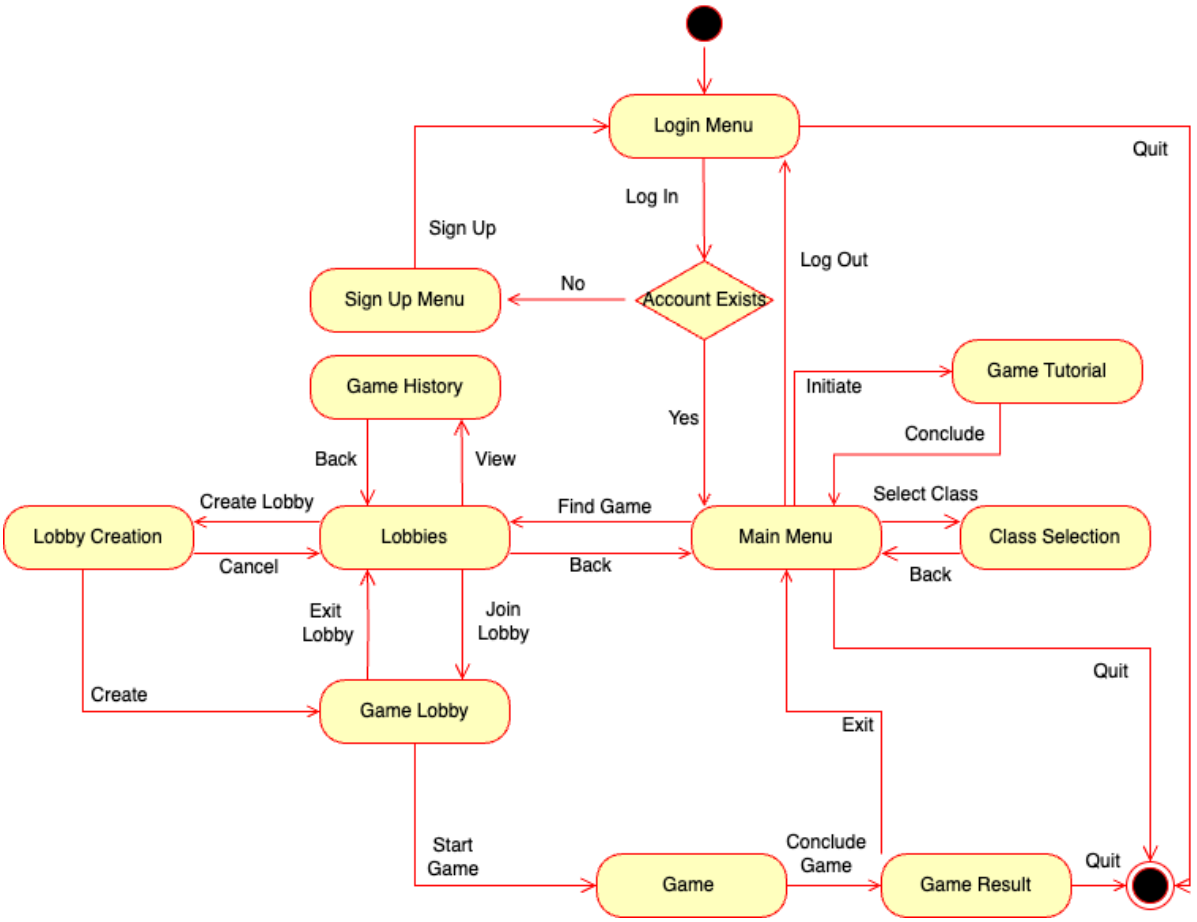

Figure 10 - Process View Activity Diagram

7.2.3 Communication Diagram

The communication diagram in Figure 11 illustrates the interactions between the client and all the different services in the system. For this diagram, we have chosen to showcase only successful callbacks from the services, under the assumption that any unsuccessful callbacks are appropriately handled.

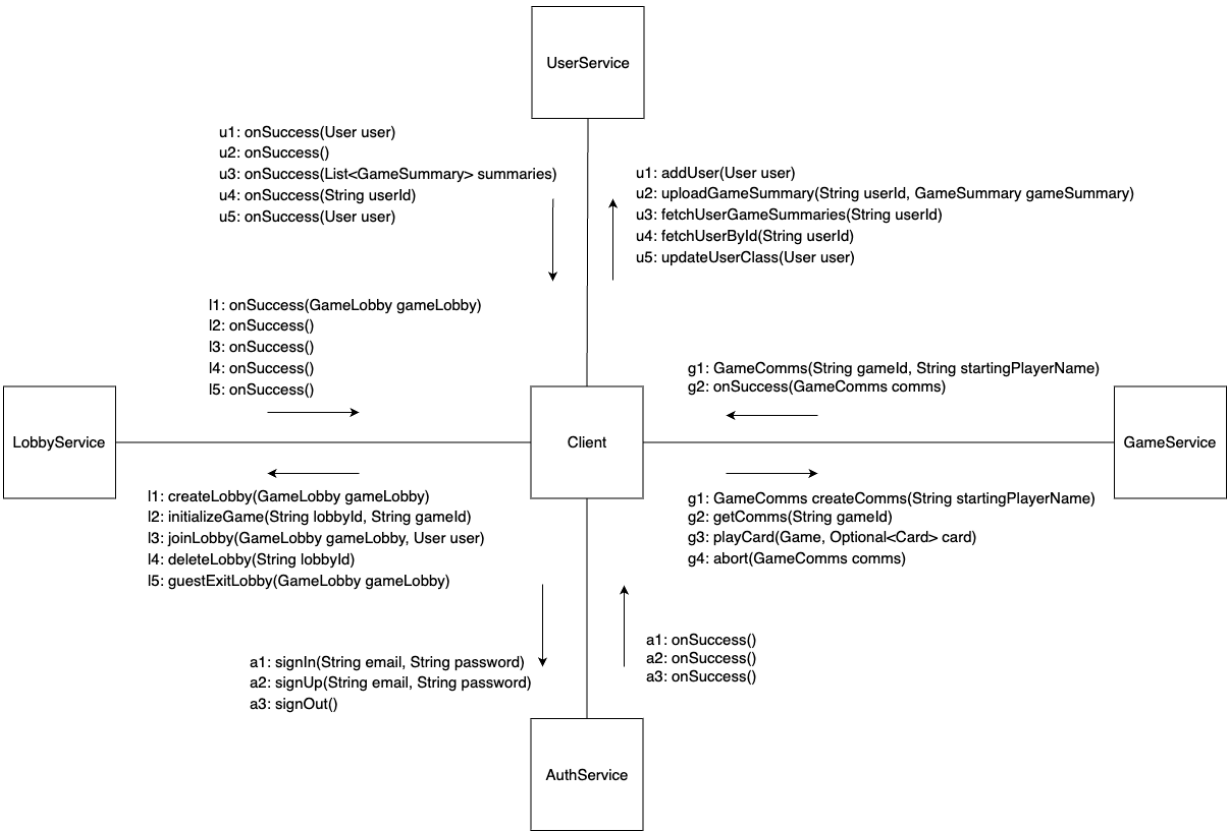

Figure 11 - Communication Diagram of the client and services

7.3 Development View

The development view is concerned with the development process and tools used to build and maintain the system. It shows the system from the programmer's perspective, as seen in Figure 12.

7.3.1 Package Diagram

The package diagram below gives an overview of the structure and dependencies between the main modules of the model-view-controller architecture. The arrows all represent dependencies between packages. Bidirectional dependencies are represented by blue arrows, while unidirectional dependencies are represented using red arrows.

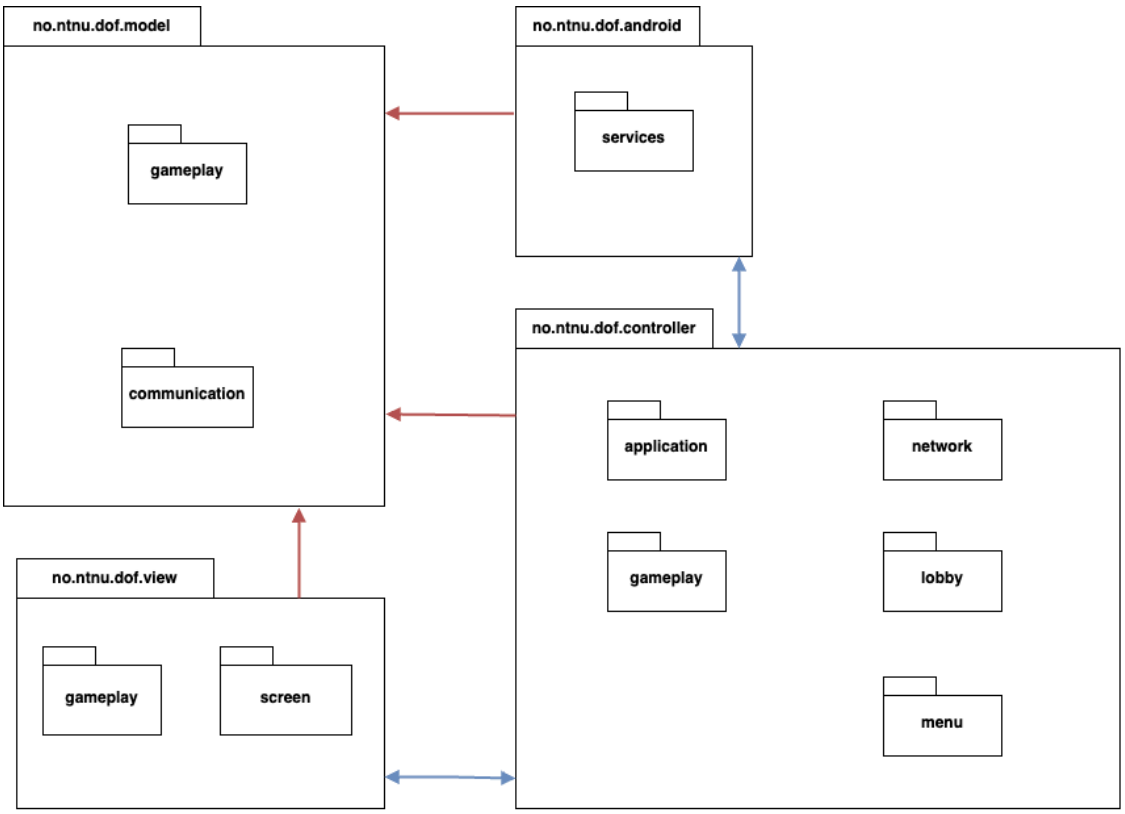

Figure 12 - Development View Package Diagram

7.4 Physical View

7.4.1 Deployment Diagram

The deployment diagram in Figure 13 exhibits how the Android application will be deployed to compatible mobile devices as well as how these devices will make use of Google Firebase for communication and authentication, accessed by a web API.

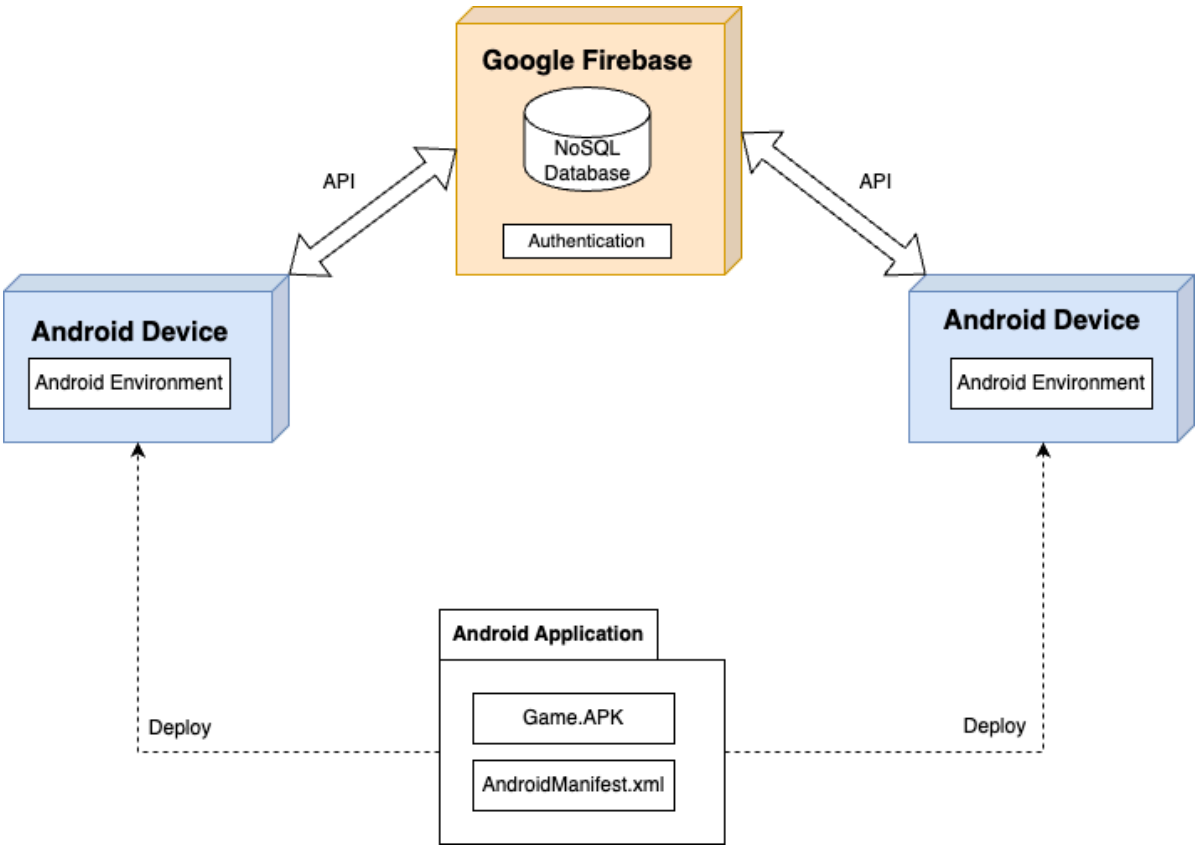

Figure 13 - Physical View Deployment Diagram

7.5 Inconsistencies Among Views

Each view was created with a different purpose in mind, therefore there are no major inconsistencies among the majority of the views. However, due to their abstract nature, some views exhibit more components than others. One example is that in View (7.1.1), where we highlight the implementation of MVC, not all components of the main entities are displayed as they are in View (7.1.2), such as PlayerClass, Health, Mana, Armor and so on. Additionally, View (7.1.1) contains a significant subset of what a full expansion of the Package Diagram View (7.3.1) would potentially show, however doing that would only make the point of (7.1.1) harder to exhibit.

In the sequence diagrams (7.2.1), the adapting details mentioned in the Class Diagram for Services (7.1.4) are abstracted away by a Firebase object and an update method. This is because the focus of those diagrams is to present the idea of how information flows from one client through the server to the other client, rather than the technical details of parsing and adaptation.

8. Architectural Rationale

The architecture for the project was selected based on the quality attributes of modifiability and usability, alongside the requirements proposed by the 1-versus-1 duel card game inspired by "Slay the Spire". Among the Architectural Significant Requirements (ASRs) are also turn-based multiplayer gameplay and the implementation of numerous in-game cards.

The Architectural pattern of Service-Oriented Architecture (SOA) in combination with Shared-Data was chosen over a peer-to-peer solution, mainly due to the simplicity of testing and integration with the LibGDX framework. By utilizing Firebase's Realtime Database, we can easily establish an online connection between players and host the game state in an efficient and orderly manner. Furthermore, by utilizing Firebase as a BaaS, we are also easily positioned to leverage other services provided by Firebase such as authentication and storage. Ultimately, utilizing Firebase increases the modularity and modifiability of our application by centralizing the game state (Shared-Data) in the Realtime Database and through the easy integration of their plug-and-play services. We can leverage this to have users subscribe to new cards played in the game they are active in, in a Publish-Subscribe fashion, so that everything is kept consistent between both users as well as their views.

Supporting the same primary quality attribute of modifiability, the Model-View-Controller (MVC) pattern segregates business logic from the user interface. Entity Component System (ECS) further decouples game logic from data, allowing for dynamic composition and interaction of game entities, like players and cards. These patterns also contribute to the usability quality attribute by making it easier to implement tutorial-related and UI-enhancing elements through the decoupling of code logic. The architecture also incorporates a set of creational, structural, and behavioral design patterns.

Singleton, Builder, and Facade will be used to enhance code modifiability and usability, especially as there will be a creation of a plethora of game objects with different attributes and functionalities. Strategy and Observer will play a major role in the versatility of Effects, creating opportunities for multi-turn and sophisticated applications of them. Together with the architectural patterns, these choices support the requirements of turn-based multiplayer gaming and a dynamic incorporation of new cards. On the graphical side, in order to avoid repetition and allow for shared behavior between screens, the usage of Template Methods will be key. Also, our main graphical COTS - LibGDX - makes significant use of Composite structures, and due to the complexity of our application, we plan to override them so as to

implement our own functionality with them. Finally, Adapters are necessary so as to ensure compatibility between different interfaces. Our architectural and design choices facilitate a smooth development process both for the initial development as well as future iterations and updates to the application.

9. Issues

9.1 Phase 1

9.1.1 Architectural Views

In the first version of this document, a variety of architectural views were used to describe our architecture. However, due to this process being in a primitive stage, they were either skeletal or subject to considerable change

9.2 Phase 2

There were no particular architectural issues in the revisions of this document.

10. Changes

Date	**Change**	**Comment**
March 3, 2024	Initial document	N/A
March 22, 2024	Modify (7.3) Deployment View and (7.4) Physical View	Adjust based on examiner feedback, include Firebase Authentication
March 24, 2024	Modify (7.1) Deployment Diagram and (7.2) Process View	Adjust based on examiner feedback and re-evaluation of screen flow, compromises
March 26, 2024	Modify (5.1) Modifiability Tactics and (5.2) Usability Tactics	Adjust based on examiner feedback, narrow down to specific

		tactics rather than discussing generally
March 29, 2024	Modify (6.1) Architectural Patterns, (6.2) Design Patterns and (8) Architectural Rationale	Adjust based on revised architecture after reiterating
April 16, 2024	Modify (1.2) Game Concept General Modifications, (11) Individual Contributions	Adjust based on the final result. Minor adjustments and corrections in phrasing and grammar

11. Individual Contributions

Member	Part of the document	Approximate Hours
Adrien Rosset	Architectural Tactics, Modifiability	3h
William Marleau	Architectural & Design Patterns, Architectural Rationale, Review	7h
Ken Nguyen	Architectural Drivers, ASRs, Architectural Tactics: Usability	6h
William Hassel	Architectural Views	5h
Dionysios Rigatos	Introduction, Stakeholders and Concerns, Architectural Viewpoints, Architectural & Design Patterns, Architectural Tactics, Architectural Views, Architectural Rationale	11h
Márk Somorjai	Modifications on Architectural and Process VIews	11h

12. References

[1] MegaCrit. Slay the Spire. Steam. Available at: https://store.steampowered.com/app/646570/Slay_the_Spire/

[2] Bass, L., Clements, P., & Kazman, R. (2021). Software Architecture in Practice (4th ed.). Addison Wesley.

[3] Wang, A.I., & Montecchi, L. Chapter 8: Modifiability. *TDT4240 - Software Architecture Lecture Slides*. Norwegian University of Science and Technology.

[4] LaRoche, C. (2020). Usability Testing: An Introductory Workshop. Available at: https://studentlife.mit.edu/sites/default/files/Documents/Usability_Testing_Workshop_March2020.pdf

[5] Wang, A.I., & Montecchi, L. Architectural Patterns from “Software Architecture in Practice 3rd edition”. *TDT4240 - Software Architecture Lecture Slides*. Norwegian University of Science and Technology.

[6] LibGDX. Scene2d. Available at: https://libgdx.com/wiki/graphics/2d/scene2d/scene2d

Appendix

1. Host-Remote Communication

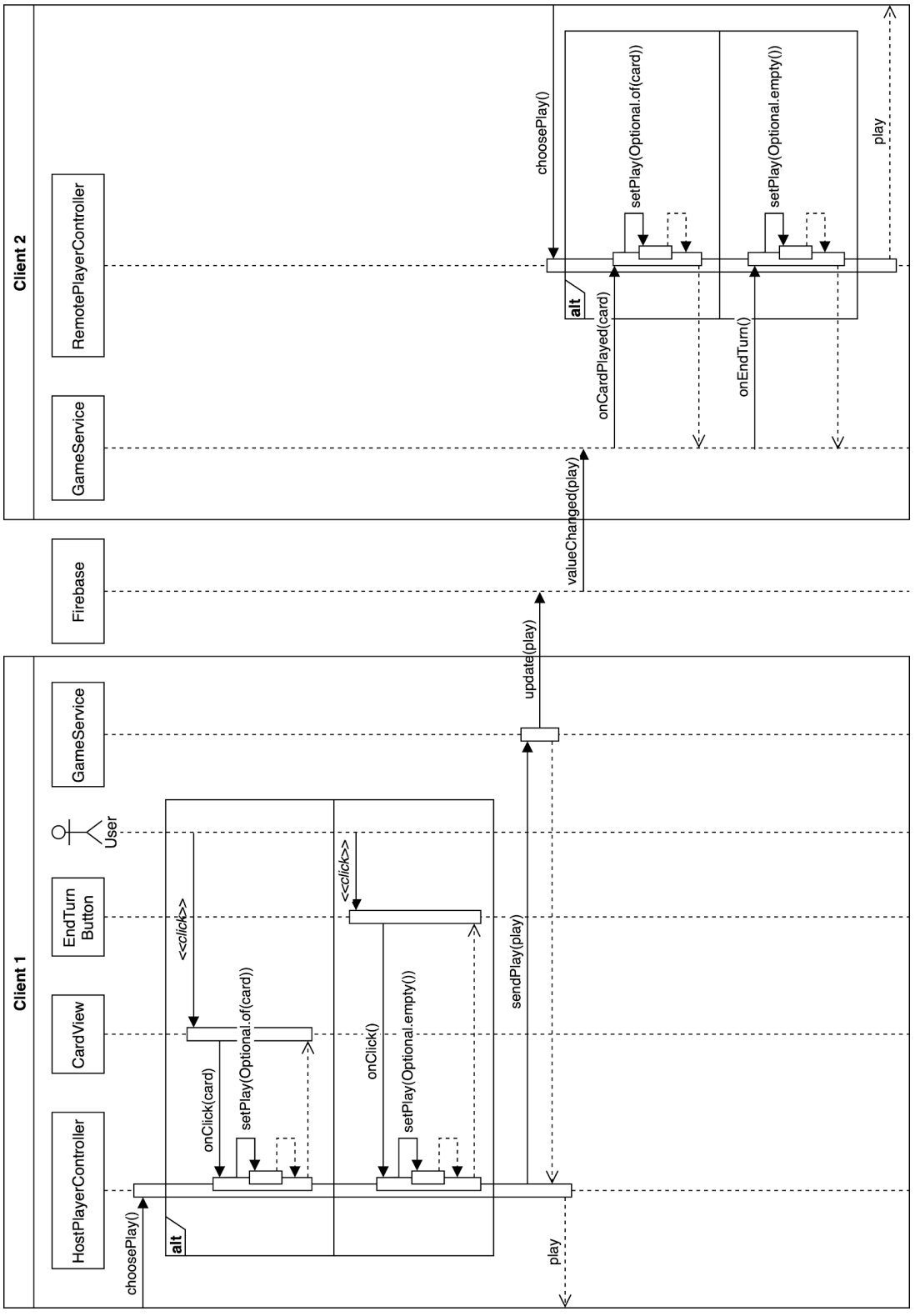

Figure 14 - Full Host-Remote Communication Sequence Diagram