NTNU
Kunnskap for en bedre verden

TDT4240 - Software Architecture

"The Nearly Impossible Game" Architectural Description

Group 12:
Jakob E. Eikeland,
Teo Chen Ning,
Tan Chuan Xin,
Gaute Brandser,
Ruben Rokkones,
Sigve Sjøvold

COTS: Android, LibGDX, Firebase

Primary QA: Modifiability
Secondary QA: Usability Secondary QA: Availability
April, 2022

Contents

1 Introduction

The goal of this project is to create a functioning multiplayer game or app for mobile devices. This document provides a detailed description of the project architecture.

1.1 Description of the game concept

Our group had several ideas about what type of game we wanted to create. after some discussion we decided to try to create a version of "The Impossible Game", a mobile game some of us had played when we were younger. It is a game where you play as a square going through a course made up of various obstacles appearing in its path with increased frequency and difficulty. The goal of the game is simply to reach the end of the level without dying. The basic flow of the game is that the game starts, the square (the player) starts gliding forward on the ground, and spikes start appearing. If hit from any direction by the spike, you will die, and restart the game. The player controls the square in each direction, and can jump.

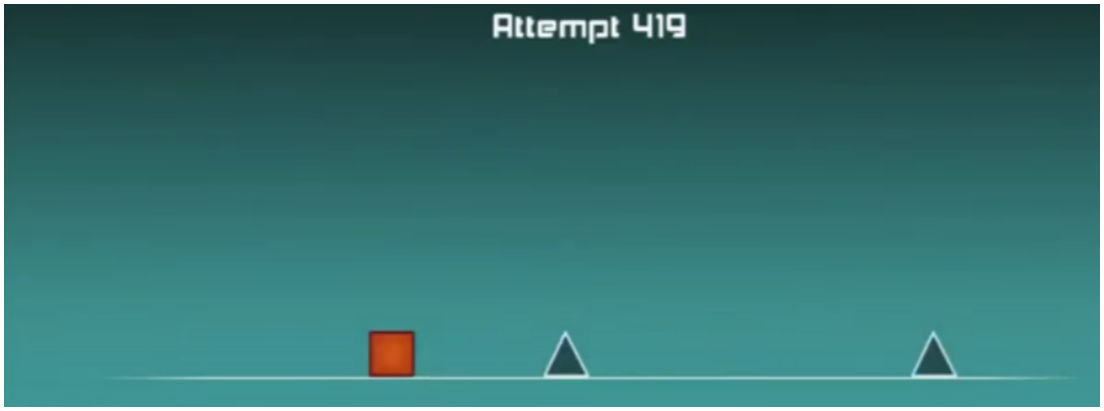

Figure 1: Original game

Our version of this game will have both single player and multiplayer modes. In single player mode, it is similar to the original Impossible Game. In multiplayer mode, two players will alternate in turns playing the game. We will also implement an online high score list. We considered various different ways of having multiplayer functionality, such as two mirrored courses on each side of the screen, or maybe two players next to each other on the same course, but decided to have a two players taking turns playing the same level. When the first player dies or wins, the game switches to the other player, who tries

playing until he dies or wins. When both players have beaten the level, or died, they can type in their names and submit their scores to the database. They also have the option to skip this step, if they feel their scores are too low.
This version of this game ends up looking very different. We wanted to make our own game, and not copy the amazing original game. However we kept the core components of blocks and spikes. In our game you play as a jelly, and your goal is to get to the end of the map.

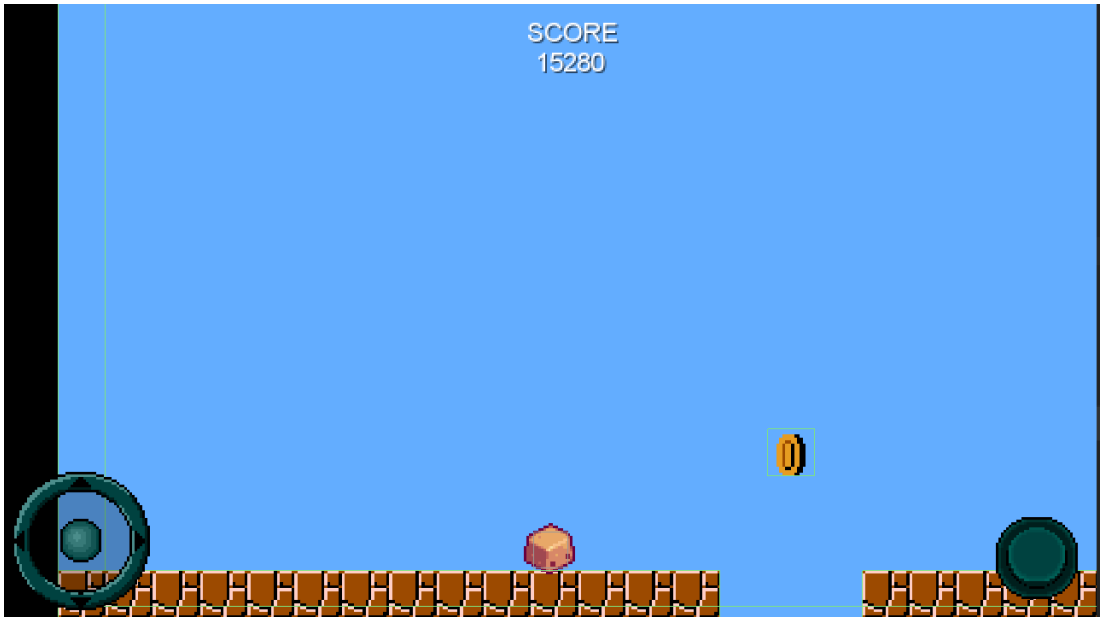

Figure 2: Beginning of game

The player has a jump button, and a joystick that controls lateral movement. On the way to the goal, the objective is to get the highest score as possible. This is achieved by picking up as many coins as you can, and going as fast as you can. The player only has one life, because this is "The Nearly Impossible Game". Another element we added to the game is fireballs. When you're gliding through the map with ease, a fireball can suddenly appear and mess up your game.

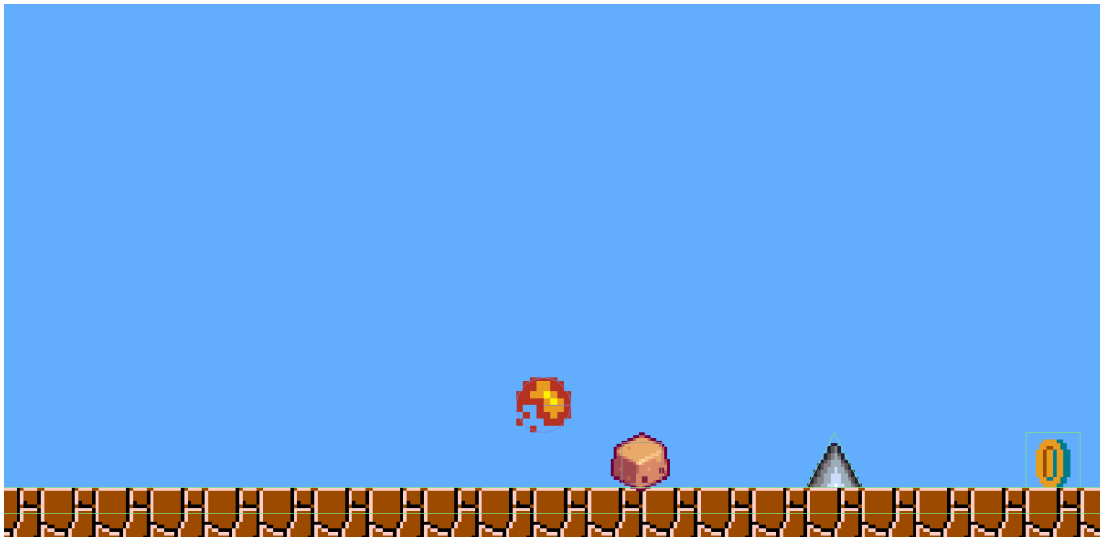

Figure 3: Fireballs and coins

Another way of messing the player's game up comes from the other player in when playing multiplayer. The player not currently playing can press a button in the top right hand corner, shoving the active player in a random direction in order to try to stop him from getting a good score.

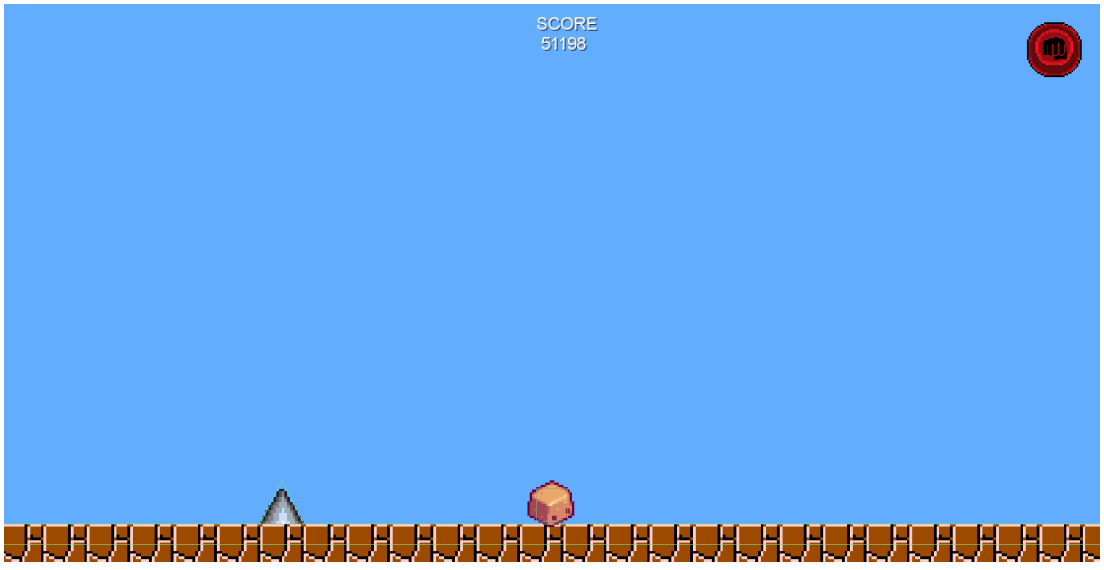

Figure 4: Button to mess with opponent

1.2 Structure of the document

This document gives an overview of the architecture we have outlined for our game. Section 2 discusses architectural drivers for the project, while section 3 is about stakeholders and concerns. Section 4 details our choice of architectural views, and 5 is about architectural tactics. Section 6 is about patterns relevant to the project, 7 shows the actual architectural design for the project, with illustrations. Section 8 is about the consistency of these views, and 9 discusses architectural rationale. Finally, we show the individual contribution of each member, changes in the project from beginning to end, and references.

2 Architectural Drivers

In this section, we will explore the various architectural drivers considered by the team. These led us to choose certain design tactics and architectural patterns to fulfil the functional requirements set out by the team while achieving our chosen quality attributes.

2.1 Technical constraints

The technical constraints imposed upon our project will have significant impact on our chosen design and architecture. The inherent requirements are as follows:

1. Must be built for Android
2. Must be multiplayer in some way
3. Must have an online component

2.1.1 Android client

In translation, this means that we have to support an efficient implementation as it has to run on mobile devices. To that end, we are choosing LibGDX as a game development framework as it will help us optimize our application, allowing us to focus on the game itself.

2.1.2 Multiplayer capabilities

Due to time constraints and complexity, we opt to forgo real time multiplayer and instead implement turn based multiplayer (where only one player plays at a time, and they compare scores after both players have played). Therefore, it is important that the game state is persisted throughout both turns.

2.1.3 An online component

In consideration of creating an online component, we opt to build a high score list to encourage competitiveness and bring a greater incentive to players to perform well in the game (other than beating their friends). Syncing high scores across multiple clients (especially globally) is a challenging task as the data has to be up to date and handle fast updates and potentially many queries.

Rather than implement our own database which necessitates us managing a server and backend to handle reading and writing from the database, it would be prudent to dedicate the infrastructural responsibilities to a managed service

like Firebase, where highscores can simply be stored in their NoSQL database and queried easily through their API. This would save a lot of developmental and maintenance efforts down the line, allowing us to focus on the core aspects of the game.

2.2 Business constraints

2.2.1 Timing and budget

The project has to be implemented within 2-3 months, including documentation. Similarly, the project team consists of 6 students without much resources. In consideration of these factors, certain trade-offs have to be made architecturally.

As mentioned, real time multiplayer will not be considered since the complexity is high. One will have to deal with syncing of states, movement, latency, disconnects and more. By choosing to implement a simpler game with a turn based multiplayer system, we localize all code to a single client. This allows us to create a better product with our limited time instead of under-delivering with a complicated product.

Similarly, an online component is implemented as a high scores table rather than a full fledged server-client pattern where game states are synced between two separate clients.

We chose a high score list as the online component for several reasons. Firstly, since our game is not planned to be a real time multiplayer and does not require two separate clients (the game can be played on the same phone), we do not require a complex setup of a server-client architecture to manage and sync game state.

Secondly, as the team is small and has little resources, it is infeasible for us to host a server and database to implement many online tasks such as state syncing and server-client communication. This would involve many other connected tasks such as encryption and certificate management, availability, load balancing, distributed servers and the like.

It would be more feasible to start small and use a managed service like Firebase to contain database state and high score tables, and expand our infrastructure from there if necessary.

2.3 Functional requirements

2.3.1 High score table

We require a high score table to be readily available in the application. When a user finishes a game, scores are automatically compared to the high score list and added to the list in a sorted order. The table should be synced in real time across all clients that play the game. A central server and database should be integral for this requirement so that the state of the high score list remains consistent.

As this is the only online component we require right now, it makes sense to simply outsource it to a managed service like Firebase rather than host a whole server by ourselves just for this. Firebase incidentally also provides user registration and management, allowing us to keep user profiles without having to manage passwords and authentication on our own.

2.3.2 Different views

The view should be easily modifiable as there are different buttons that are tappable when the player runs. For example, when a player presses the jump button, it may be possible for the button to turn to a "stomp" button to cancel the jump halfway. It would be prudent to consider something like a Model-View-Controller design pattern so that views can be dynamically updated based on the data and state of the game, and in case of changing requirements.

2.3.3 Game states

The player should be able to move left and right as well as jump. We must also consider the player direction when rendering our animations. To keep track of the state of the player, we may consider something like a State pattern to modify behaviour and the MVC pattern as well to modify the player animation based on his/her state.

There are various game states such as the menu screen, the high score list screen as well as the game state itself. Allowing for these states to be easily modifiable and separated is key for a maintainable architecture. The State pattern and model view controller (MVC) can be implemented to allow for the change in behaviour of various objects and scenes depending on the state.

2.3.4 Customizability

We also plan to have the player pick different character skins for their runner. This system should also allow for modifiability in that we can add more skins to the game and everything should still work when skins are swapped. A Factory pattern together with an MVC pattern can be considered to create characters with different skins. In this way, characters will all retain the original interface but may have different functionality and designs.

2.4 Quality attributes

2.4.1 Modifiability

We are also concerned with the modifiability of the game. As the game requirements are not set in stone yet, we may choose to change or add several features such as adding new environment obstacles that we think of, or adjusting player behaviour (e.g. jump and slide instead of just jumping, or being able to break some environmental obstacles instead of just dodging them).

To that end, abstractions and behaviour must be loosely coupled, and the application of SOLID principles are heavily advised. A possible design pattern that could help is the Factory pattern, whereby a general Factory interface is created, and the exact implementation(s) are individually implemented by child classes. This allows for extensibility while discouraging modification (which could introduce breaking changes).[1]

2.4.2 Usability

A key quality attribute for us to achieve is usability. For a game, this is paramount as a user can easily uninstall the game from their phone within the first few minutes if they don't understand how to play it or don't find it engaging and fun.

We are trying to make sure the user is able to learn the basic controls of the game as soon as possible. To that end, we must design the user interface to be intuitive and familiar, such as using common concepts like a menu screen and tapping left and right buttons to move left and right.

The beginning of the game must also be designed to start slow to allow a new user to familiarize themselves with the game and its interface. The running user will quickly learn how to jump as there is only one button for him/her to press. Levels have to be designed to progressively introduce players to new mechanics, with little prompt necessary. This emphasises learnability, which enhances usability.

The high score list is integral to keeping users entertained and competitive, inducing replayability. We aim for the high score list to load quickly within one to two (1-2) seconds and be synced in real time. To that end, it is recommended that we consider Firebase as a managed solution as it provides all of those functionalities out of the box and is free to start with. This provides satisfaction for players, when they see their names and scores on the leaderboard.

2.4.3 Availability

Availability is the final quality attribute that we are considering. A major source of frustration when using any piece of software is for the software to fail unexpectedly. This is very detrimental towards the users' experience, and such software failures must be avoided. High availability is about ensuring services do not fail when they are needed, and even when they fail that the failure be handled gracefully.

For our game, we implemented extensive error checking and handling in order to ensure that if an unexpected event occurs, the user will be smoothly guided towards the main menu, and can restart their journey from there. Commercial Off-the-Shelves (COTS) software systems that are chosen to be integrated into our game should also be reputable, and show a proven track record of maintaining high service availability.

3 Stakeholders and concerns

The stakeholders of the system and their concerns related to the software architecture can be seen in Table 1.

Stakeholder	Concern
Course-staff	The course-staff are concerned with the learning outcomes of this project, and whether the student has sufficiently demonstrated understanding and application of various architectural and design patterns that are relevant to the project at hand. They are also concerned with the readability and ease of understanding of the code and documentation in order to grade the submission. They may also be looking forward to whether the game will be the next big hit such as Fun Run or Kahoot! All architectural views are important to them since they have to grade the project based on the documents and also use it to understand the project and reasoning behind certain architectural choices better.
End-users	These are the main users of our application and thus most of the design and implementation should be made with their needs in mind. They want a fun and engaging game that is easy to pick up but has high replayability, making it an enjoyable experience for them. The game should function well and be free of errors and lag, which could detract from the user experience. They are also interested in what new game features and updates are coming. They are mainly interested in the logical view as it provides details about the game's functionality and features, allowing them to get a good grasp of what the game aims to be and future improvements.
Project team	The project team is highly concerned with the maintainability and developer experience of the project. By designing a solid and flexible architecture, long term support and extensibility of the code base will be made easier. The team is also invested in creating a highly enjoyable and interesting game that is played by many users. They are also interested in receiving a high evaluation from the course-staff. All architectural views are important to the team since they have to create the views and develop based on them.
Evaluators	Evaluators are other teams who are also building their own game. Their main concern is that the architectural views are explained and displayed in a clear and concise manner as they have to understand and evaluate it. All architectural views are important to the evaluator as they have to evaluate them as a whole.

Table 1: The stakeholders and their concerns

4 Selection of Architectural Views

This section describe the architectural views that are used to visualize and explain the system architecture. The different views can be seen in Table 2, along with their purpose, notation, and the stakeholder that the views are of importance to.

Architectural view	**Purpose**	**Stakeholder**	**Notation**
Logical View	Describes the features the system provides to the user	end-user, developers, course-staff	Class diagram
Development View	Describes the major components of the system	Developers, course-staff	UML Component Diagram
Process View	Maps software to hardware	Developers, course-staff, end-user	UML Navigation Diagram
Physical View	Visualizes how classes and components interact to provide the required functionality	Developers, course-staff	UML Deployment View

Table 2: Selection of architectural views

5 Architectural Tactics

In this section we will describe the main architectural tactics we will use in our project, to ensure that the functional and quality requirements, described in the requirements paper, are fulfilled. Sources are chapter 4, 7 and 8 in the syllabus book [2].

5.1 Modifiability tactics

The goal of modifiability tactics is to control the complexity of making changes, thus saving cost. Fundamental complexity measures of software design – coupling and cohesion. "Generally, a change that affects one module is easier and less expensive than a change that affects more than one module.".
Below, we go more into depth on coupling and cohesion, and how we can use it.

5.1.1 Loose coupling

Coupling is the relationship describing the probability that a modification to one module will propagate another overlapping module. A low coupling is therefore of high importance to enhance modifiability. To reduce coupling, one can use the tactics listed below.

- **Intermediaries:** Intermediaries acts as links between components that don't need, or should not have a direct access to each other. This will break dependencies between components, so one can work independently from the other. To use this tactic we plan on implementing the broker pattern and the observer pattern (see section 6.1.2 and 6.2.5).
- **Encapsulation:** Introduces the ability for a class to present an interface that other components can interact with, while internal implementations of the class are hidden. Through this, you can minimize the probability that the change of an element propagates to other elements. This will reduce the number of dependencies, and thereby enhance modifiability.
- **Event driven architecture:** Communication between decoupled services can be mediated through an event driven architecture. Specifically for games, a change in an objects' state could be brought about by any number of interactions in the game world. By pushing events and subscribing to events, we can achieve extremely loose coupling as event consumers do not need to know anything about event publishers.

5.1.2 Strong cohesion

Cohesion means to keep elements that are related to each other tight, and is also a measure of the probability that a change that affects a responsibility, will also affect other responsibilities. "The higher the cohesion, the lower the probability that a given change will affect multiple modules.". Coupling and cohesion is closely related, and one can reduce the need for the other. The main focus to gain a strong cohesion should be to create modules that have a single and well defined task, and rather create more modules instead of have multiple tasks in one module. Tactics to raise cohesion are:

- **Split module:** By splitting modules into smaller cohesive sub-modules in a sensible manner, will decrease the cost of modification.
- **Redistribute responsibilities:** Gather similar responsibilities together. As the project goes on, we may end up having similar responsibilities spread out in several modules. These could be redistributed into a new module together, or moved to existing modules.

5.2 Usability tactics

We want to ensure that the user can easily navigate the application, and that the applications controls and options are intuitive enough. The goal with this is that the user should correctly expect the outcomes of different actions within a minimum time frame. Tactics to ensure good usability are:

- **Keeping the controls simple and intuitive:** We want the application to use controls that users can recognise the functionality of, based on the design. E.g controls should follow design conventions. Our platformer game should follow industry conventions as much as possible.
- **Respond appropriately and provide instructions:** The user should be given appropriate feedback and assistance. e.g if the user chooses multiplayer, the application should clearly indicate that it enters multiplayer, and, at least the first time, explain how the multiplayer works. Audio cues triggered by in-game actions can further elevate user feedback.
- **Loading screen:** The game has a loading screen when initializing the startup of the game. This shows clearly to the user what is going on, contrary to if the game had just had a black screen without any indicators.
- **Pause/resume:** The user can pause and resume the game whenever while playing a level.

5.3 Availability tactics

The game has an online component, namely the high scores, which need a database for storing data. For this we chose to use Firebase, and by this we save ourselves a lot of work compared to setting up our own server with a database. Additionally, Firebase handles the server-sides recovery and synchronization by itself. What is then left is to ensure a certain degree of availability on the client-side. The tactics described below are implemented in the code for this purpose.

- **Exception detection:** For detecting exception in the running code, Java does the work for us. The purpose of the specific tactics listed under the "Exception detection"-section in the syllabus book are covered by Java (see section 4.2), except from exceptions that occurs aside from the code implementation.
- **Sanity checking:** Additionally, we will also use the sanity checking tactic to detect fault and errors which could trigger an exception. To do this we will use conditions to verify correct behaviour of the code.
- **Exception handling:** To prevent system crashes when an exception is detected we will implement code that gives instructions when the exception is detected. This could be a set default return value to mask the fault, and a printed message to console to help the developer find and fix the problem. We won't implement new exception classes as we see this as excessive for the games size.
- **Recovery:** The application should handle disconnection to the database used for the high score table. This error could be displayed on the screen for the user so that they know why they cannot view the high score table.

6 Patterns

Both architectural patterns and design patterns will be discussed in this section.

6.1 Architectural patterns

Architectural patterns (also known as architectural styles) provide high-level definitions for the fundamental structuring of a software system. It lays out a set of conceptual subsystems and their responsibilities and specifies their interactions with other subsystems. Architectural patterns form the underlying backbone and framework of a software system, upon which everything else is built on.

6.1.1 Model-View-Controller (MVC)

The MVC pattern is a widely applicable architectural pattern for software systems, mostly used to achieve a separation between the business logic, system control, and presentation (Section 13.4 in syllabus) [2]. The model will contain the business logic for various game objects, including its data units and how to update these attributes. The views define how the underlying data in the model is displayed to the player and how a player can interact with the game. Multiple views can be defined for the same model; this allows the game to operate on a diverse range of host devices, from phones to tablets to desktop computers and cater to multiple modes of interaction across various devices. The controller processes all requests to and from the model and the view, allowing for communication between these two subsystems through an interface whenever changes are made to either subsystem.

The MVC model greatly enhances the modifiability of the game by providing a clear separation between the data, the view, and the controlling interface. Developers can work on each of these subsystems independently without affecting the other components, thus new views or new business logic can be implemented easily. Given that the presentation of the game is handled by views which can be developed independently, the presentation of the game can be tailored to the device it runs on, thereby increasing usability as well.

6.1.2 Client-Server pattern

The Client-Server pattern helps to share a pool of shared resources with many distributed clients (Section 8.4 in syllabus) [2]. These shared resources are held by the server, which ensures availability, integrity, and quality of data on behalf of the clients. Clients then request for the resource or service, which will be provided by the server on demand. In the context of our game, the online leader board

constitutes the shared resource, and the server must ensure its availability and assure its integrity. Clients check the leader board infrequently, and the leader board includes all clients, hence it is suitably held by a server, and delivered only when requested for.

The Client-Server pattern will also be applicable should the game be extended to a live online multiplayer game. The server will then provide the shared resource of a game state to clients. Clients will then request for the game state from the server and send the server commands to update the game state. This model helps prevents cheating on the clients' end, as the server will be the one making final changes to the game state.

6.2 Design patterns

Design patterns are typically applied within the scope of the subsystems that have been defined by the architectural patterns. The implementation and design of these subsystems often raises recurring and prevalent design problems for developers, to which design patterns can be used to provide a structure by which these problems can be tackled. Design patterns allow for refinement of subsystems within the architectural pattern.

6.2.1 Singleton (Creational pattern)

The singleton pattern is very useful when the game has an object of which there will only be one logical instance of throughout the entire gameplay, and easy access to that object is required. Typically, controllers are created singletons, since there should only be one instance of a class that provides controlling logic over game objects. This ties in well with the MVC architectural pattern in which controllers are a major subsystem of the architecture. In our game, a GameStateController can be suitably implemented as a singleton, and its job is to move the game through the menu state, play state, pause state, and end state. There is no need to create multiple instances of this controller since there is only one game, and the controller must be always available and easily accessible to pause or end the game.

However, singletons are frequently abused to make objects available throughout the entire software system, simply for convenience. Unit testing can also be affected if singletons are stateful, as we cannot isolate the unit from the states that the singleton carries.

6.2.2 Factory Method (Creational Pattern)

Our game will constantly spawn a variety of environmental objects, from blocks that act as steppingstones on the map, typical obstacles on the map, and hazards that are generated on player input. Many objects share common features like size, collision mechanics and interactions, and will benefit from the implementation of a factory. In our game, various obstacles like static spikes and dynamically moving fireballs are created. These objects share very similar properties, with only one or two key differences. Hence, the object creation is deferred to subclasses like Spike and Fireball, which will take responsibility for the actual object creation.

6.2.3 Strategy (Behavioural pattern)

The strategy pattern will be extremely useful when programming for different game modes. Our game has a single player and a multiplayer mode and employing the strategy pattern will allow for the game to switch seamlessly between different modes of gameplay. The strategy pattern creates a family of interchangeable algorithms that are only selected at runtime based on the configuration of the game, or user selection. This allows for us to improve the modifiability of the game, as new game modes like Challenge or Zen can be added on top of Normal mode, and we simply change the algorithm implementation in the new game mode, and the game will continue to work (Section 12.4 in syllabus) [2].

6.2.4 Template (Behavioural pattern)

The template method resolves the problem of writing several pieces of nearly identical code. Game objects tend to exhibit similar behaviour with only slight variations between different object classes (e.g. the Spike class which is an obstacle has the same behaviour as a Block class, except for the collision handling). The super class will be AbstractModel, which will adhere to the Open-Closed principle of being open for extension by other subclasses, while closed to modification in its high-level design. We can then use the AbstractModel class as a blueprint for all models in the game, making modifications through the subclass to achieve different types of obstacles while keeping the same major functionalities.

7 Architectural Views

This section describes the architectural design described through different views which shows the architecture of the system from different angles. The views makes it easier to display the system structure and behavior to the stakeholders, and allow for simultaneous development of multiple parts of the system.

7.1 Development view

Figure 5 shows a simplified overview of the packages and main classes for the system. The system is divided into three main packages: Model, view, and controller. These modules interact with each other through interfaces so developers can work on them separately and in parallel. The ScreenManager class works as a state manager for the screens of the game. Therefore, all screens are decoupled from each other, and in turn makes it easy to assign different developers to work on different screens. Also, we have chosen to encapsulate the physics engine in its own class implementing an the Engine interface to keep it loosely coupled from the rest of the system. The same goes for the networking module. For the time being we plan on using firebase as our network service. The module will be implementing the NetworkService interface to keep it loosely coupled from the other modules, giving us the ability to change the service in the future if needed. For use in development, this view helps in some ways. There is a clear seperation of models, views and controllers, and how they connect, which makes it possible to divide up some tasks that can be developed in parallel.

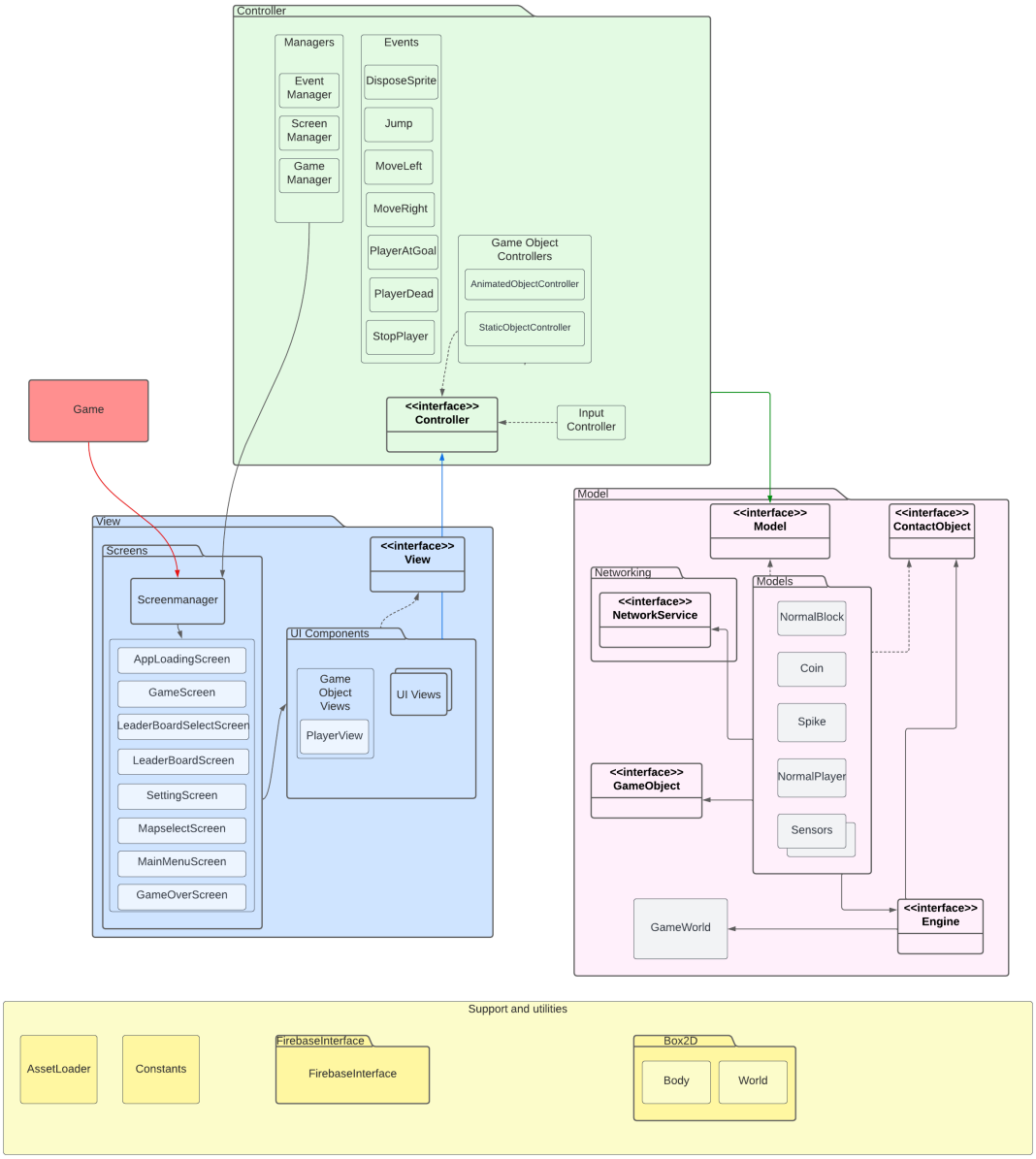

Figure 5: Development view

7.2 Process view

The process view describes the different processes of the system, and how they interact with each other. We have created three different views to show the processes of the game. One for how the different screens change and connect,

one for the process of the basic game loop, and one to show the interaction of between server/client.

7.2.1 Screen overview

Figure 8 shows how the different screens in the system are related to each other and how the screen navigation is implemented.

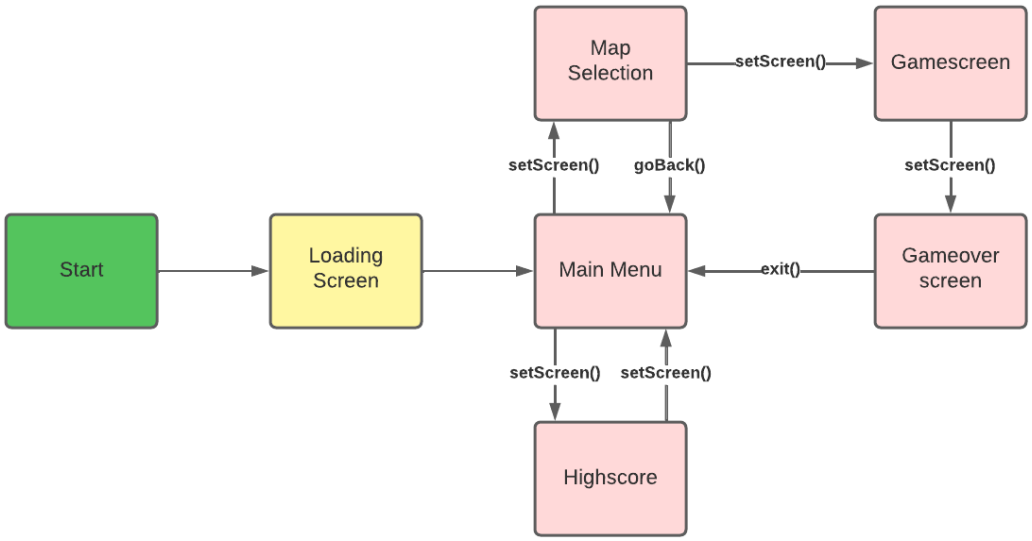

Figure 6: Overview over game screens

7.3 Game loop view

This diagram shows the basic process of a normal game loop.

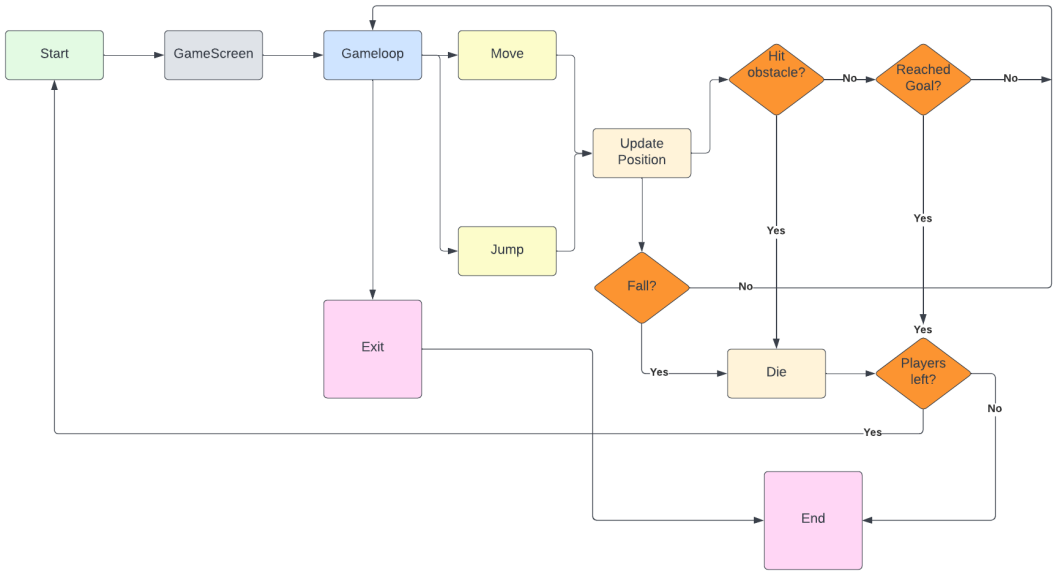

Figure 7: How a gameplay loops

7.3.1 Client-server process view

This view is intended to very simply show the interaction between client and the Firebase server

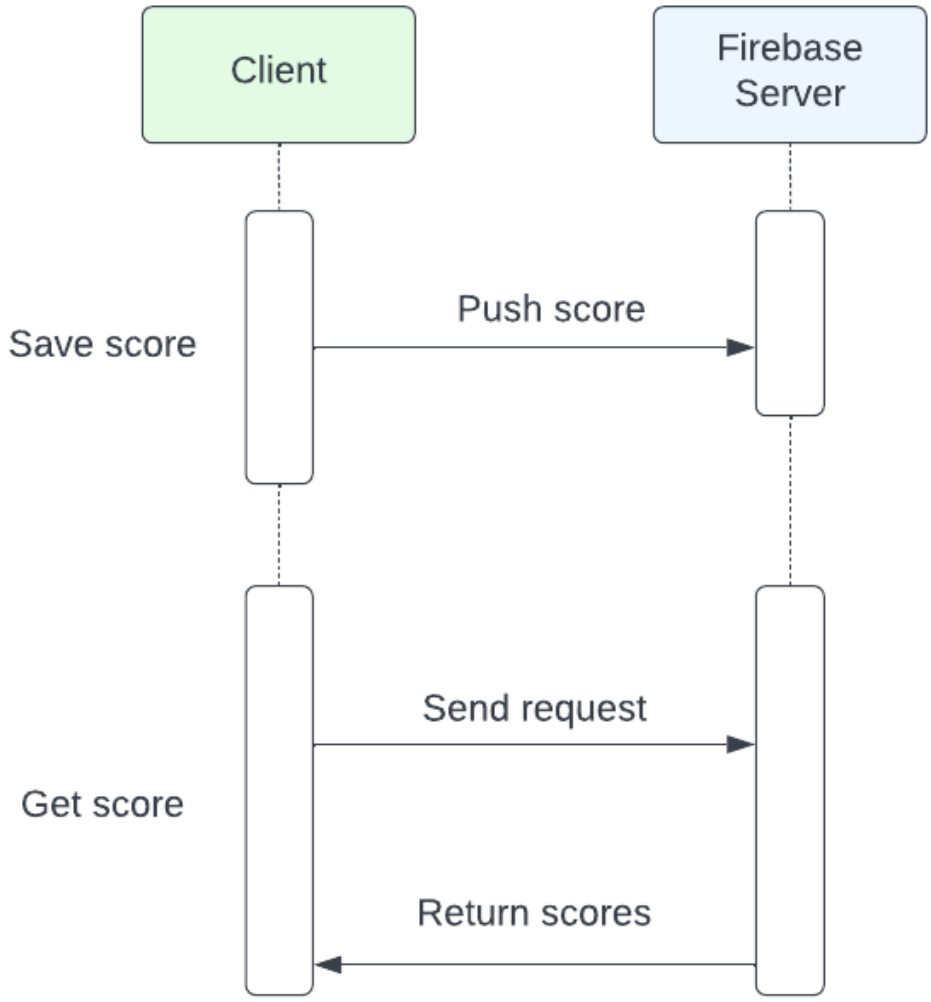

Figure 8: Overview over client-server interaction

7.4 Logical view

The logical view in Figure 8 is intended to show how the architecture of the game is structured with a class diagram. The idea is to show this during the game, i.e the game screen, and not all the menu screens and such. This is to focus on how the classes related to game logic are interconnected. From the figure, it can be said that its a kind of hierarchy. On top is the ScreenManager with

the GameScreen beneath. The GameScreen has one main component, which is the GameWorld. Underneath there are GameObjects consisting of views and models, with Box2D bodies. We have controllers and our physics engine provided by Box2D. Using this approach, the models can be decoupled nicely and the role of the MVC-pattern can be somewhat interpreted from the illustration. To show the full scope of the game one could make diagrams for all the other screens, and show what classes would be used in what way, but we deemed it redundant here.

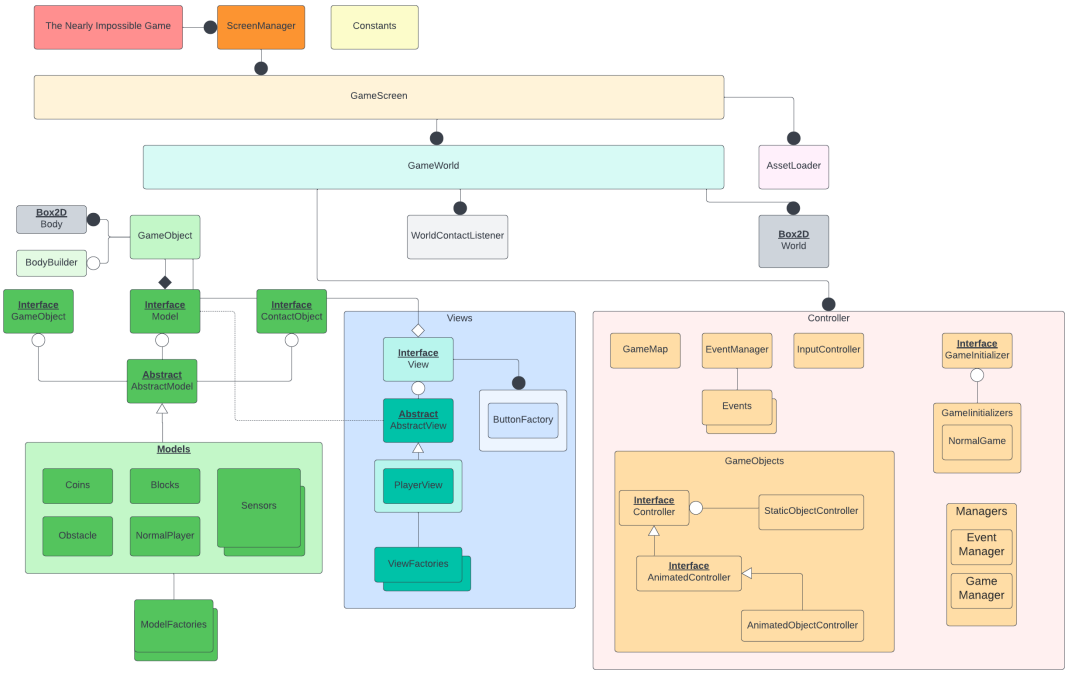

Figure 9: Logical view

The logical view with all the screens in figure 9 is intended to show how our interactions with Firebase are set up. This diagram is pretty basic. All external utility classes imported that we did not make, only used to display text, images or handle input, are defined absractly as 'utility classes'.

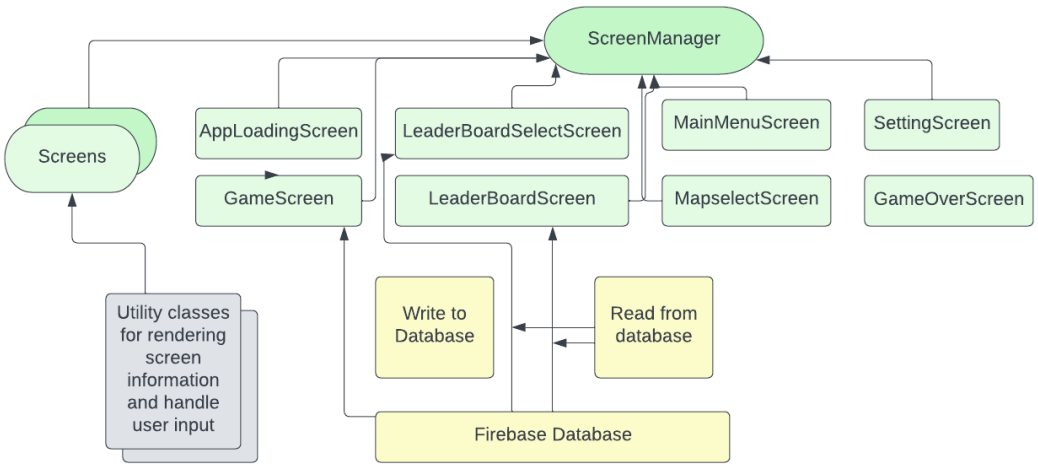

Figure 10: Logical Screen Diagram

7.5 Physical view

Figure 11 and Figure 12 shows how the Android mobile devices are connected to Firebase through Wi-Fi, using the client-server pattern. After the game is finished, a request is sent to firebase with the name and the score of the player. Firebase then updates the database in real time and synchronizes with other devices.

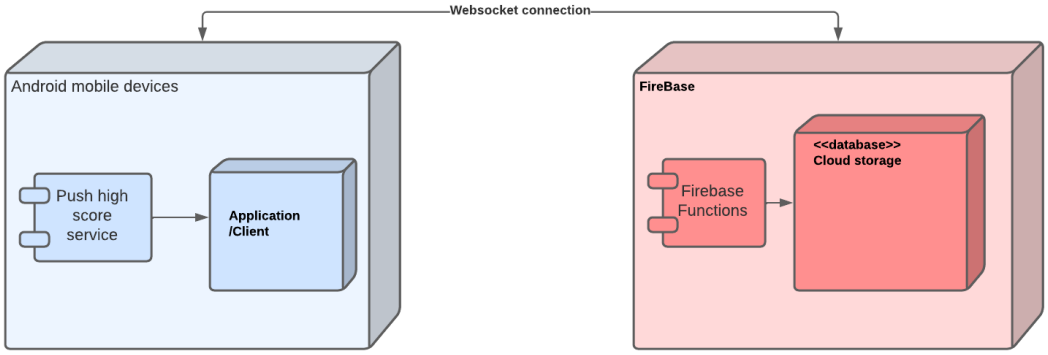

Figure 11: Deployment Diagram

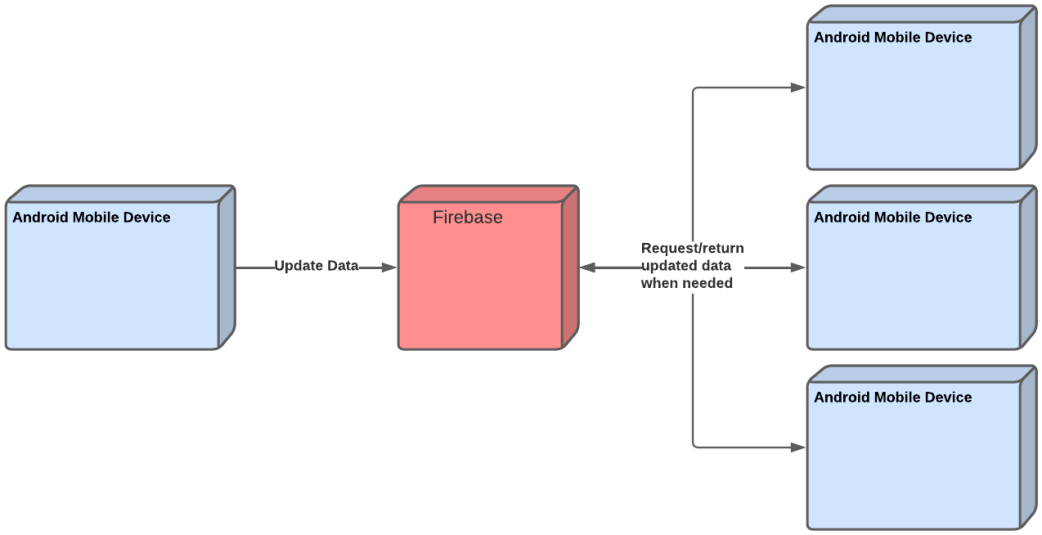

Figure 12: How clients are updated

8 Consistency among architectural views

No inconsistencies.

9 Architectural Rationale

In this section we explain the motivation behind the choice of architecture, how it will work and why it will fulfill the requirements for our project.

9.1 Model-View-Controller

The most important requirements for this project are modifiability and usability. To fulfill these requirements we have chosen to build our system with Model-View-Controller architectural pattern in mind. MVC promotes seperation of concerns, therefore changes in a module doesn't affect the other modules, hence increasing modifiability. This seperation also enables us to work independently on different modules in parallel, which is crucial given the limited time-frame. Additionally, the modules only uses the interfaces of each module and can therefore be reused in other projects in the future. Since it is difficult to get the user interface perfect the first time, it is essential to be able to change the UI easily to create a better user experience from feedback from the stakeholders. MVC is a good fit for this project, but it isn't perfect. For example, since the UI will be quite simple, MVC might introduce unnecessary complexity to our system. Also the pattern might add some latency to user interactions, but we believe this will be minimal for this type of game without too much complexity.

9.2 Client-Server Pattern

The client-server pattern ensures loose coupling between the clients themselves, and the clients and the server, hence increasing modifiability. The number of clients is easily scaled, only limited by the capacity of the server. Since we're using Firebase as our server, there shouldn't be any issues with the capacity. Availability could be affected by the robustness of the server provider. However, opting for an established platform like Firebase should mitigate availability concerns, versus a self-hosted server.

9.3 Singleton

Using the singleton pattern will be tremendously helpful to our architecture. The reason we chose this is because it will be useful for not creating multiple instances of classes that are not needed. One example of this is the mentioned factories we will have for creating the different element of our game. These factories will be

instantiated by using the singleton pattern, such that one factory is used for all production. Use of singleton will because of this help with modifiability.

9.4 Factory method

The main reason for choosing to use factories in our architecture is to have an efficient way of creating many different views or models with common elements, but also to help with modifiability. it will be easier to make changes to views and such if they are created in a factory class. In addition, a significant reason we chose to use factories was that expanding the game in size in the future will hopefully be easier.

9.5 Strategy

The strategy pattern creates a family of interchangeable algorithms that are only selected at runtime based on the configuration of the game, or user selection. This allows for us to improve the modifiability of the game, as new game modes like Challenge or Zen can be added, and we simply change the algorithm implementation in the new game mode, and the game will continue to work.

9.6 Template method

The template pattern is overall useful for making more efficient code, where classes that use a lot of the same functionality, but also have differences, can share code between them instead of writing duplicate code.

9.7 Architectural Tactics

Using tactics known to increase modifiability, usability and availability will be essential to meet the requirements for this project. Ensuring loose coupling between the modules of the system are one of these tactics. For example, by encapsulating the network service (Firebase) with an interface we give us and future developers the possibility to swap it out for a different service in the future. Having strong cohesion reduces the need for coupling between the modules, thereby improving modifiability. Usability on the other hand is more subjective, but keeping the controls simple and intuitive will especially help with meeting the requirement U1 in the requirements document. For availability our choices of tactics are quite natural, with focus on exception detection/handling giving availability by ensuring a predictable runthrough of the game without unforeseen events.

10 Issues

No issues

11 Changes

This section describe the changes carried out in the document from the first draft until the final delivery.

1 Introduction

- Fixed small typing errors + punctuation

2 Architectural Drivers

- Added subsection for availability

4 Selection of Architectural Views

- Added logical view

5 Architectural Tactics

- Added tactics for availability
- removed usability tactics that didn't affect architecture

6 Architectural/Design Patterns

- Removed broker pattern and observer pattern from description because they were not implemented. They were both deemed to be unnecessary and would not really make things better

7 Architectural Views

- Added Logical view
- Altered development view to better function as a way to split tasks
- removed tactics not used in the final product

9 Architectural Rationale

- Added rationale for patterns that didn't have it before

12 Individual contribution

Everybody contributed equally.

References

[1] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. *Head First Design Patterns*. O'Reilly Media, 2004.

[2] Len Bass. *Software Architecture in Practice*. Addison Wesley, 2013.

[3] Felix Bachmann, Len Bass, and Robert Nord. Modifiability tactics. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2007.

[4] Steve McConnell. *Code Complete*. Microsoft Press, 2004.

[5] Carl-Fredrik Sørensen. Design patterns. 2021.

[6] Alf Inge Wang. Chapter 13: Patterns and tactics. 2022.

[7] Oop good practices: Coding to the interface, 2017. URL `https://medium.com/javarevisited/oop-good-practices-coding-to-the-interface-baea84fd60d3`.