Architecture Documentation

The Pendulum Paradox|

GROUP 22
Rune Strøm Brekke
Tore Dybdahl
Andrea Falk Lind
Henning Einar Luick
Matěj Mňouček
Simon Smeets

COTS: Android Studio, LibGDX and Google Play Services
Primary Attribute: Modifiability
Secondary Attributes: Performance, Usability and Availability

May 12, 2019

Contents

List of Figures

List of Tables

1 Introduction

1.1 Project Description

This architecture documentation is written as part of a project in the course TDT4240 Software Architecture at the Norwegian University of Science and Technology. The project work will be conducted during the spring semester 2019. The participants are Rune Strøm Brekke, Tore Dybdahl, Andrea Falk Lind, Henning Einar Luick, Matěj Mňouček and Simon Smeets.

The goal of this project is to make a functional multiplayer game that can run on a smartphone. Our group uses Android as our game platform, and the intention is to implement a real-time multiplayer game. In our work, we will use several different architecture and design patterns. During the process, we will focus on designing the game and implementing it correctly. We will then evaluate it and conduct testing of the result. We focus on making a high quality game with respect to important qualities in software architecture. Our targets are described in sections 2 and 3.

The project officially started 12th of February and will continue for approximately 12 weeks, ending on the 29th of April. During these weeks, the project will be divided into three phases: "Requirements and Architecture", "Evaluation (ATAM)" and "Design, Implementing and Testing". What is more, we will also contribute with feedback to other groups taking the same course, as well as receiving feedback.

In the current phase, Architecture design, we will focus on specifying the software architecture of our game, and provide an architecture that satisfies the requirements specified in the requirement documentation. Architecturally significant requirements are requirements that play an important role in the design of the architecture of our game. In this phase, we will focus on uncovering these, as well as specifying architectural tactics, architectural patterns and design patterns in order to design an architecture which fulfils our requirements. Furthermore, we will look into views in order to communicate, understand and describe our architecture.

1.2 Game Description

The Pendulum Paradox is a platform game, and therefore a subgenre of action game. The player will control a sprite object, and advance through different levels, starting in a forest. During the game, the player will face

several obstacles and enemies will attack and attempt to kill him. The basics of the game dynamics are borrowed and inspired from the games Jazz Jackrabbit, Super Mario Bros., and A Great Adventure.

The game can be played in single player and multiplayer mode over a real time network. During each level the two players will have sci-fi characters that are lost in time.

Both players will be equipped with a basic shooting device in order to defend themselves.

The players will move through the game world, where they get points for shooting enemies and collecting gears. They can also collect artefacts that can improve their survival skills. The artifact will only last for a specified amount of time after the player has collected it.

Both players have to survive to the end of the current level in order to advance to the next one. If one of the players loses all of his lives, both players will die. The players lose lives when they collide with an enemy or fall in the water, where the last will result in total death. The game works as an endless runner game, and will proceed through levels until one player dies.

The game is played on a smartphone. Buttons in the left and right corner of the screen will provide functionality for the player to move his play character left and right, and give him the possibility to jump and shoot.

Figure 1 to 3 show screenshots from three different platform games. These are all used as inspiration for our game.

Figure 1: Screenshot from the platform game A Great Adventure

Figure 2: Screenshot from the platform game Super Mario Bros.

Figure 3: Screenshot from the platform game Jazz Jackrabbit

1.3 Document Structure

This document contains a description of the architecture of our game. In chapter 2 we will give a description of our Architecturally Significant Requirements. We present functional requirements and the quality requirements Modifiability, Performance, Usability, and Availability. Furthermore, the business case of the game is presented. Chapter 3 gives an overview of the stakeholders of the project, and their concerns. These will include the project team, end users, teaching assistants and professors, and the ATAM evaluators. Chapter 4 gives an overview of architectural views. Chapter 5 gives a detailed description of the architectural tactics that are used in the architecture in order to satisfy our quality requirements. In Chapter 6, we present patterns that are used with the some purpose. Views will be described and shown in detail in chapter 7. These include logical view, process view, physical view, and development view. We finish the chapter with a

section about the consistency among architectural views. Chapter 8 will focus on the rationale of the architecture. The last two chapters will include issues and changes

2 Architectural Drivers / ASRs

Both functional requirements and quality attribute requirements are specified and described in detail in the Requirement documentation. This is a short summary of the main drivers affecting our system's architecture.

2.1 Functional

2.1.1 Networked Multiplayer Game

Our main functionality will be that we have a networked, real-time multiplayer game. Both players use different devices and are able to witness the other player's moves. In order to achieve this, we use a peer-to-peer architecture. Furthermore, it requires that a real-time highscore is available on both play devices, and the delays are as small as possible. This will be reflected in the architecture, as it has to be structured in such a way that as few packages as possible are sent.

2.1.2 Controllable character

The game has to provide functions for the player to be able to move his character. The players should be able to make his character move left/right, as well as jump and shoot. As not all parts of the game are set in stone at this point, making sure that the character actions are reusable when designing the architecture can save us from valuable time in case we intend to make other entities with similar functionality.

2.1.3 Multiple levels

Our game consists of several difficulty levels. This functionality should be reflected in our architecture. Each level will contain the same basic functionality, but there should be room to make individual specifications. Ideally the level loader will load all levels in the same way, regardless of difficulty, but this requires us to build a complex level loader, and we have to be careful to not sacrifice modifiability to achieve this.

2.2 Quality

2.2.1 Modifiability

Our primary quality requirement attribute is modifiability. The architecture we create has to reflect the fact that the game should be easy to change

and extend. We will use a Model-View-Presenter (MVP) pattern to divide the program into data and user interface. We will also make use of other patterns, for example Abstract Factory, to make it relatively easy to add new play sprites, new levels, new power-ups, and new enemies. See section 6 for list of patterns we wish to use.

2.2.2 Performance

Since our game is a real-time, networked game, it is crucial that we achieve consistency between what the two players observe on their devices. The architecture of our game has to support a high enough FPS in order to achieve this. It is also important that the search for another player happens relatively fast, and the player is notified if another player is not found.

2.2.3 Usability

The user should find our game easy to understand and fun to play. We have to make it easy for new players to understand the game logic. Our architecture must be able to handle input from users, and give visual or auditory feedback based on input from the user.

2.2.4 Availability

Our last quality requirement availability. Since our game is networked, it is important that the game is available at any time. So, our architecture has to support a system with minimal downtime.

2.3 Business constraints

The primary goal of our implementation, is to make a good architecture that is able to achieve the functional requirements and the quality attribute requirements of our game. Due to time constraints, we focus on making a minimum viable product (MVP) to begin with, with an architecture that is easy-to-follow and sufficient to complete the given assignment. With our primary attribute as modifiability, the game should be easy to extend later on.

What is more, we intend to make a game that the user finds fun and enjoyable playing. Some of the team members have experience with game making from previous projects, while other members are new in the field. Our common goal is to make a game that are members can be proud of. The architecture

should be so that independent of the experience level of the members all should be able to describe, explain and understand how it is implemented.

2.4 Technical constraints

2.4.1 Java and Android Studio

We will be using Android Studio as our API with Java as our programming language. Android studio is extremely easy to set up with libGDX, which makes getting started very easy. We see no immediate advantages of electing to go for any other API as we have decided to develop for Android phones, and we all have some experience with it beforehand. Java has it's own constraints and coding standards we have to follow, but there are no true limitations with java with regards to our game other than perhaps varying familiarity with the language.

2.4.2 Google Play Games Services

When using Google Play Games Services as our multiplayer and highscore hosting solution, we can leave a lot of the responsibility to Google. This comes at the cost of losing some control over the package flow that a custom tailored networking solution would give us. We consider this trade-off worthwhile, as networking is not the primary focus of this project.

2.4.3 LibGDX

Using LibGDX as our framework lets us focus on the architecture and skip a lot of the very low-level game development things such as configuring rendering and the update loop. LibGDX is still fairly low-level, and should provide a lot of room for learning about architectural implementations. We also elected to use Ashley, a simple entity system for LibGDX, to alleviate some of the extra work that comes with making a game from the ground up.

3 Stakeholders and Concerns

Anyone who has an interest, or a stake, in the development of our game can be described as a stakeholder (Bass, Clements, & Kazman, 2013). Bass et al. (2013) explains that stakeholders typically have different concerns and focuses on what the system developed should satisfy. This section will describe several of the stakeholders for our project, and see how their concerns can be connected to the architecture.

3.1 Project Team

The main concern of the project team, is to be able to make a game that satisfies the project requirements and that each member can be proud of. The members wish to get a high grade. In order to achieve this, we hope to deliver documentation that is of high quality, make a structured and well-organised architecture where relevant patterns and tactics are used and explained in detail in the documentation. What is more, we are invested in creating a group environment with good cooperation, and where everyone is able to learn something new, independent of experience level from beforehand. Viewpoints will be of importance during the entire project life-cycle. The development view gives a view of task-dedication, while sequence diagrams are used to reflect the different modules relations. We also use the process view to achieve an effective task-allocation during development.

3.2 End Users

Our end users, the game players, will be concerned with how easy and flexible the game is, the usability of the game. The player should quickly be able to understand how the game is played, and once playing they should find it enjoyable. We provide a textual tutorial with the intent to make it easy for the player to understand the functionality of the game. Most often, the user is not interested in documentation, however the logical view could be of interest as it provides information about features and functionality. These are probably the characteristics that a user potentially would be most interested in.

3.3 Teaching assistants and Professors

The teaching assistants will judge our game based on the requirements of the project. They will therefore be interested in all aspect of the system. Especially, what the team members have learned during the project will be

in focus. It is therefore of importance that the documentation we deliver is of high quality, making the evaluation easier for the teaching assistants. The architectural and design patterns we choose to implement must be implemented correctly and be thoroughly documented. All viewpoints will be of interest for this group of stakeholders.

3.4 ATAM Evaluators

In order for the ATAM evaluators to be able to evaluate the architecture of the system, they require professional documentation explaining it in detail. Detailed architectural view will be of utmost importance in order for the this group of stakeholders to get an overview of the system. It is important that the evaluators are able to understand our game and the rational of the architecture behind it.

4 Selection of Architectural Views (Viewpoints)

The viewpoints are based on Kruchten's 4+1 model (Kruchten, 1995). The viewpoints we use are presented in table 1.

View	**Purpose**	**Target audience**	**UML Notation**
Logical	Concerned with the system's functionality to end-users	End users, teaching assistants, project team, ATAM evaluators	Class diagram
Process	Communicates non-functional requirements by showing the processes and communication between them	Project team, teaching assistants, end users, and ATAM evaluators	Activity diagram
Physical	Maps the software to the hardware, and is primarily concerned with the non-functional requirements	Project team. teaching assistants, and ATAM evaluators	Deployment diagram
Development	Decomposes the system into subsystems . Shows the software module organisation of the development environment	Project team, teaching assistants, and ATAM evaluators	Component and Package diagram

Table 1: Architectural views

5 Architectural Tactics

This section describes the architectural tactics that will be used to ensure that the quality requirements will be met by being able to respond to stimuli.

5.1 Modifiability

Changes in code are inevitable, and modifications can become very expensive, especially with a large codebase. A good way to reduce this cost is to plan for future modifications by making the code easy to modify or replace. Modifiability can accompany overhead, because modifiability often requires reducing the responsibility of the individual modules, resulting in an increase in information being passed between them. The overhead is often a fairly small percentage of the total execution time when the modifiability is properly implemented, and is not considered a high risk compared to other execution time sinks.

5.1.1 High cohesion/Low coupling

Among the most effective ways to increase the modularity of a code base is to have high cohesion and low coupling. Cohesion defines the strength of the relation between the functions of a module, while coupling describes how dependent modules are to one another. Ensuring that the modules have high cohesion and low coupling not only significantly improves the readability of the code, but also reduces maintenance and modifiability cost by a large margin.

Modules will have clearly defined roles and limits to help enforce this tactic. Modules which can not be clearly defined will have to be refactored into multiple modules, while modules which are highly similar can possibly be merged by refactoring them.

5.1.2 Information hiding/encapsulation

Member variables will be made private to further reduce coupling by disallowing direct access to said variables from other objects. Hiding implementations of objects behind interfaces, more commonly known as encapsulation, will also be incorporated. Interfaces make the code highly modular by allowing whole modules to easily be replaced with few to no consequences given a proper implementation of the new module. Encapsulation can also make planning the code skeleton easier as you can estimate which methods each

module requires and better estimate the cohesion of the module without requiring an implementation.

5.1.3 Intermediaries

There may be cases where the previous techniques are insufficient. For instance in cases where you have a module that is tightly coupled with another module, but removing the strong coupling proves to be difficult. Intermediaries will be implemented in such cases to act as a bridge or layer of communication between modules.

5.2 Performance

It is generally difficult to predict whether or not a game will experience performance issues before completing at the very least a prototype, especially considering the wide variety of hardware and operating systems available. We will abstain from sacrificing modifiability preemptively for the sake of performance. However, there are general good practices which will be taken advantage of without sacrificing modifiability. Overhead from the modifiability will be cut in extreme cases of unexpected performance hits.

5.2.1 Asset management

One can completely eliminate loading times for resources during gameplay by preloading all resources during a loading screen or similar. This comes at the cost of RAM and longer time before the game starts, but also guarantees that all the assets are available before they are required. Preemptive resource loading will be done concurrently to save some loading time. The game also assumes that the last accessed assets are the ones that are the first required ones, and assets are therefore stored in quick access LIFO containers. The last accessed assets can also be cached due to the same assumptions as for the LIFO containers.

We will eliminate the need for managing the sampling rate of our graphical and sound assets by using appropriate image and sound formats/sizes.

5.2.2 Resource management

Only required events which are subscribed to will be broadcast across the modules, meaning that irrelevant touch events and other input events will be

discarded. This will help prevent too many events being queued up, which should remove the need for limiting and prioritising events.

Major parts that can easily be done concurrently will be processed in parallel with multithreading. By only focusing on parallelising major parts of the code, we spend as little time as possible on implementing the concurrency, while gaining the most performance out of it. Examples of major parts are: Physics, networking, rendering. Proper structuring can also reduce blocked time.

5.2.3 Network tactics

To ensure we meet our networking requirements, we will limit the data that is transferred to the bare minimum. Interpolation and extrapolation of entities will be applied in cases of packet loss or other data loss to hide the latency. Assets will be stored locally on all devices, eliminating the need for sending large amounts of data across the devices. Packets will be processed FIFO, as they are assumed to be received chronologically.

5.2.4 Other

When bottlenecks are discovered, either through profiling or other means, they will be resolved by refactoring the relevant part of the code by for instance replacement, adding concurrency or exchanging the algorithm.

5.3 Usability

Apart from navigating the menus and waiting for network connectivity, all of the time spent by the user will be during gameplay. It is therefore paramount that it is explained to the user how to accomplish the desired task of eventually beating the game. An optional tutorial will be presented to the player which will teach the basics of the game.

The nature of touch screens limits the amount of actions the user can take at any given time. This allows the system to take initiative in some situations and register a button press in cases where the user just barely misses the button with their finger, as the system can assume that the intent was to hit it.

Additionally, the system will grant feedback to the user in form of sounds and responsive controls as a response to user initiative such as colliding with

collectibles, attacking, getting hit, or using abilities.

5.4 Availability

We will be using an external service and API, namely Google Play Games Services, for network communication. Using an external service which we trust allows us to focus more on the game architecture, and less on the networking architecture. We will still be detecting faults and handle the exceptions, but recovering and prevention will be done entirely by Google.

6 Architectural and Design Patterns

This section presents applied architectural and design patterns, discusses how they work and what their purpose is.

6.1 Architectural patterns

Several architectural patterns are used in order to create proper problem abstraction and achieve good structural organization of the system. In addition, the used architectural patterns support chosen quality attributes.

Model View Presenter (MVP)
Model View Presenter was initially derived from an older similar pattern called Model View Controller (MVC). The main goal of this pattern is to separate data from the application logic and also the way data are presented from the rest of the application. For this reason, the pattern utilizes three main building blocks: model, views and presenter. The presenter is the core part of the pattern which encapsulates the application logic and acts as a mediator between views and a model. Views consist of user interfaces and level scenes. The model contains data structures and game mechanics. The whole concept helps to reduce system coupling and supports good modifiability.

Entity Component System (ECS)
The model part of Model View Presenter is further organized according to the Entity Component System architectural pattern. The game model is fairly complex, so it definitely benefits from additional structure. The main three parts of the pattern are: components, entities and systems. Components are the only data part of the model. Entities just organize those components into larger data units. All the logic and game mechanics reside in systems.

Mobile Backend as a Service (MBaaS)
The game also utilizes multiplayer over network, so a specialized backend service is used. Google Play Game Services were chosen because of the good compatibility with Android platform and proven quality. In terms of architecture, the solution internally uses P2P network networking for message exchange between all players and also provides additional functionality like authentication, leaderboards, achievements and so on.

6.2 Design patterns

Aside from architectural patterns the game also uses several design patterns for dealing with recurring software design difficulties on a smaller scale. The choice of particular patterns is driven by the quality attributes.

6.2.1 Creational patterns

Abstract factory
Abstract factory pattern provides a unified way of creating related or dependent objects and also abstracts from details about the actual creation procedure. The game uses it for easy and straightforward model component creation.

Builder
The Builder pattern is focused on flexible object creation. In this game, the pattern is closely related to abstract factories. It is used for composing entity objects from individual existing components.

Singleton
The Singleton pattern maintains the life-cycle of one particular object instance and also provides a central access point to it. As singleton is not considered a good solution in terms of reducing coupling, the game avoids it as much as possible. In fact, the only place where it is used is the Dependency Injection container.

Dependency Injection
The Dependency injection strategy is to provide a system with particular objects instances during runtime. Requirements for each object are specified in forms of interfaces. This approach greatly reduces coupling and also support easy modifiability of the whole game.

6.2.2 Structural patterns

Proxy
The idea of proxy patterns is to have a class that represents the functionality of another class. This strategy is really beneficial for the game's multiplayer solution. This pattern hides the fact that the synchronisation happens over network and provides the same interface like a local multiplayer class would.

Marker
The Marker pattern is a really simple and easy to implement concept. It consists of an empty interface whose presence associates some kind of meta data with the implementing class. The game uses it for marking classes injectable and therefore ready for dependency injection.

6.2.3 Behavioral patterns

Chain of responsibility
The Chain of responsibility pattern builds on the idea of having a source object and several processing objects that modify behaviour or handle events originating from the source object. This is a great fit for handling temporary in-game objects modifications (e. g. power-ups, special abilities...). The pattern also reduces coupling between source objects and actual processing objects which produces reusable and modifiable solution.

Observer
The Observer pattern introduces events and event handlers. The idea behind them is to automatically notify particular objects about modifications of others. This approach goes along with Model View Presenter architectural pattern as the separate building blocks need reliable communication without unnecessary coupling.

State
The State pattern provides a way to capture and handle various states of objects and also a possibility to react to eventual state transitions. The game makes use of this approach mainly in handling different view states which introduces a well-defined structure which is also easy to extend or modify.

6.2.4 Sequencing patterns

Game loop
The role of game loop is simple yet important: decoupling game time progress from inputs and hardware speed. This pattern is already embedded in LibGDX library but is an important part of the design that is worth mentioning.

Update method
The Update method is closely related to the game loop pattern. It adopts

the same approach like is used in computer simulations. In fact,it simulates the behaviour of objects by advancing their behaviour progress every game frame.

7 Views

7.1 Logical View

Figure 4 shows the diagram for the logical view. At the core of the game is the GamePresenter class which connects the different parts of the game together in accordance with the model-view-presenter architecture.
The view is comprised of states implemented in the ViewState class. The StateMachine class contains all the possible states of the game as well as transitions between the states. Each ViewState is made up of one BaseScreen and one Scene. BaseScreen has a Stage which is the input processor, and contains labels and buttons. Using an observer pattern a button will trigger an Event when pressed which notifies subscribing EventHandlers so that actions can be performed by the game. The Scene class contains the background, a TiledMap which is the level, as well as entities which are actors in the game such as players, enemies and collectibles.
For the model we are using the Ashley entity-component-system provided by LibGDX. Each entity contains components defining it. Components are simply data containters. EntitySystem processes entities, and the engine manages entities and EntitySystems. For simplicity we have not modelled all implemented classes of the Ashley interfaces here, however we have chosen to include ControlComponent. The ControlComponent has a ControlModule which can either take inputs from the user and invoke the appropriate event on an entity, or it can have a CommandQueue that invokes Events in a certain order and is used for the pseudo-AI controlled enemies. The NetworkSynchronizationProxy is used to communicate with Google Play Services for setting up game rooms, saving highscores, and keeping two devices in sync when playing multiplayer.

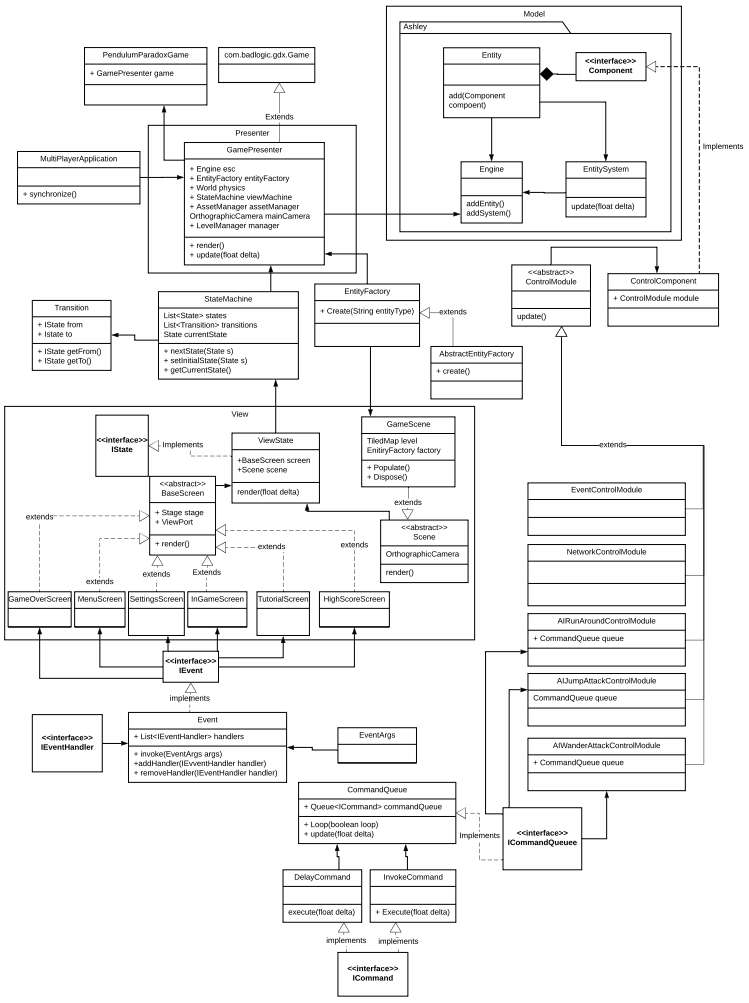

Figure 4: Logical Viewpoint

7.2 Process View

Figure 5 shows the state machine for the application. It shows the relationship between the views, and is a good way to show some of the architectural decisions.

When the game launches you will start in the MainView. Here you can directly access most of the other views by pressing the name of the respective view you want to navigate to. From the different views, it will be possible to get back to the MainView by clicking the back arrow in the top left of the screen. Some architectural decisions that is worth noting is that you can only access the SettingsView from the MainView. Another thing is that there is no end state, since the application can be quit from all views using the home button.

To access the highscores, you will communicate with the Google Play Games Services, to always have the highscore updated. There are three separate ways to play our game. Single player, multiplayer quick play with a random player, and multiplayer with a friend. Google Play Games Services will be used when matchmaking in multiplayer, as illustrated in Figure 6. When a player dies, the game is done and the onGameEnd-function will be called and the player(s) will be returned to the MainView.

Since you have the possibility to play with random players and the games will be quite short, we do not have a pause/ resume option, but it is possible to quit the game by returning to the MainView.

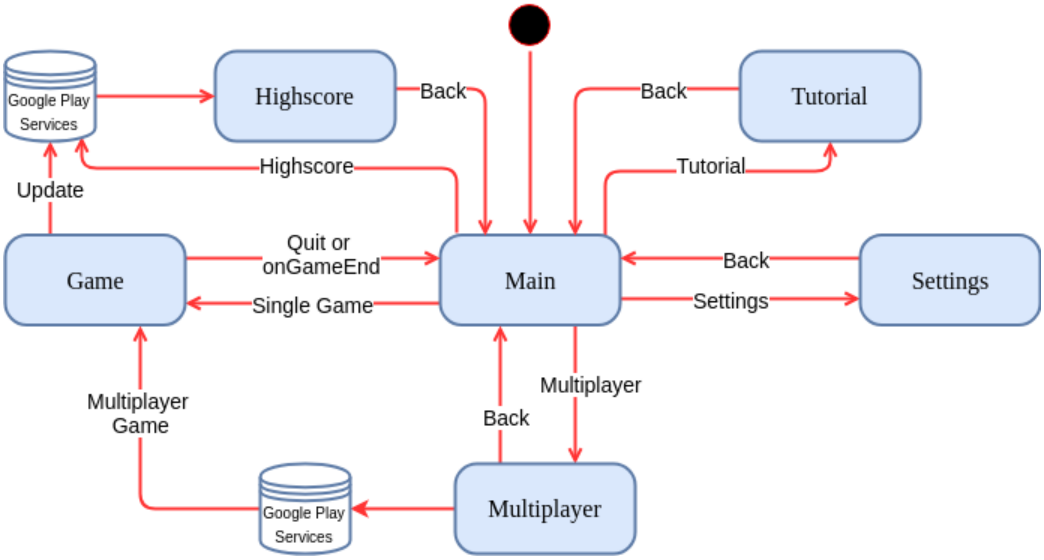

Figure 5: State machine, process view

Playing multiplayer

Figure 6 illustrates how two random players are matched. This case represents two players, Client A and Client B, who want to play the game cooperatively. Each of them request joining the the matchmaking system individually. The players who have joined the queue will be notified by the matchmaker that lets them know that they are ready to be matched. Once the Google matchmaker has enough ready players in the queue, the players will be notified and the game starts.

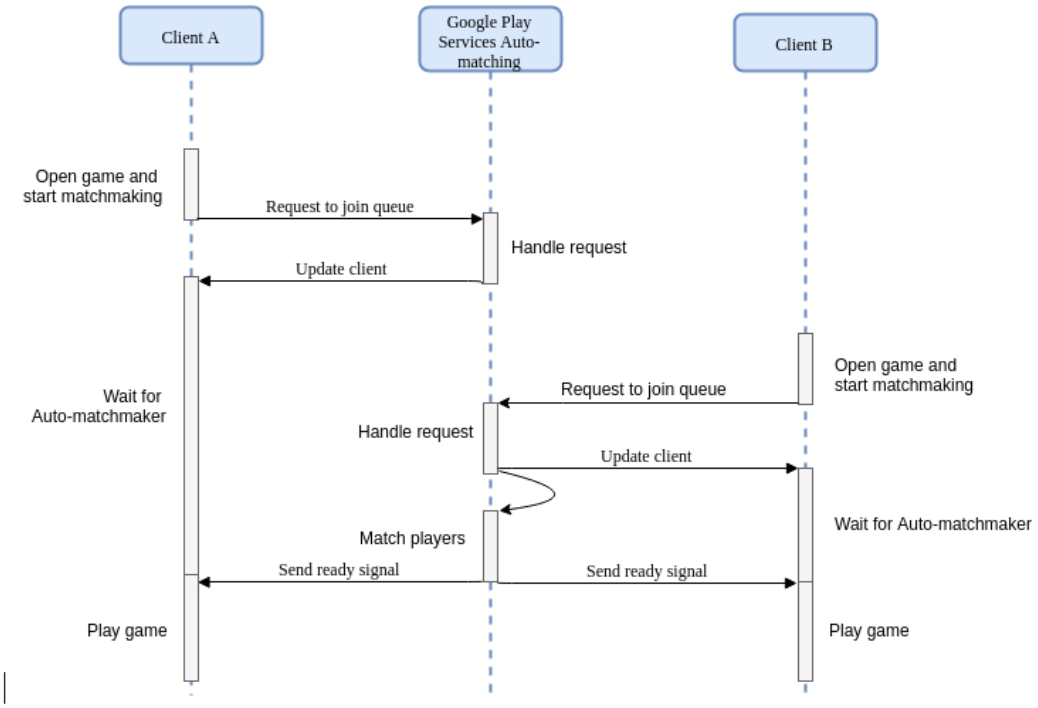

Figure 6: Sequence diagram of starting a multiplayer quick game with random opponent with the Google Play Server Services

A player leaving the queue is not depicted, but the player will be dropped from the matchmaking when he leaves the queue voluntarily or exits the game in some way.

7.3 Physical View

figure 7 shows a diagram describing the deployment of an application onto an Android device. The apk archive packages code, data and resources along with a manifest file with requirement specifications. Android uses a Linux operating system that serves as an abstraction layer between the hardware and the rest of the software stack, mapping the application's functionality to the appropriate hardware components. (Android Application Deployment, n.d.) We are using Google Play Game Services to facilitate matchmaking between players and to host game rooms. The device will connect to a server using Wi-Fi or the cellular network to find an opponent and establish a game room. Google Play Game Services then sets up a peer to peer mesh, over which the devices will communicate during the game. If latency proves to be to high, or messages per second exceed the threshold, the switch to unreliable, UDP style messages can be made.

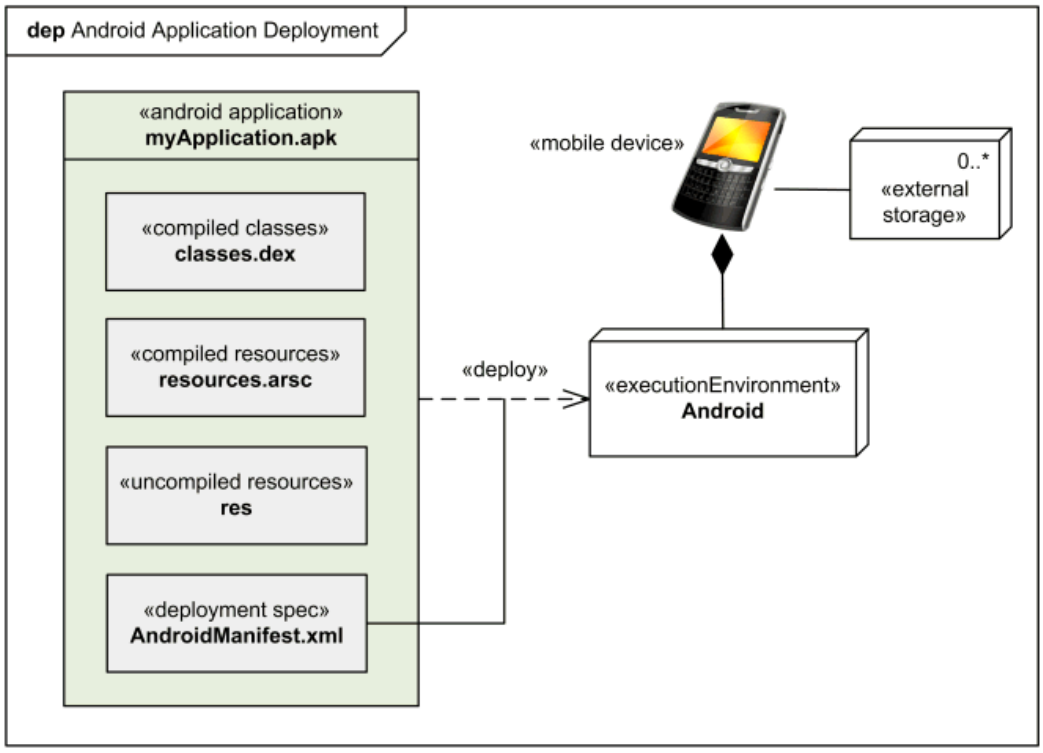

Figure 7: Android app deployment diagram

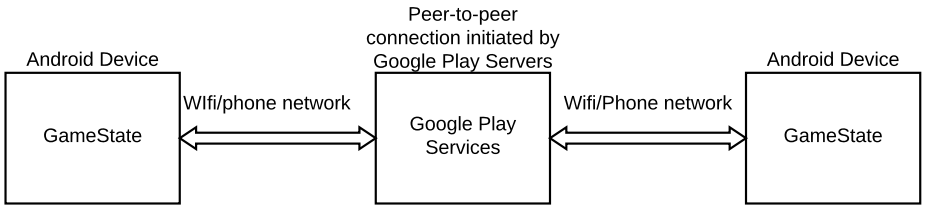

Figure 8: Physical View

7.4 Development View

The package diagram shows the different packages in our code and the classes within the respective packages. Interfaces are marked with a green background, abstract classes with a yellow background, exceptions with red background and normal classes with blue background. Most of the packages contain interfaces and/or abstract classes, that several classes of similar nature can implement or extend. This can be done simultaneously by several programmers so as to spread out the work load. There are individual packages for the model, view and presenter of the game in order to stay in line with the MVP pattern. Note that the view and model package contain other packages within them.

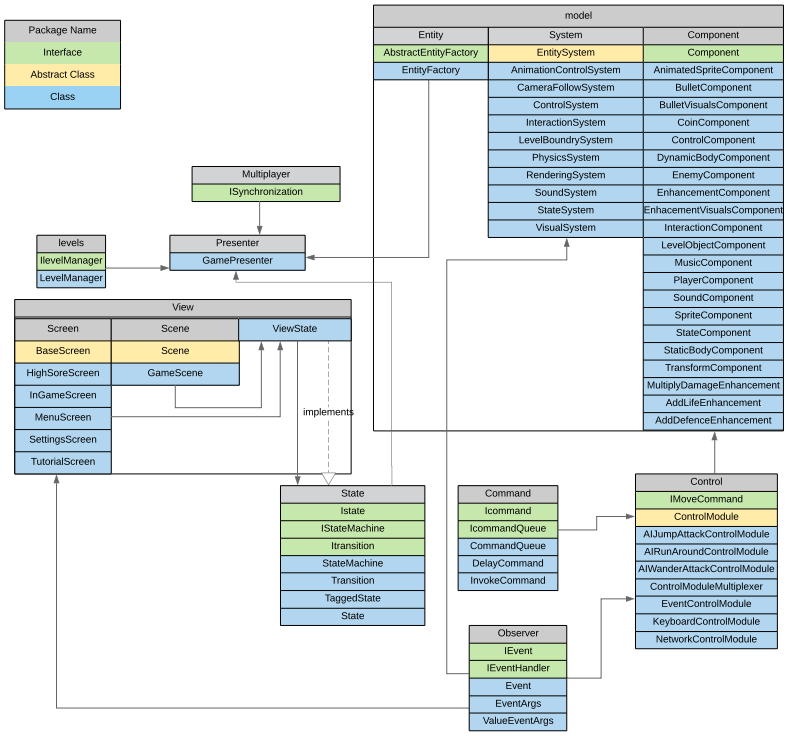

Figure 9: Development View

7.5 Consistency among architectural views

We do have a few inconsistencies between the different views. Our logical view shows the MultiplayerApplication which facilitates commmunication with Google Play Services and the other device during multiplayer, but does not go deeper into how this is done. The physical view does not specify how the the inner workings of the application connects to Google Play Services, only the physical components that are communicating. The development view shows all classes of the game, even ones not present in the logical view. The development view only shows packages that have dependencies, but does not show the show specifically which classes within the package that is being used outside of it in the same way logical view does. To keep the logical view simple we have for example only included the interface for components, but not all the concrete classes suggested in the development view. In the process view we present the flow of the game and communication with the servers and the other device. What causes these state changes and messages to be sent between devices are the Event classes in the logical view which are

triggered by button presses and other events within the game.

8 Architectural Rationale

The game is designed with all the chosen quality attributes in mind. Several solutions were considered as the main architectural pattern: MVC, MVP, MVVM and VIPER. It turned out that VIPER is too complicated for a project of this scale, LibGDX lacks support for proper data binding necessary for MVVM and MVC introduced additional coupling between model and views. Therefore, the golden mean, more precisely MVP, was chosen.

The next problem that arose was the complicated model part of MVP. The plan was to further organize it and set additional structure restrictions. After a bit of research, the ECS architectural pattern showed up as a commonly used and proven solution and several tests have proven the same. Furthermore, ECS builds on flexibility and modifiability which corresponds to the primary quality attribute. As it is expected that the ECS model will get fairly large in size, the builder and abstract factory design patterns will be used for easier creation and tear up of various components and entities.

The view part of MVP also suffers from great complexity. To tackle this issue, the state pattern will be involved. It will define a set of possible view states and transitions between them which will make views more consistent and easier to manage.

To reduce coupling between various parts of the system,the observer pattern and an event system based on it will be implemented. The event system will also have the possibility of passing data inside its events in order to increase the range of possible use cases.

The availability and performance attributes are to a great extent achieved by using the Google Play Games Services back end as these features are guaranteed by it. When it comes to application performance, all the code is written with efficiency in mind and several optimisation strategies were executed in order to eliminate performance losses.

The usability is achieved by good UI/UX design and simple input controllers that complies to solutions that mobile gamers are used to.

9 Issues

No issues so far

10 Changes

Date	Change	Reason for change
30.04.2019	Removed separate Ashley diagram from logical view. Added classes from the control and command packages to logical view and development view. Some other changes has been made to the views	Based on feedback and better understanding of how classes work together their places in the diagram became clearer
02.05.2019	Added "Availability" to the architectural tactics.	The feedback reminded us that we should specify tactics for availability because we have requirements for it.
02.05.2019	Removed playing with friends from process view section	Did not prioritise this feature for the game
05.05.2019	Improved section 2 with the help from the feedback.	The feedback gave us some pointers to fix, and fix we did!
06.05.2019	Added Android deployment diagram and description of how the kernel works as an interface between software and hardware in Android to physical view.	Feedback asked us to specify in more detail how software and hardware is connected.
09.05.2019	Changed game description to more closely resemble the implemented game	Some features (i.e Minimum highscore, obligatory collection of items, restart on death,..) where not implemented in the final product while other features (i.e Single player) where added.

Table 2: Changes during the project to the architectural design and documentation

References

Android Application Deployment. (n.d.). *Android Application Deployment Diagram Example.* Retrieved from `https://www.uml-diagrams.org/android-application-uml-deployment-diagram-example.html`

Bass, L., Clements, P., & Kazman, R. (2013). *Software architecture in practice.* Pearson Education, Inc.

Kruchten, P. B. (1995). The 4+ 1 view model of architecture. *IEEE software*, *12*(6), 42–50.