Architectural Description

Group 11 - Mandroid Studio

Team members:

Shivam Verma

Martin Halvor Korstveit

Kjetil Vaagen

Dag Erik Gjørvad

Kristian Huse

Thomas Skarshaug

Chosen COTS (Components and technical Constraints):

Android devices

Android Studio

Google Play Game Services

Primary quality attribute - Modifiability

Secondary quality attribute - Usability

Table content

1 Introduction

1.1 Description of the project

The educational goal of the project is to "Learn to design, evaluate, implement and test a software architecture through game development.". We are free to choose what kind of game we want to make, but there will be some requirements;

1. It should be a multiplayer game, focusing on the specified quality attributes.
2. The project must utilise architecture and design patterns in the game architecture.

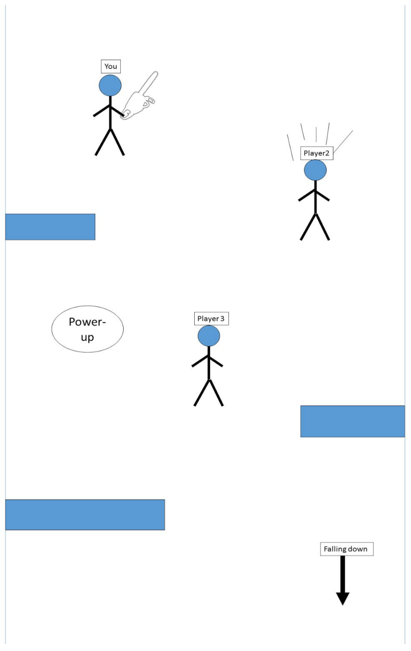
As well as these requirements we have to consider quality attributes about our product. All groups have to focus on modifiability in the design and implementation, but we also have to choose at least one secondary attribute from the following;

- Testability
- Availability
- Usability

In the first phase of the project we will create and document requirements and architecture of the game. This includes agreeing on the project requirements and our concept. All required fields and base layout is given.

1.2 Description of the game concept

We want to make a game based on one or more players moving in free-fall to avoid obstacles in the path with special focus on modifiability and usability. It should be a multiplayer platform, where you can challenge and play with friends. The players should be able to use power-ups and collide with each other to enhance the level of competition among the players. Below is a concept draft of the game.



1.3 Structure of the document

In this document we will go through our architecture, quality assurance and how we are supposed to implement it to our product.

- Introduction
- Architectural Drivers / Architectural Significant Requirements (ASRs)
- Stakeholders and Concerns
- Selection of Architectural Views (Viewpoint)
- Architectural Tactics
- Architectural and Design Patterns
- Views
- Architectural Rationale
- Issues
- Changes
- References

2. Architectural Drivers / ASRs

Our product is realized as a learning experience during a university course. As such we are learning as we develop. We are therefore modifying our architecture during the development.

2.1 Quality

- **Target platform**
  The application will only be published for use with Android devices. As such, Java and partly XML is the main focus and portability towards other platforms are disregarded.
- **Relatively small game**
  The concept and size of the game is of a relatively small size. This means it is important to be simplistic with as high usability as possible to lower the user barrier of the application.
- **Target market**
  The game will contain some violence, and as such is most suited for young teens (9+) and upwards. As such a high usability is required to ease all users into the application without needing an extensive amount of training.

2.2 Functional

- **Internet connection**
  As this will be a multiplayer game over a peer to peer connection, it is crucial that the game has connection to the internet. The quality of the peer to peer connection is then
  decided by the the internet connection of the device used and the the host device.

- **Multiplayer game**
  The multiplayer aspect of our game is crucial for the overall functionality in our app. All players are required to go through a game lobby before they start playing. In the game lobby you decide how many players you want to play against, whether it´s friends or randomly selected players, and load the game session. No one will be able to join the game after the game has started.

2.3 Business

- **Meeting personal objectives**
  All of the group members look forward to working with Android-development, and creating a product which can be enjoyed by others as well.
- **Meeting project objectives**
  The biggest concern is to meet all assignment requirements. Thus first and foremost we will focus on the essential functionality, but through good modifiability be able to add new and exciting content.
- **Responsibility to stakeholders**
  There are several deadlines to be met in regards to documentation and implementation.
- **Time constraints**
  We have focused on achieving a satisfactory documentation and find suitable design and architectural patterns. As such implementation of code and development will not begin before the evaluation of the first document, and corrections/additions has been made. Therefore the remaining time (4.5 weeks) must be spent diligently on making a product from the documented plans.
- **Lack of experience**
  Our lack of experience and practice in software architecture and app development will also probably affect the architecture.

3. Stakeholders and Concerns

As this is a learning experience we have considered stakeholders that are specific to our realization process. There are few projects that would consider the course staff as a stakeholder, but we realize it could be translated in another situation. In a company you would maybe have a supervising team that overviews the process, or a consulting group at the company you are working for.

Stakeholder	**Concern**
Developer	*Modifiability*: Should be easy to make changes and add new implementations to the source code, such as resources, levels and such.
Course Staff	*Reviewability*: The code should be implemented in such a way that it is easy to read and well commented. Architecture, requirements and similar aspects should be well-documented to make the process easier. The whole system should also be hassle-free to set up.
End user	*Usability*: Intuitive system and error tolerant. *Performance*: All users should have a similar performance, not giving some users a greater chance of winning.
ATAM-evaluators	*Reviewability:* Architectural documentation written in an easy-to-understand way for the evaluation.

4 Selection of Architectural Views

The architectural views illustrates the different stakeholder's viewpoints. We have chosen to base our views on the 4+1 architectural view model. There are four different views of significance: Logical view, process view, development view and physical view.

View	**Purpose**	**Stakeholder**	**Notation**
Logical view	Relates to the functionality that the system offers and how the various objects interact.	End-user Course Staff Developers	UML class-diagram
Process view	Relates to the dynamic aspects of the system like the the execution time behavior. It contributes to non-functional requirements and quality attributes such as: performance, scalability, concurrency etc.	End-user Course staff Developers	UML activity-diagram UML state-diagram UML sequence-diagra m
Development View	Addresses decomposition of sub-system and organizational issues. Maps elements of software components to actual file directories.	Developers Course staff	Package diagram Component diagram (UML)
Physical View	Describes the process of deployment, configuration and installation of the system.	Developers Course staff	UML Deployment diagram

5 Architectural Tactics

In this section we have considered our situation. As we have limited time and resources we ensure that all our tactics don't conflict with this.

Modifiability

Reduce the size of modules:
If the module being modified includes too much functionality, the modification costs will likely be high. Refining the module into several smaller modules reduces the average cost of future changes.

Well documented code:
By having a well documented code, we will increase the likelihood of not encountering errors when modifying the code. It is also important to define conventions to avoid different implementation. We have chosen these conventions:

- Camelcase naming of functions and variables
- Intuitive variable names(e.g healthBar, attackPower)
- Javadoc comments.
- Every function or larger logical unit of code should be explained with inline coding
- Every module should have a explanatory document listing input and output of functions

Increase cohesion:
Cohesion measures how strongly the responsibilities of a modules are related. We want a high cohesion of modules that affect the same responsibility. The reason for this is to prevent that different parts of the system are connected too strongly together, making it difficult to modify.

Reduce coupling:
There are several tactics that reduce the coupling between modules. We have decided to only mention a few we feel are important:

- *Encapsulation:* With this tactic we reduce the probability that a change to one module propagates to other modules by restricting the access.

- *Restrict dependencies:* With this tactic we restrict the modules that a given module interacts with or depends on. In practice this tactic is achieved by restricting a module´s visibility.

- *Refactoring of code:* Refactoring the modules gives a natural grouping to elements that have similar responsibilities, making them less dependent of the system

Usability

Prototyping and Usability tests:
One of the best ways to make a system with a good user interface is to facilitate usability testing. By rapidly producing prototypes and performing usability tests we will get continuous constructive feedback on our user interface and be able to make changes to improve the user experience along the way. With modifiability as our primary quality attribute it will allow us to do exactly this - make changes to the user interface with minimal cost.

Avoid using excess elements:
Instead of having a lot of elements on the screen for different controls we can use the same button(e.g tap two times) to do the same thing. This will make it easier for the user to get an overview of how the game works.

Maintain task model:
The task model is used to determine context so the system can have some idea of what the user is attempting and provide assistance. This will be useful in many scenarios in our game. For example with getting the correct input when a user creates an account.

6 Architectural and Design Patterns

Model-View-Controller:
Will be used by having a model class for the sprite and game states, a controller for the game logic and a view for the appearance. The view and model are only connected through the controller. This allows us to customize the game easier by for example changing the view when the game is played on a new device.

Factory method pattern:
Gives us the ability to easily obtain the object we need to work with by having sub factories specific to our use to create the desired object. This will for example be needed to generate different types of obstacles that share some common features. It would also be of great help when we dynamically need to create objects during the game runtime.

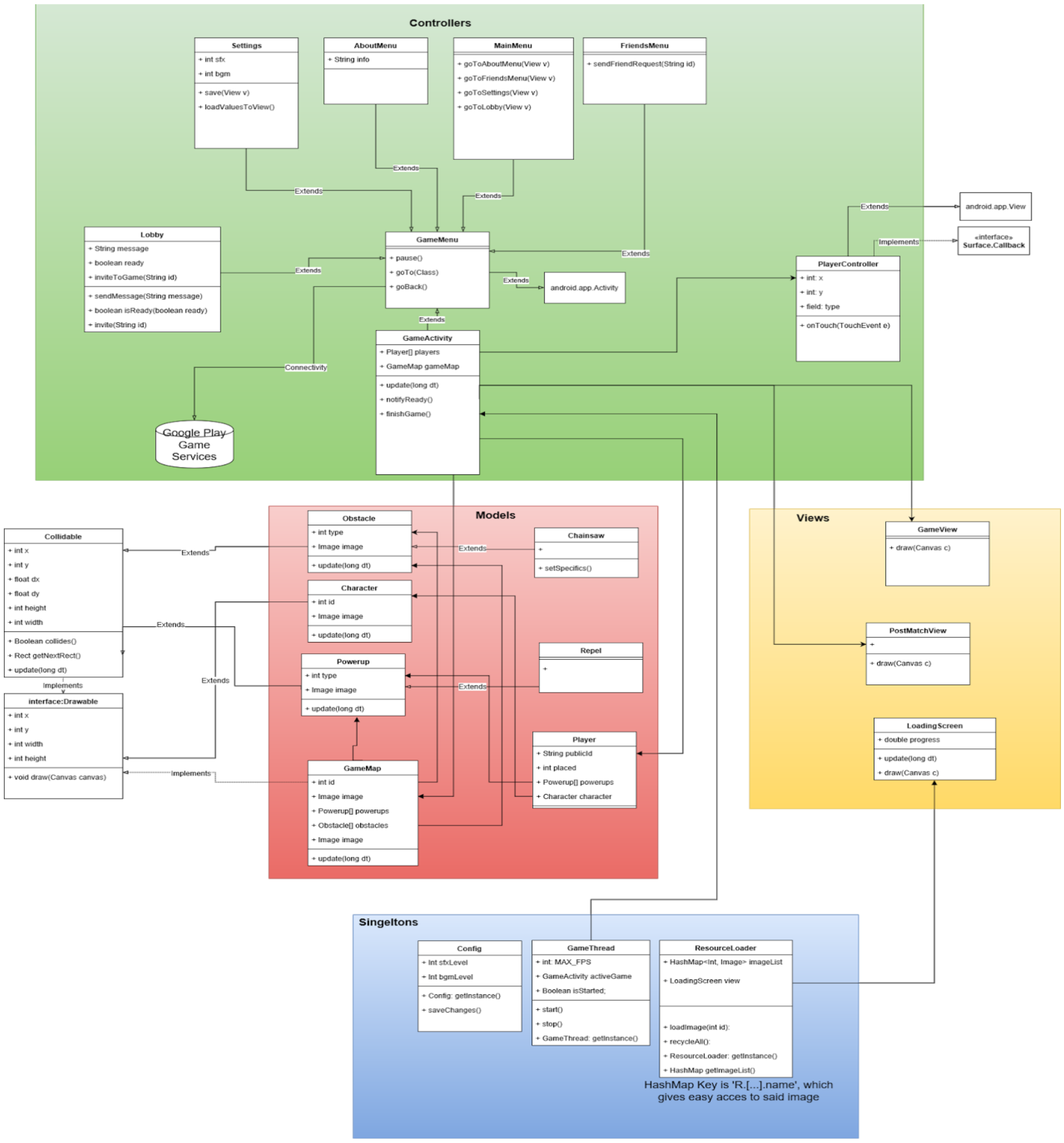
Singleton:
Often, we only need to use a single object at a time. By using the singleton pattern we make it impossible to make more than one instance of a class. This will be of need when we create objects like DataHandler so that we are certain that the state is consistent throughout the whole system.

Peer-to-peer pattern:
Our game will use a peer-to-peer pattern rather than a client-server pattern. With this pattern components will directly interact as peers where all peers are equal and no peer or group of peers can be critical for the system. The communication will go through internet connection and use typical request/reply interaction without the the asymmetry found in the client-server pattern.

7 Views

We chose to base our views on the 4+1 model using Logical view, process view, development view and physical view. This gives a good overview of the architecture from the perspective of different people that are concerned with it such as end-users, developer and course staff.
The notation decided on to represent the views was selected as they are the diagrams we are most familiar with, as well as best suited to visualize the internals of the system.
The views showcase the flow of the application, its components, distribution of packages and connections between devices.

7.1 Logical View



7.1.1 Explanation of the class diagram and its components

The class diagram is split into several parts according to the MVC architecture; Controllers, models and views. In addition there are 3 singletons, with data globally available across the platform.

The controllers are mostly used with regards to menu-navigation in XML. The superclass GameMenu consists of various methods and functionality that can be reused, such as “goBack” and “goTo”. In addition pause (onPause if Activity-default is used) will be available to said classes, halting various aspects of the application when the user changes to another application or exits.
PlayerController is the class which the user interacts with when actually playing. It consists of a glass-layer (android-view) on top, which was essential to split up controller and view in regards to the MVC-architecture. The packages imported from android.app is part of the COTS Android-api, as discussed in the requirement document.
Additionally, the Activity-class GameActivity is connected to most parts of the in-game application. It acts as a connector between models, controllers and views.

There are only 3 views in regards to the MVC architecture. LoadingScreen, which will get displayed each time resources are getting loaded into memory, GameView, which draws the various models and PostMatchView which displays the results after a game.

There are several superclass models, which in turn can be extended to create specific models, such as the Obstacle Chainsaw. By creating these classes, it will be easier for us to modify and add new content to the game.
The interface “Drawable”, is an interface that assures an object to be drawable by a view, depending on the x,y, width and height values.
Additionally, there is the class Collidable, that takes care of the movement of an object in addition to collision between objects of type Collidable. Most models used in the game extends this class, which decreases the amount of redundancy of coding, as well as increasing the modifiability of the end product.
The Player class is the object which contains values used in communication between devices via Google Play Game Services.

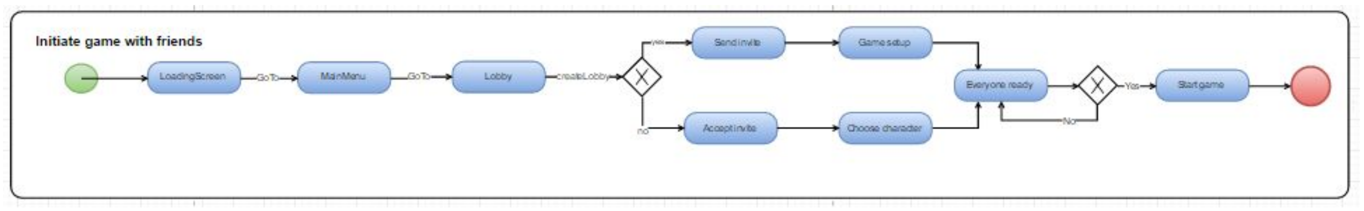
The most important factor to make a working game, is the GameThread, which assures the update and draw methods of the views and models are run. The thread will only run during matches, and as such requires a GameActivity to know what to update. From there, GameActivity runs the update methods of all its models and views, including communication with one of the COTS, Google Play, resulting in a functional game.

Factory pattern is realized through classes such as Obstacle, which can then be a chainsaw, or other similar extensions. Powerup will also be extended by the actual powerup-class, such as repel. It will also be possible to add sub-classes of character, to customize appearance and similar attributes.

P2P-pattern is realized by having the sending and receiving nodes being identical, as all communications is done by subclasses of GameMenu, which contains methods of communication with Google Play Game Services, and in turn, other peers.

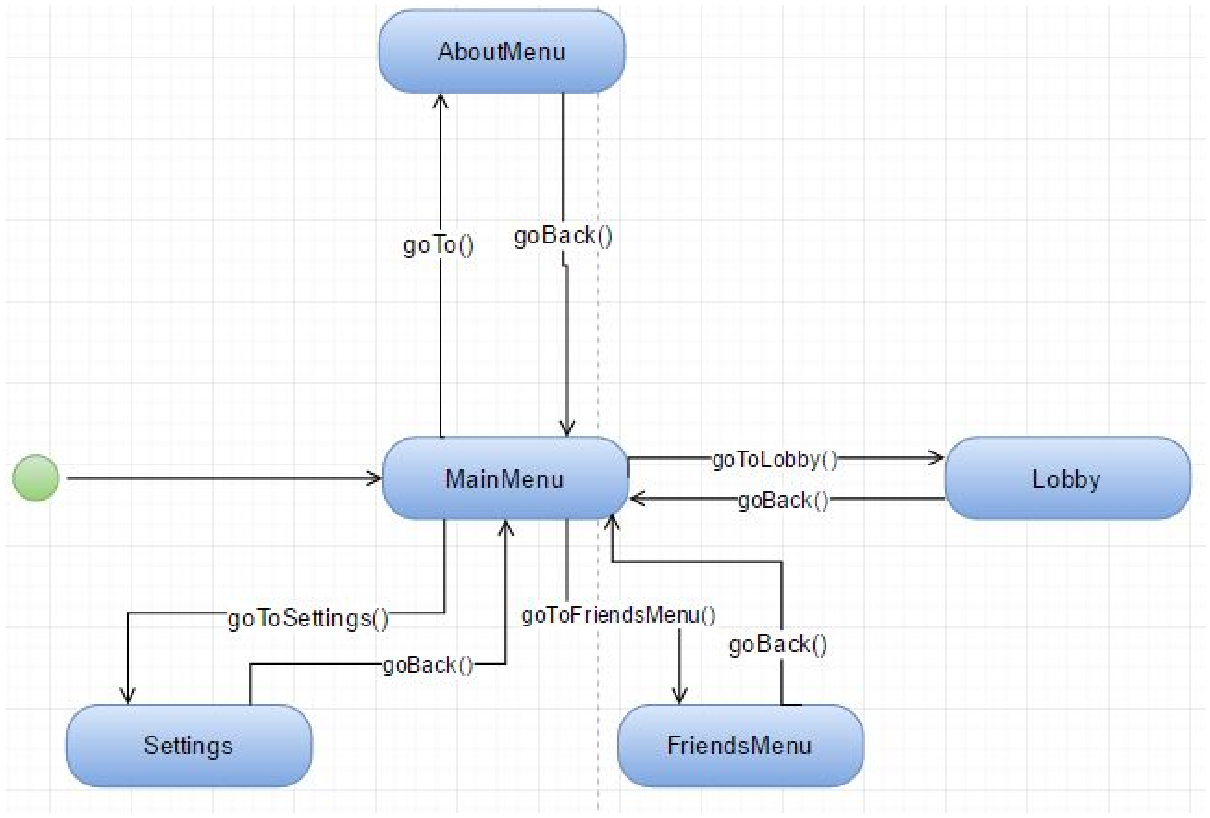
7.2 Process view

7.2.1 Initiate a game with friends



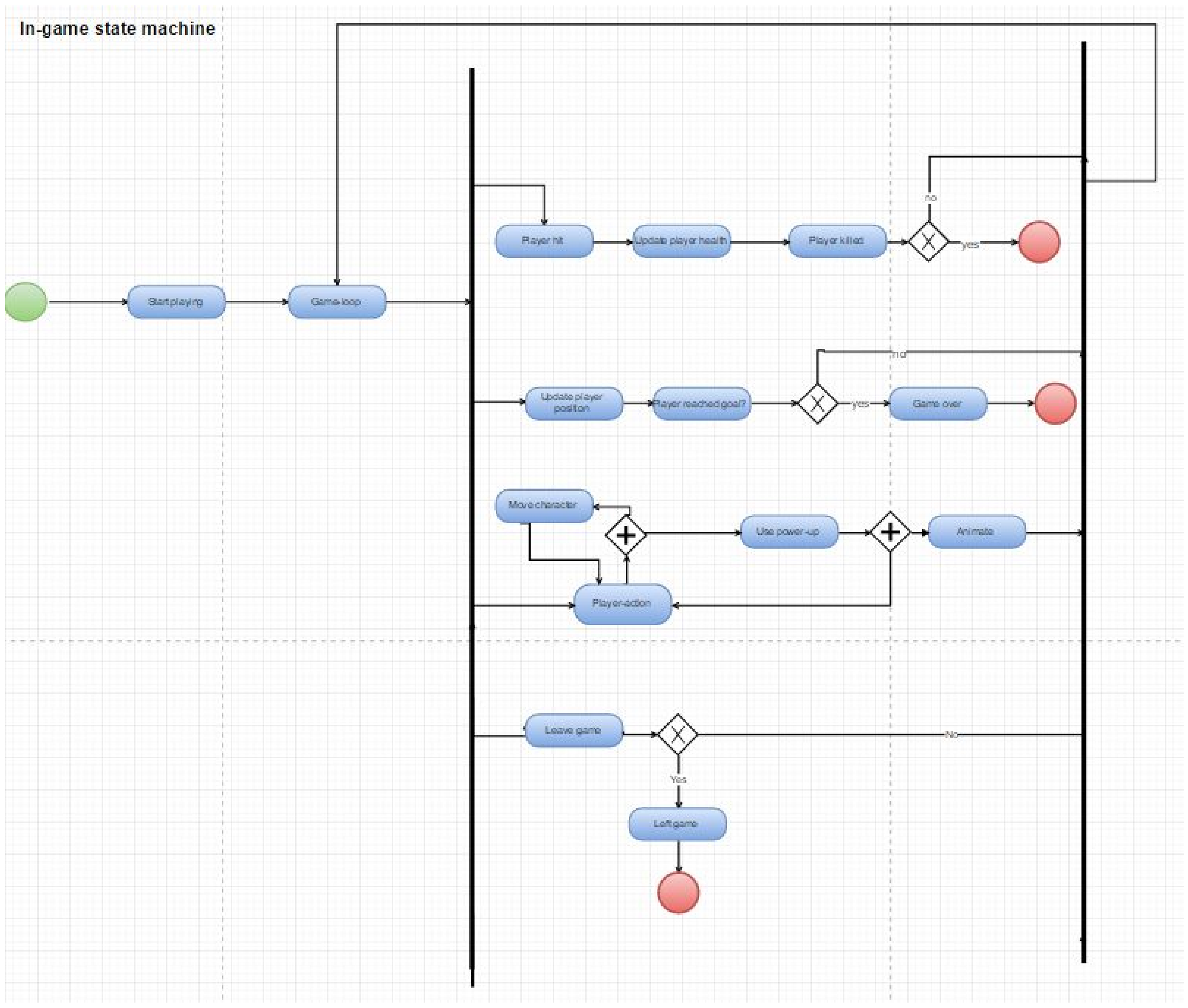
For this description of how to initiate a game with friends we assume the user just have started up the application. The first thing that happens is that the application shows a loading screen until the application is properly loaded. Then one automatically gets routed to the main menu. From there one has to go to the lobby. Inside the lobby one can either accept an invite from a friend, or invite friends to the game lobby. This is where the flow of the process splits into two parts. One part for accepting the invite, and one for inviting friends. This is done since we decided that the one inviting the other players has to set up the game (choosing level, allowed power-ups, choosing character etc.) The ones accepting the game invite can only choose a character and wait for the game to start.

7.2.2 Menu-selection



This diagram show how the user can move between the different menus. The way this is done will be quite standard, with a main menu which contains ways to enter every other menu, and a way to get back to the main menu from each sub-menu. The text on the arrows are the function calls required to get from the menu A to menu B in accordance with the direction of the arrow.

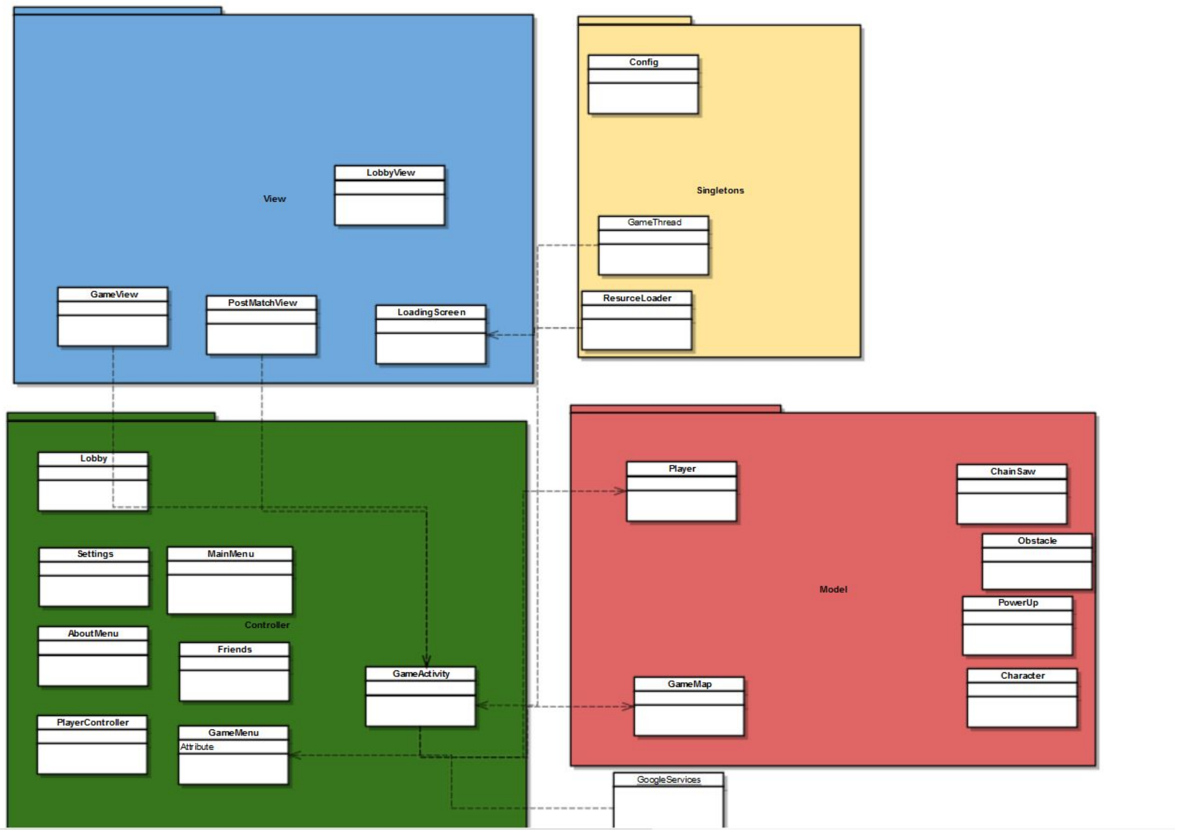
7.2.3 In game state machine



This diagram shows all possibilities for the user inside the a game. The bars represent the actions which will be done in concurrency with the game loop. This shows that it shall be possible to hit a player as well as move the player. It shall also be possible to leave the game whenever the user sees fit. The red dots in this case represent every state that results in that the player is not able to do any actions related to the game anymore (end state). This includes the player leaving the game, the game is over because one of the players has reached the goal or the player is killed and is incapacitated for the remaining duration of the game. All the states between bars leads back to the Game-loop state which represents that this state machine continues until one of the end states is reached.
The Move character block implicitly describes the collision since the character model extends the collidable class.

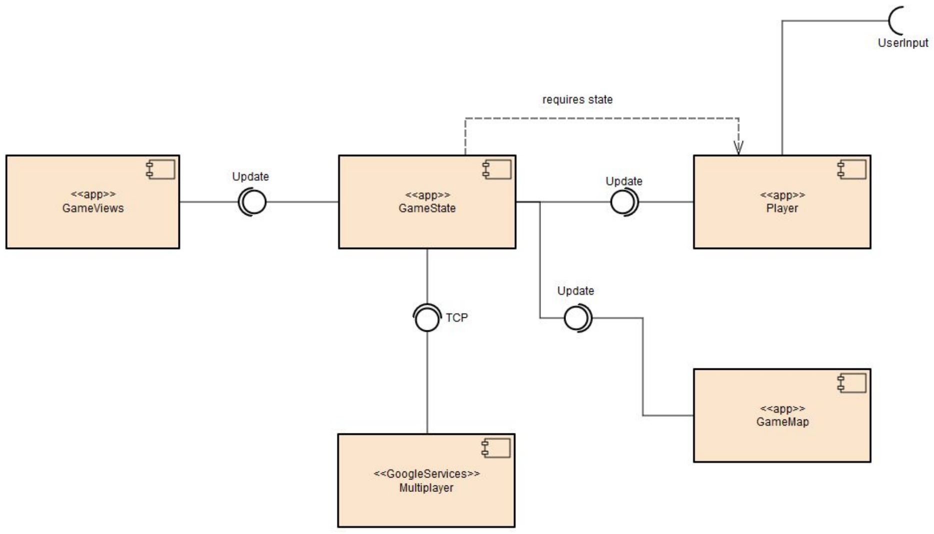
7.3 Development view

The package-diagram shows an overview of the packet structure for the project. The application is divided into four main packages, based on the MVC pattern. There is also a separate package for the singletons, this is make sure the singletons are easily identifiable and usable.

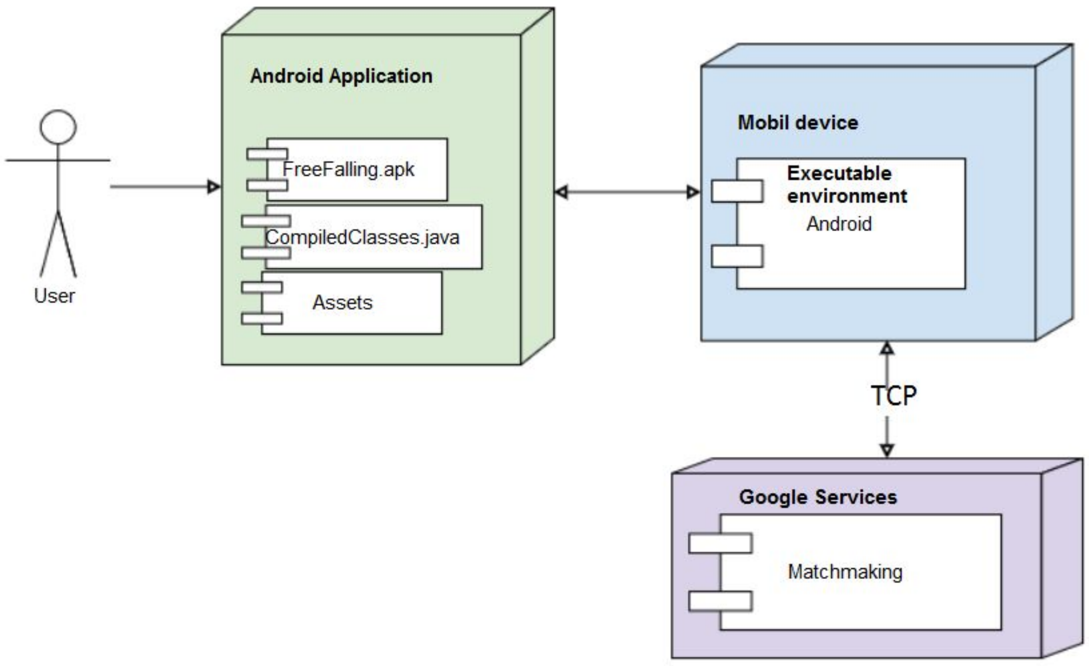


The interactions between the packets are clearly outlines and shows which classes within the packages will require information from outside their own package and which classes it is they are interacting with.

The gameActivity class is the fundamental class when it comes to interacting between the packages as it is the one updating both the view class and requesting updates from the models. It is also responsible for initializing the matchmaking and initializing and maintaining the peer-to-peer connection. This view is simpler compared to the logical view in an effort to highlight the dependencies between the Views, models and controllers.

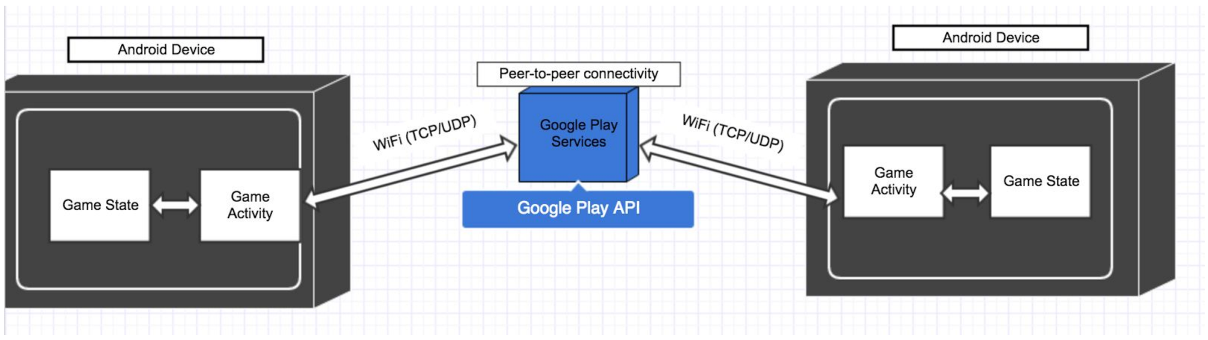


The component diagram shows the interaction between the components of the application, giving a clear overview for which parts should that can be developed independently. The gameState is updated with information provided by the Player and the gameMap. Updating is delegated to the gamestate, which gets the user input from the players and the updated map from the gamemap. Starting multiplayer is done by the multiplayer component, while maintaining this is delegated to the gameState after startup.



The deployment diagram shows how finished assets will be deployed and the distribution among the physical assets. The main game apk and it's assets will be deployed as an android application, that will run on android. It is depended on Google Services to initiate and maintain the matchmaking, while the multiplayer itself will be running on peer-to-peer basis.

7.4 Physical



The devices are connected from peer to peer through google play services which will allow easy matchmaking and create friend lists to play against each other repeatedly. We will utilize Wi-Fi to communicate through google´s API and to each end-user.

7.5 View inconsistencies

The use of collision detection was omitted in the process view but is specified in the logic view. This is so because the Move character block in the In-game state view under process view implicitly describes the collision since the character model extends the collidable class, and hence is not explicitly specified in the process view.

8 Architectural Rationale

By choosing this architecture we think and hope that we have a strong fundament to obtain the declared qualities. To make the game more modifiable we chosen use an MVC pattern. This makes the game more modular and defined so that a module easily can be changed without changing the whole system. We also chose the Factory pattern to make it easier to extend the game. In addition we have listed several tactics that helps us obtain better modifiability. All of these things have been taken under consideration when choosing the COTS. We chose Android studio because it naturally uses the MVC pattern so that it is easier to adapt to changes in devices screen size.

The choice of Google Play's friends service for the network-multiplayer matching increase usability because it is an well established service that many are familiar with. In addition it saves the user the hassle of having to register a new account.

By having a modular architecture we also get the benefit of easier work delegation amongst the project group. This will lead to increased effectivity which is needed to fulfill our defined requirements in time.

Obtaining good usability is a little more difficult since the field is not that mature and depends on a lot of variable factors like subjective preference, other elements used and physical condition. However we have tried to come up with relevant tactics to make the usability as high as possible. To do this we have tried to put ourselves in the user's position and defined architecture that we prefer in already existing, similar applications.

9 Issues

We had no issues that is suitable for this section.

10 Changes

Date	Change	Reason for change
16.03.17	Removed Quality Assurance from 2.1	Content was not correct.
16.03.17	Added multiplayer game under 2.2 Functional drivers	Very important for our architecture
16.03.17	Target market is moved from 2.2 to 2.1	Target market belongs under 2.1 Quality.
16.03.17	Removed availability from stakeholders and concerns (end-user)	Remnants of a server from early versions of the project.
16.03.17	Changed the name of the tactic “Facilitate experimentation” to “Prototyping and Usability” tests	More suitable header.
16.03.17	Removed introduction of section 2	Unsatisfactory introduction, mentioning a server not in the project anymore.
16.03.17	Added introductions to sections 2-5, and 7	Introductions were needed

16.03.17	Added realization of patterns to 7.1	Further specified how patterns were to be used through logical view.
16.03.17	Updated titles under usability	Fits better with the contents of the paragraph
16.03.17	Added time constraints and lack of experience to business drivers.	Was a valid aspect of business drivers to be added.
16.03.17	Removed the Layered pattern as an architectural pattern.	We felt that the complexity by implementing such a pattern would cost us more than it would benefit us.
18.04.17	Changed the Stable connection header under 2.2 to Internet connection and changed its content to something suitable.	Old reference to the server-client architectural pattern which now is peer to peer.

11 References

- Len Bass, Paul Clements, Rick Kazman, “Software Architecture in Practice – Third edition”, Addison Wesley, September 2012
- eToturials, “Usability Tactics”; web: http://etutorials.org/Programming/Software+architecture+in+practice,+second+edition/Part+Two+Creating+an+Architecture/Chapter+5.+Achieving+Qualities/5.7+Usability+Tactics/ Accessed 26.02.2017
- Thomas Owen, ”Mapping between 4+1 architectural view model & UML”; web: http://softwareengineering.stackexchange.com/questions/233257/mapping-between-41-architectural-view-model-uml Accesed 26.02.2017