Castle Crush

Architectural description

Nina Marie Wahl
Erik Kjernlie
Ludvig Lilleby Johansson
Truls Elgaaen
Jørgen Mortensen

COTS: **Android SDK**, **Google Play Services SDK** and **LibGDX**

Primary quality attribute: **Modifiability**
Secondary quality attribute: **Usability, Availability**

26th of February

1. Introduction

The goal of the project is to learn to design, evaluate, implement and test a software architecture through game development. We are supposed to develop a simple multiplayer smartphone game. We are supposed to focus on some specific quality attributes, and we have chosen modifiability (which all groups are supposed to), usability and availability.

In the architectural phase we decide what kind of patterns and tactics we want to use in the game development, based on the functional requirements and quality attributes we decided on in the requirement phase. We will also design the structure of the game and the code.

This is the architectural description of our game "Castle Crush". It includes our choices of relevant functional requirements, quality attributes, architectural and design patterns, tactics and architectural views. The document is based on recommendation from IEEE 1471, and will be updated when changes are made throughout the whole project.

The game is a turn-based multiplayer game. Although it is turn-based, the game will be running as real-time in online multiplayer mode. You can choose between several game modes, either offline against the computer or online versus friends. Each player has a castle of boxes protecting a game winning object, along with a shooting cannon.

When it is the current players turn, a moving pointer indicates the angle of the shot. The player touches the screen to choose an angle. The player touches the moving pointer again to decide the power of the shot. The moving pointer makes the shooting less precise, preventing repeating shots. The speed of the angle and the power can be modified to change the difficulty of the game.

If the bullet hits one of the boxes of the opponent's castle, the box get cracks (needs several hits to vanish) or disappears right away. All the boxes have gravity, which means when a box is hit, all the boxes on top of it will fall rectilinear down. The goal of the game is to be the first to hit the opponent's object (in the illustration below, the object is a heart).

Preliminary illustration of how the game may look after implementation is shown on the next page.

Figure 1: Player 1 shoots at player 2.

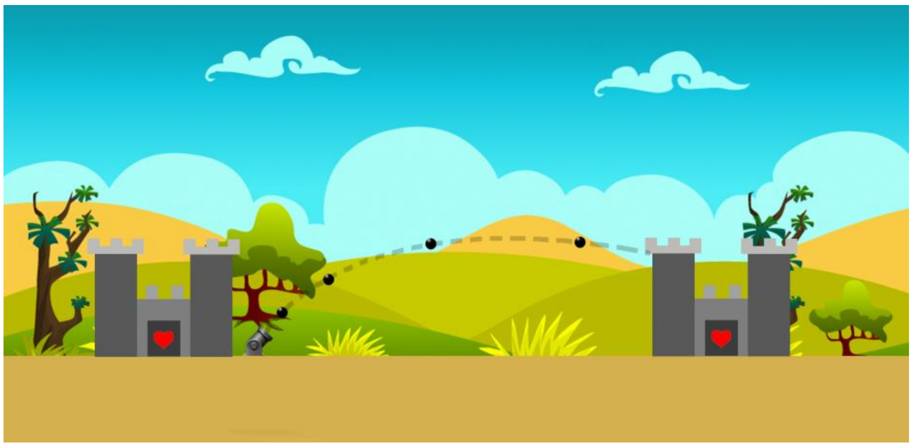


Figure 2: Player 2’s castle is hit, and the boxes cracks.

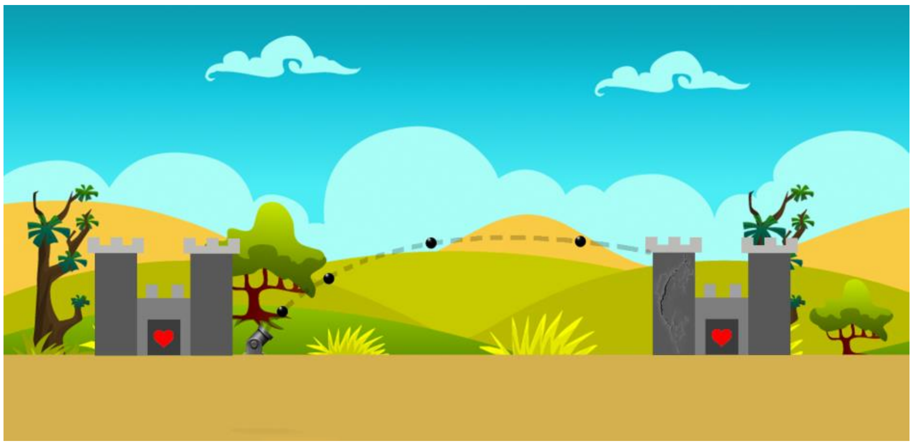
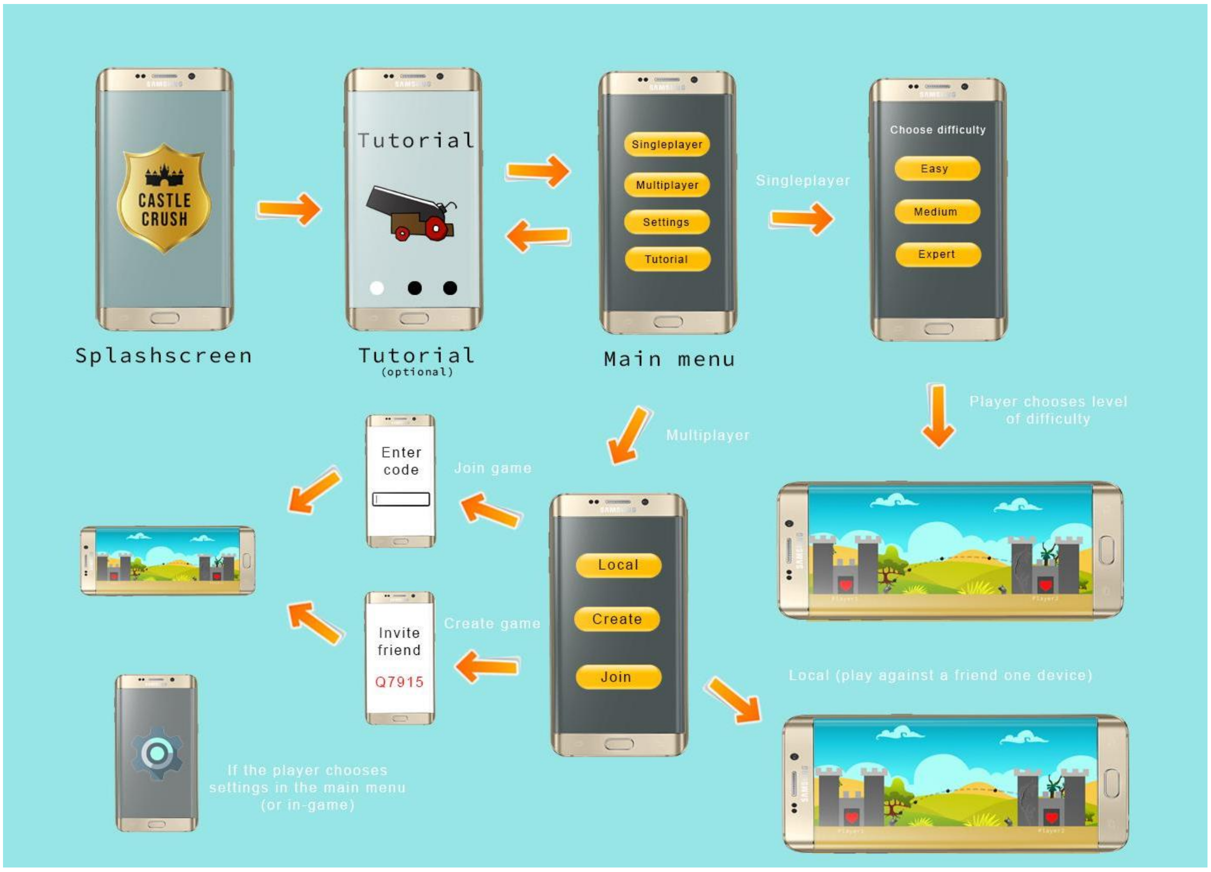


Figure 3: Template of how the application are supposed to work/look.



2. Architectural drivers / architecturally significant requirements

2.1 Functional requirements

Multiple game modes

The game should be playable both online and offline, which will affect the architecture. The game will have to be able to switch between receiving input from both the internet and from the device it is running on, and receiving from only the device.

Online multiplayer

Users should be able to play against each other over a peer-to-peer connection, so an internet connection is crucial to maintain a game with minimum delay. The game must have some functionality for sending and receiving information over the internet. We are going to use Google Play Services for handling real-time peer-to-peer network games, where we will set up a multiplayer game room where players can invite other players with a unique game ID.

COTS and devices

The game will run on mobile devices, which puts constraints on input-type as touch screen is the only sensible input to use in our game (we don't want to use microphone or accelerator). In addition, the game will only be made for devices running Android, which means we will have to implement our game in java or other languages that can be used to develop android apps. The size of the screen also puts limits on the number of rendered objects and size of the playable area as we plan to have a static game view that shows the entire game world. The amount of processing power and memory is also constrained by the device it is running on, but this should not be a problem as our game will not need large amounts of RAM or GPU as the rendering is relatively simple and the number of assets used is small.

Since we are using libGDX we are constrained by this framework, and if we want to implement different tactics and patterns than the framework implements by default it will make the development process much more time-consuming as we have to override libGDXs default systems.

2.2 Quality attribute

Modifiability

Change happens, which is why modifiability is our primary quality attribute. The game must be easy to modify, without making major changes to other parts of the system. This means that we have to take this into consideration from the beginning. We need to design the system with separate modules that communicate with each other, and when one module is changed, it does not affect the others.

Usability

This is one of our secondary quality attributes. It is important that the game is intuitive and easy to use, so inexperienced users can fully understand the game within minutes. If necessary, the user can choose to do a quick tutorial explaining the basic concepts and the goal of the game. This is selected as an ASR, because the menus, buttons and navigation must be as simple as possible, and this will influence the architecture.

Availability

Availability is also a secondary quality attribute. A big part of our game is to be able to play online with friends or other players. To make sure this functionality is always up and running, it is important to have good availability. The connection between the players should be reliable and it is necessary to take precautions to be able to both avoid and recover from faults. Because we use an API created and hosted by Google Play Services, we are partially dependent on the availability of this service. It is still important that we keep our application up-to-date with their systems and use tactics on our own to make sure the game is reliable.

High performance is often associated with an online game, but since we are making more of a turn-based game instead of a real-time game with strict requirements to delays, we are primarily focused on the three other quality attributes.

2.3 Business constraints

Time constraint

We are working against a clear delivery date with a time frame of only a few weeks. Because of this short time limit, this is an ASR because we won’t have time to do major changes to the game or the architecture if it is needed in the later stages in the development. There are also several deadlines of documentation, evaluation and implementations that have to be met. We will focus on making a functional product with a well-documented architecture.

Resources

Our team has five members that have limited time available to work on the project, which places constraints on what we can achieve. The team members also have limited experience working with architecture and patterns.

3. Stakeholders and Concerns

3.1 Professor and teaching staff

Lecturer and teaching staff is concerned with the learning of students and is invested in the project. Therefore, we aim to make decisions that are deliberated regarding patterns and architecture. We will familiarize ourselves with the curriculum before making choices. To make it easy for the staff to get insight to our work we will need to document the project well, using architectural views and other relevant tools to explain our thoughts and implementation.

3.2 The team

The team is also concerned with learning, not just about architectural patterns, design and their implementation, but also about the software development process. The team also want to deliver a well-designed and documented system, which results in constraints for the project.

3.3 End-users

Primarily, the end users are the ones who are going to play the game. To gain the best possible experience, their concern is to have a simple and intuitive design, along with good program usability.

3.4 ATAM evaluators

The concerns of the ATAM evaluators are that the documentation is well structured, so they easily can evaluate it, as well as that we deliver on time, or else the evaluators will also be delayed.

4. Selection of Architectural Views

We have chosen to follow the 4+1 view architectural model, which specifies five different viewpoints. They are logical, process, development, physical and scenario, but we chose to focus mainly on the first four, as we felt scenario wasn't that relevant for our game. Our choice of architectural views can be seen in table 1.

Table 1: Architectural views

View	**Purpose**	**Stakeholder**	**Notation**
Logical View	Specify the functional requirements of the game. The class diagram will give a picture of all the classes/interfaces in the code.	Developers ATAM evaluation group Course staff End users	UML class diagram
Process View	Shows the different components of the game and how the communicate/interacts in runtime.	Developers ATAM evaluation group Course staff End users	State diagram and sequence diagram
Development View	Describes the	Developers	UML diagram

	organization of the system components.	ATAM evaluation group Course staff	
Physical View	The physical view is a technical description of the implementation of the logical view. It shows the topology of the components, as well as their physical connections.	Developers ATAM evaluation group Course staff	UML deployment diagram

5. Architectural tactics

We have decided to prioritize the Modifiability, Usability and Availability qualities. **Tactics** are a collection of design techniques that an architect can use to achieve a specific quality requirement. We have chosen a selection of tactics belonging to each quality attribute. We will focus on these tactics in the architecture and implementation.

5.1 Modifiability

About how difficult it is to introduce desired changes.

Reduce Size of a Module:

Split module: Large modules with a lot of complexity lead to higher costs. Refining them into smaller modules will make it cheaper to make changes later.

Increase Cohesion:

Increase semantic cohesion: Refine modules so that each module only contains functionality associated with one specific area. When a module is exposed for a change, all responsibilities in the module should be affected. If they are not, some of the responsibility should be moved into a new (or already existing) module.

Reduce coupling:

Refactor:
“Clean up-step” made on two modules that are (partial) duplicates of each other. Common functionality is moved outside, so we get a main module with multiple submodules. This way, we ensure that common functionality is not repeated - we reduce coupling. You should not need to make the same changes to more than one module.

Encapsulate: Encapsulation make sure that when making changes to a module, these changes do not propagate to other modules. This happens by introducing an explicit interface (including an API) for the module. The modules are only allowed to publish what may be used of other modules.

Restrict dependencies:
Restrict which modules that may interact or depend on each other. May be done by controlling the visibility and authorization.

Use an intermediary:
By having the modules communicate with an intermediary and not directly, the type of messages a module send does not matter. Therefore, the developer can change a module without having to change the modules it communicates with, only the intermediary.

5.2 Usability

How user friendly the system is.

Support user initiative:

Because our game is a real time multiplayer game, undo and pause will not be relevant user initiative. **Cancel**, on the other hand, is very necessary.
If a player wishes to cancel/terminate the game along the way, this has to be an option and the online opponent have to be notified.

Support system initiative:

Maintain system model: Maintaining an explicit model of the system, that is used to determine expected system behaviour to give the user an appro-
private feedback.

5.3 Availability

About how reliable the system is.

Detect faults:

Ping/Echo: By using ping/echo each time a player gives the system input, we can check if the opponent is still connected.

Recover from faults:

State resynchronisation, Passive Redundancy: The two devices that is connected and play against each other each have a version of the current state of the game. Thus, they work as each other's backup. If a fault is detected, and the devices is out of sync, the system can check the last state that was accepted by both devices and act accordingly to re-sync the devices.

6. Architectural & Design Patterns

6.1 Model-View-Controller (MVC)

Model-view-controller is used to separate the application’s logic from the rest of the user interface by dividing the application into three interconnected components. MVC is a commonly used architectural pattern, mainly because the model part is separated, and can be tested independently of the user interface. By separating these concerns, MVC increases testability and modifiability of the application. In our implementation, we plan to use aspects of the MVC-pattern to construct the larger structure of our system. We will have classes that provide rendering separated from the classes that contains game logic and try to have a separate section that accepts user input. The logic and information on location and state of the game elements will be stored together in what will work as our model.

6.2 Singleton pattern

The singleton pattern will be used for objects which only are supposed to be instantiated once. These restrictions are made by making the constructor private, so the object can only be created inside the class.
We should only have one bullet object, and it should not be possible to make more of them. Both players are supposed to use the same bullet, and we do not want a player to be able to shoot, before the other player is finished with his turn. We will solve this by use of the Singleton.

6.3 Template method pattern

Template method pattern is a design pattern, which works well with the modifiability attribute. The invariant parts of the behaviour are implemented only once in an abstract class, and the subclasses can implement the varying behaviour.

A lot of the objects in the game should be able to be “upgraded”. It could be to choose a bullet which causes more damage or boxes with different strength. The template method pattern makes sure all the different versions of the object have the same basic behaviour.

6.4 Observer pattern

When a box is hit and disappears, the other boxes might need to update their position. We could run an update method on all the boxes all the time, but to save resources we would instead use the observer pattern and notify the other boxes when a box disappears. This means, the boxes will be both observers and subjects.

LibGDX also use the observer pattern to deal with event-handling.

6.5 Peer-to-peer

To make the game a local multiplayer game (players can invite other nearby players to join) we will use Google Play Services networking API called “Nearby Connection”. This is a peer-to-peer API which exchange data between different participants in the game.

7. Views: Logical, Process, Development

In this section, we are going to present the different architectural models of Castle Crush.

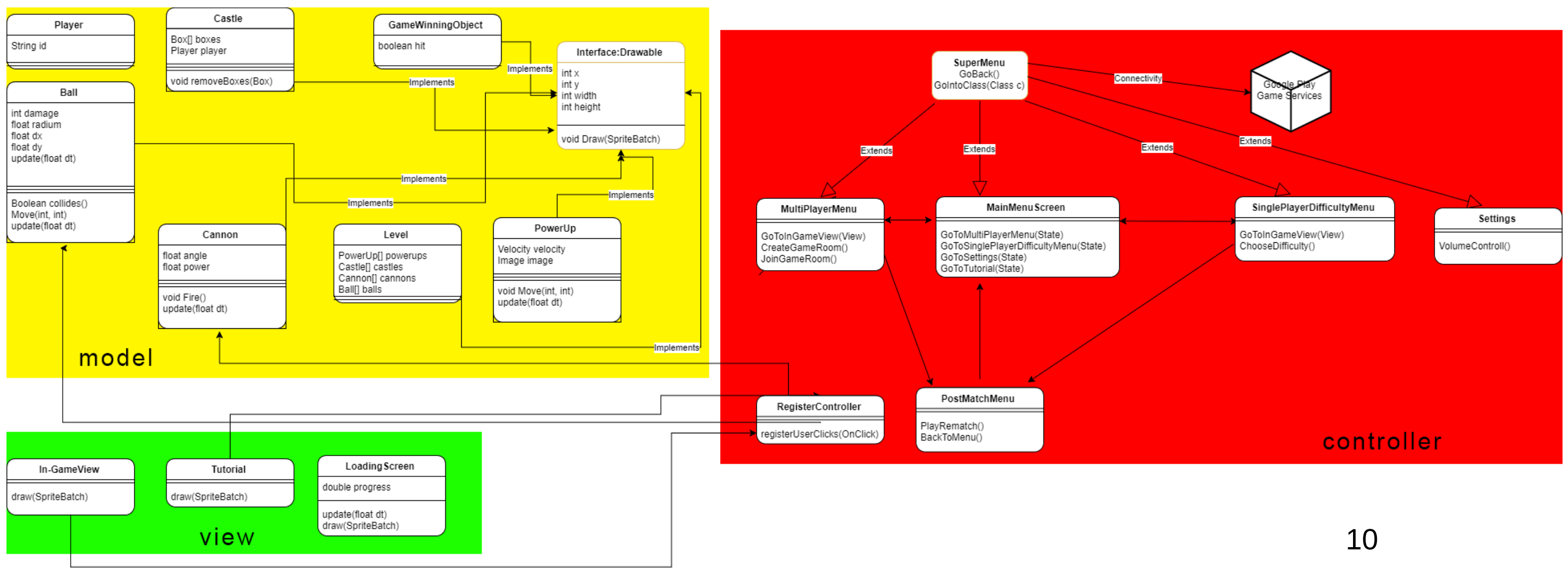
7.1 Logical

The main application and logic of our game is shown in the class diagram in Figure 4. We chose our preliminary architecture with respect to the Model-View-Controller architecture in order to support our primary quality attribute modifiability. In our controller section, the superclass SuperMenu provides functionality for the other menu-classes through inheritance, which will improve the efficiency when doing changes to the code. Our controller section mainly contains different classes related to navigating in the menus along with a Settings-class. To handle the user input, such as user clicks and text input, RegisterController connects the view and model components so they can interact with each other.

The models are all of the physical objects and their logic that are going to be used in our game. Because the most of our models have the same functionality and behaviour, they implement an interface that has functionality of drawing the objects to the screen.

We only have three views because the controllers are going to contain most of the functionality in our game. Our tutorial-view will consist of a slideshow with no logic or functionality other than just swiping to left or right. The main view of our game is In-gameView that are going to draw the different models on the screen and show the gameplay.

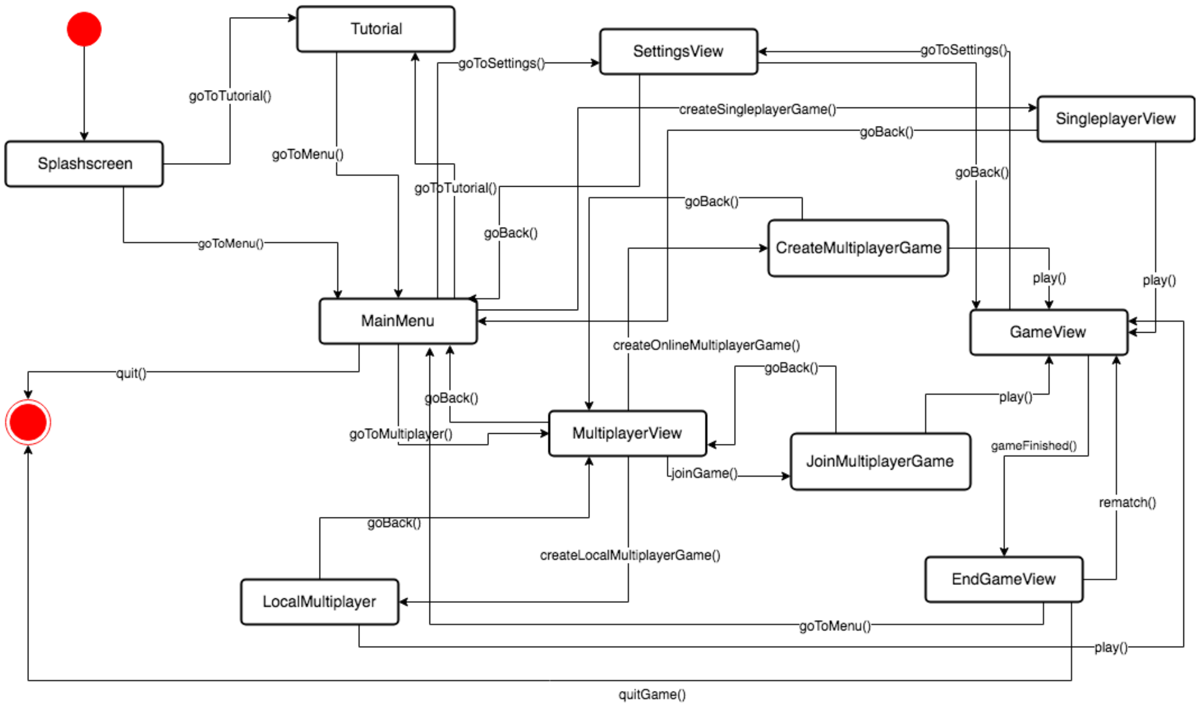
Figure 4: Class diagram of the logical view



7.2 Process

For the process view, we have chosen to create a state diagram. We are assuming the user begins in the initial state, which is immediately after the user opens the app. A splash screen will appear for a few seconds. The first time the user opens the app after it is downloaded, a tutorial of the game will appear. This is just a slideshow of pictures where the user can swipe through to get familiar with the basics of the game. The tutorial is also accessible via the main menu. After the tutorial is done, the user will be navigated to the main menu, where it is possible to choose between single player or multiplayer. Both in the main menu and in-game, the user can always press a small settings-icon to set the volume (and possibly other features) to a desired level. If the user presses single player in the main menu, the user can choose to play against an easy, medium or expert bot. If the user presses multiplayer in the main menu, the next thing to do is to choose between local or online multiplayer. Local multiplayer is playing on one android device (without internet connection) against an opponent. Since playing against a random opponent is quite the same as playing against a bot, the online multiplayer works by playing against a friend with a code. The user can therefore create a code or join a game via a code. After a match is done, it is possible to play a rematch or quit the app.

Figure 5: State diagram



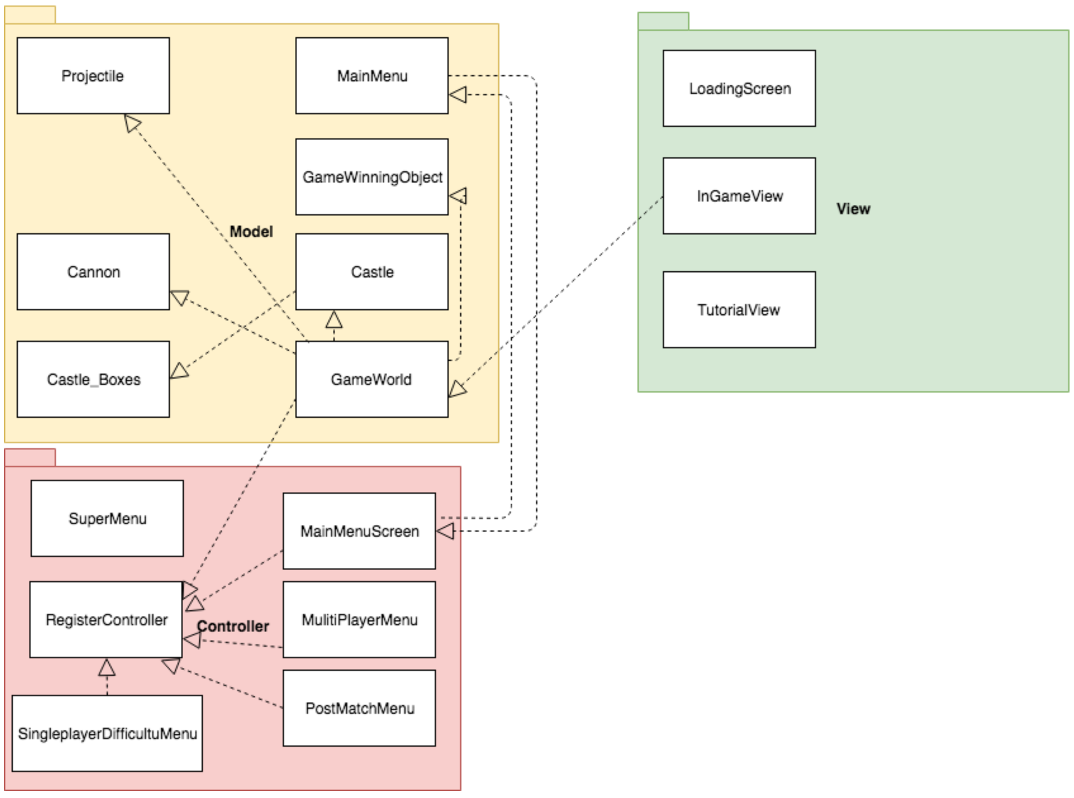
7.3 Development

The development view, as can be seen in figure 6, shows the connection between the three packages we plan to divide the application into. The packages are based on the MVC-pattern. The dependencies between components is illustrated by dashed arrows showing which components depends on which. The RegisterController will handle input from the user, therefore many other components will depend on information from this component. The GameWorld component will have responsibility for the game world and the objects within and will need to get information from the components responsible for the objects in the game. The communication protocols and interfaces between the different packages and components are not yet defined, as we do not yet have enough information on the most effective way of implementing this.

The diagram shows which components can be implemented without much regard to other components, and which needs to consider the workings of other parts of the software. For example, the TutorialView-component can be developed without much thought to other components, but the GameWorld component needs to take into account how the Cannon, Projectile and Castle components are implemented.

Different packages will communicate with each other through interfaces, which means they can be implemented independently.

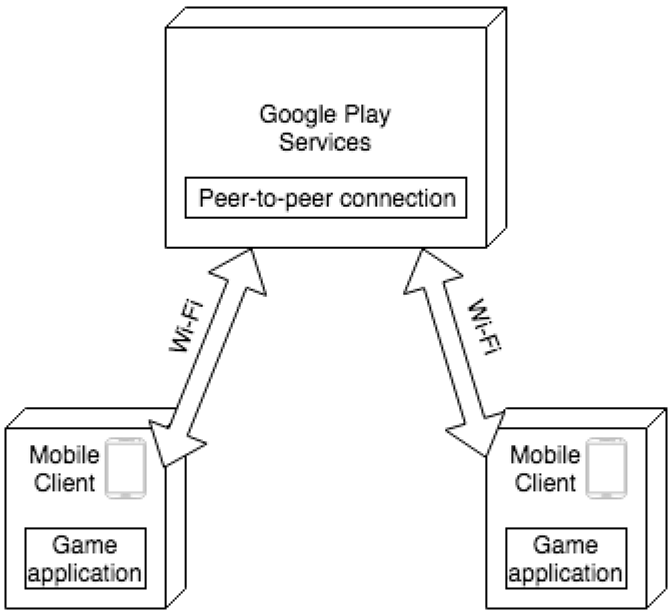
Figure 6: UML diagram of the development view



7.4 Physical

Devices are connected peer-to-peer through Google Play Services and communicates over Wi-Fi through Google's API and to clients.

Figure 7: UML deployment diagram of the physical view



8. Inconsistency among views

There are some inconsistencies between our views. This is mainly a problem for function- and filenames because of asynchronous updates in diagrams without having an explicit code to refer to (no code skeleton). This has also led to some lack of details in our views.

In the process view, Google Play Services was omitted as a connection point for online multiplayer as it isn't relevant to the internal process of the game flow.

In the logical and development view, we have added a view called LoadingScreen, while we have SplashScreen in the process view. This is because we think that the SplashScreen is independent of the other logic and doesn't really make a difference in our logic and development view, while it is of some importance to the end user in the process view.

9. Architectural rationale

We will use the **Model-View-Controller** pattern as an overarching architecture to facilitate that the larger parts of the system are modifiable. It also makes sense to separate the models handling user input and the game logic as our game will be user input-driven and will be dictated by the input given to the Controller.

To make the modules handling game logic modifiable we will use the **Template pattern**, so that the modules will have to follow the same structure.

The **Observer Pattern** will make it easier to implement dependencies between the in-game boxes and will assist in increasing the performance of the game since it will reduce workload required for each frame update.

The **Singleton pattern** will help with encapsulation, making some components very hard to access without following the proper protocol which makes the system more modifiable.

The choice of using Android Play Services real-time multiplayer API forces us to use its innate **peer-to-peer** pattern, but this makes it easier for us to implement the system as we don't have to communicate with a server, but rather a virtual game room, so this is not a problem.

10. Issues

We had some trouble choosing the architectural patterns for the game, and this took a lot of time. We did unfortunately not find time to make a code skeleton in this phase. Therefore, we are somewhat uncertain about what we intend to do will work.

11. Changes

Here we will log changes we do after the first delivery.

12. References

Len Bass, Paul Clements, Rick Kazman, “Software Architecture in Practice–Third edition”, Addison Wesley, September 2012

Phillipe B. Kruchten, “The 4+1 View Model Of architecture”, IEEE Software Magazine, Volume 12, Issue 6, 1995.

Ian Gorton, “Essential Software Architecture - Second Edition”, Springer-Verlag, 2011