TDT4240 Software Architecture

Othello 2.0

Group 28

Group members:

Henrik M. Backer
Quentin Depoortere
Robin Alexander Finstad
Hanna Eide Solstad
Espen Sørhaug
Romain Vo

Primary quality attribute: modifiability

Secondary quality attribute: usability

Chosen COTS:

Android devices
Android Studio
LibGDX
Google Play Services
Github

TDT4240 Software Architecture

1 Introduction

1.1 Description of the project and the architecture phase

This project is a part of the subject TDT4240 software architecture. The goal of the project is to develop a multiplayer game with focus on the architecture of the design and implementation of the game. Our initial thoughts were to make our multiplayer game competitive, so we decided to make a turn-based board game and chose Othello.

In the architecture phase the guidelines for the implementation, and all further development, testing and integration will be decided. As choosing the architecture affects the structure of a project and can be challenging to change at a later time, it is worth spending some time actuating oneself with different architectures. To be able to start programming within the limitations set in this phase, it is important to first assess the architectural drivers at hand. To do this, the team must ask itself: “What sorts of concerns do we have to make based on the characteristics of the finished application and the frameworks in which we will be developing”. Second is assessing the concerns that must be made regarding different stakeholders. Furthermore, one must decide upon architectural views, architectural and design patterns and discuss the architectural tactics that aid in achieving this required architecture.

1.2 Description of the game concept

The game is an implementation of the classic game Othello and uses the same set of rules as the original. The game starts out with four pre-placed discs with the starting position that can be seen in figure 1.1. In the game each player is assigned their respective color and the players alternate taking turns. When your turn starts, you must place one of your game pieces

on one of the available squares. The goal of the game is to have more pieces on the board than your opponent. This constitutes a win.

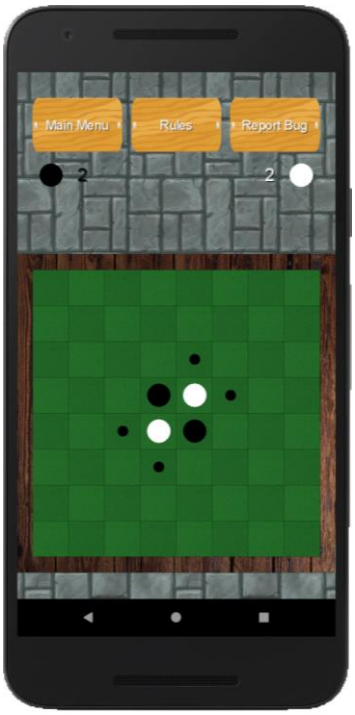

Fig 1.1 - Starting position for every game of Othello.

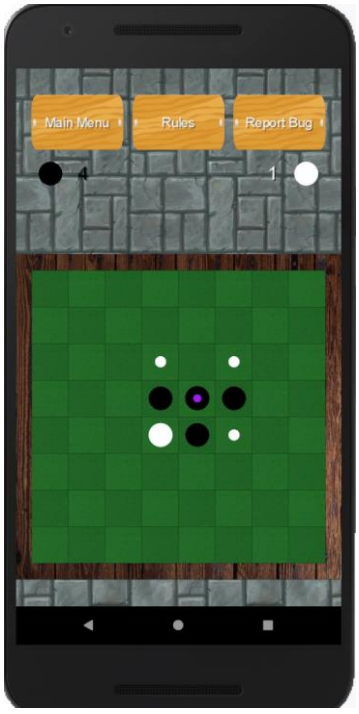

Fig 1.2 - A possible position after black has made one move.

The player with the black discs starts the game by placing a disc such that it surrounds a white disc on a line, either horizontally, vertically or diagonally. The white discs that are enclosed by the previously- and now newly placed black disc is flipped and changes color to black. In the starting position there are four possible moves for black and they are indicated by the small black circles beside the white discs, as seen in figure 1.1. If the black player chooses to place

their disc in either of the two upper-right positions, the upper-right white disc is then flipped and turns black. One of these two outcomes can be seen in figure 1.2.

The game in its entirety is played by either player choosing one of the available moves presented to them by small circles in their respective color. After black has made their first move the game continues with each player alternating turns placing discs, it's also possible to flip several discs at once if the requirements stated above are fulfilled. When there are no more legal moves or there are no unoccupied squares left, the game is finished and the player with the most discs of their color on the board wins the game. An example of this can be found in figure 1.3. In this figure the white player won the game as there are no more legal moves and this player has 32 discs on the board, opposed to 23 discs which the black player has.

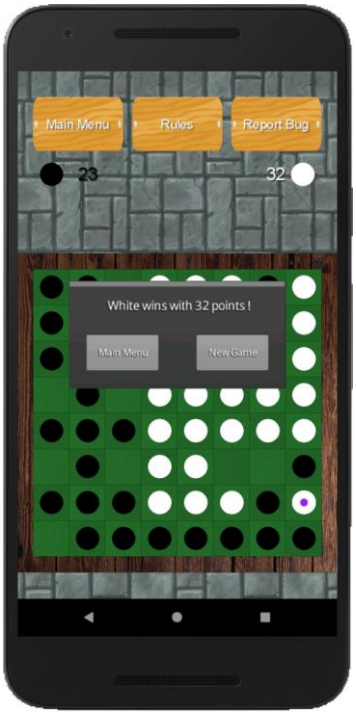

Fig 1.3 - The white player is the winner of the game as there are no more legal moves.

The game can be played in three different modes, the first being versus the computer. Here the player plays on their own device and there is no internet connection required. The second mode lets you play versus another human player, this mode is also offline, and you play by passing the device back and forth between the two players.

1.3 Structure of the document

This first part is simply an introduction to the project, especially as it pertains to the architecture phase. Part 2 is a description of all the architectural drivers that are essential for the application, while part 3 dedicated to the stakeholders and what impact their concerns will have on our development. Furthermore, in part 4 the reader will find a presentation of a selection of architectural views. Part 5 details the architectural tactics that will aid in the implementation of our chosen quality attributes, while the description of the design and architectural patterns we have chosen can be found in part 6. In part 7 the different views are

shown as they pertain to our application. A focus on consistency and inconsistency in our views can be found in part 8, and part 9 contains a rationale justifying our choice of architecture. At the end of this architectural document in part 10 and 11 is an updated description of the issues and changes that had to be made between the planning phase and the eventual implementation. Lastly are the references in part 12.

2 Architectural Drivers

When identifying an application's architectural drivers, one needs to look at all the requirements that will be architecturally significant.

2.1 Turn-based multiplayer

One of the main architectural drivers behind our game is the turn-based multiplayer aspect, allowing players to compete by each taking their turns. As the game logic is quite simple, it won't be very taxing on the hardware that's running the game and we won't be needing a backend server to handle the game logic. The nature of turn based multiplayer offline means that the game can be played on one device. It thusly can be programmed as a single device application.

2.2 In-game user modifiability

A point that has been important to us from the start is the options offered to the user to make the game worth revisiting. These options include but are not limited to adding a timer mode, adding handicap whenever you require a bigger challenge, playing with or without music and with or without visual aids for next move.

2.3 Modifiability

The modifiability quality requirement demands that you can easily change different parts of the application without much effort and without other parts of the code being affected. To achieve this the architecture needs to consist of modules with high cohesion and low coupling as specified in the modifiability tactics below. In addition, it also encompasses the ease of modification from a user perspective.

2.4 Usability

The architectural drivers for usability is solely concerned with the experience from a user perspective. For the ease of use and learning, we need to make the interface easily understandable as well as providing all the necessary information in a comprehensive way. The requirement is however highly connected to modifiability because to be able to achieve high usability it is a great advantage with rapid testing and opportunity to change the game easily from the standpoint of both developer and user and demands an architecture that supports user initiative.

2.5 Android

This application is being developed especially for Android. Apart from the fact that we are going to use Android Studios, this will impact the way in which we develop, and may influence the architecture.

2.6 Google Play

As we have no plan on monetizing this application our only goal from a business end will be to get our app released on Google Play. This simply means that our code needs to comply with Google Play's policies.

3 Stakeholders and concerns

It is important to be aware of all the different stakeholders and consider what concerns and goals they each have.

3.1 Teaching assistants and staff

The lecturer is the one that has given the task and is therefore, clearly, a stakeholder. The formal requirements come from the lecturer and other staff. The goal of this project is to fulfil all these requirements. The staff want to check how much the students have learned through this course and will do so by evaluate this project. To be able to give a fair grading they are dependent on all documentation being well-written.

3.2 The ATAM evaluators

The other group that will evaluate this project, is concerned with the quality of the architectural description and to see if it meets the quality requirements. A detailed documentation makes it easier to give a good evaluation.

3.3 The project team

The project team are concerned with making a high-quality game and have good documentation. The documentation is important to show that the architecture satisfies all its requirements. Another objective is to learn more about software architecture throughout this process. For the project team it is also important to have good cooperation and an easy assignment of tasks. We will make views that are easily understood so we can work on different parts of the project at the same time.

3.4 The end user

The end user is just concerned with how usable the game is and how fun it is. It is not concerned with the architecture if the gameplay and graphics are not affected. They might be concerned with the logical view because that affects how way the game is played. What kind of software architecture we use can also make higher or lower performance and that is also of interest to the end user.

4 Selection of Architectural Views

In table 1 the reader will find our selection of architectural views. This includes logical view, process view, development view and physical view. For each of this the purpose, stakeholder and UML notation are described.

Table 4.1: Architectural views

View	Purpose	Stakeholder	UML notation
Logical view	Describes the functionality the system provides to the end-users.	ATAM Course staff Developers End-user	Class diagram
Process view	Addresses the runtime behaviour of the system, the processes and how they communicate.	ATAM Course staff Developers	Activity diagram
Development view	Focuses on partitioning and grouping of software modules.	ATAM Course staff Developers	Component diagram Package diagram
Physical view	Addresses the mapping of software components and subsystems to hardware, and the connections between these.	ATAM Course staff Developers	Deployment diagram

5 Architectural Tactics

We have chosen modifiability and usability as the quality requirements we want to achieve. These are two quality requirements that are easily combined, and high modifiability is necessary for high usability.

5.1 Modifiability

The tactics for modifiability try to reduce the complexity, time and cost of making changes. Each module has their responsibilities and connected to each other in different ways. To achieve modifiability, it is important to look at the how the responsibilities are separated and intertwined in all of the modules.

5.1.1 Increase cohesion

Cohesion means how unified a module is in its responsibilities. We want to achieve high cohesion so that the module has deeply connected tasks. This reduces complexity and in combination with coupling makes it easier to see which modules that needs to change when we want a modification.

A way to do this is through increasing semantic coherence. This is done by moving responsibilities that do not serve the same purpose to another module. This can either be a new module or an existing one. We can check that we have high cohesion by hypothesizing likely changes to a module and see which of the responsibilities that will be affected by the change. If a responsibility is not affected, then we will move it.

5.1.2 Reduce coupling

Coupling describes how connected different modules are to each other. This describes how much module's responsibilities overlap and we want to keep this as low as possible. This is because high coupling means that changes to a module will affect others greatly and we need to make modification to even more modules. There are different tactics to achieve this.

5.1.3 Encapsulate

Encapsulation is adding an interface to a module. The interface consists of an API with all the responsibilities that the module has. This means that other modules only can interact with the module through the exposed interface and hides details that likely changes. By doing this we can keep the same functions that other modules use but change how they are implemented without affecting other modules.

5.1.4 Use an intermediary

Using an intermediary breaks dependency between different modules. This is used as a connected link between modules and responsibilities. By using an intermediary, we avoid that modules are directly connected and dependent on each other.

5.1.5 Restrict dependencies

This is done by restricting other opportunity to interact or depend on other modules. The way to achieve this is by restricting visibility or by limiting access to modules to authorized modules.

5.1.6 Refactor

This is done by moving code that has the same responsibility out into a new module. This often happens when different people are working on the same project and thereby easily produce code that should be refactored. In general modules that are affected by the same change should be refactored, because they probably are at least partial duplicates.

5.1.7 Abstract common services

When two modules provide similar services, we will replace them with a more general form. This can be done by parameterizing the description and implementation.

5.1.8 Defer binding

To defer binding means having the flexible to easily change artifacts later. This will be especially relevant for us because we want to have the opportunity to add more features and

change part of the game later and will use parameters as a way of doing this. Then we can change options like board size and number of players later.

5.2 Usability

Usability is basically how easy it is for the user to accomplish desired tasks and the support and feedback the system provides the user. The tactics can be separated into support of user initiative and support of system initiative.

5.2.1 Support user initiative

This is all the way a user can correct their errors and be more efficient in using the application. The tactics are **cancel, undo, pause/resume** and **aggregate.** Not all of those will be relevant for us as we are making a game where the rules say you can't undo for example. But the availability to cancel a game or pause and resume it will be very valuable.

5.2.2 Support system initiative

Support system initiative are the tactics where the initiative comes from the system and not the user. We will try to implement most of these to provide good usability. The three following points are measures to accomplish this.

5.2.3.1 Maintain task model

Maintain task model means that the system has a model of what the user is supposed to do. We will do this by assisting the user in showing all allowed moves, changes from previous move and use the task model is needed to do this.

5.2.3.2 Maintain user model

Maintain user model is what the system knows about the system and what the user knows about the system. We are going to use this by showing the user the instructions if it is a first-time user and let the user keep their settings.

5.2.3.3 Maintain system model

Maintaining system model is to have a model of the expected system behaviour. A way to facilitate this tactic is giving feedback to the user when the system is processing something. This will be useful when waiting to connect to another player in a multiplayer game.

6 Architectural and design patterns

6.1 Model-View-Controller

The Model-View-Controller, MVC, is an architectural pattern that separates an application into three components: model, view and controller. In our case, the model component contains all the actual code of the game and the game rules, the view is what the user sees, and the controller takes input from the user. Doing this simplifies simultaneous development and makes it easier to switch between different views presented to the user.

6.2 Abstract Factory

This is a creational design pattern used to ease in the creation of different classes that share similarities in functionality and variables. For our purposes, we are going to utilize this for the screens, as well as the controllers, since they all offer essentially the same functionality. This awards us far more advantages than limitations as the primary functions of both views and controllers are automatically integrated into the respective classes.

6.3 Singleton

The singleton pattern is a useful tool for encapsulating static variables that are shared between different classes and modules. This would be a good quality for our game state manager, seeing as this instance is parsed through all modules as it is needed.

7 Views

In this section the different views presented in part 4 will be showed and further elaborated. This is logical, process, development and physical view. The views will give a greater understanding of the different parts of the system and the relationship between them. The different views will show the application in different ways and will help to be able get an understanding of the application as a whole. These views are updated after implementation of the application.

7.1 Logical view

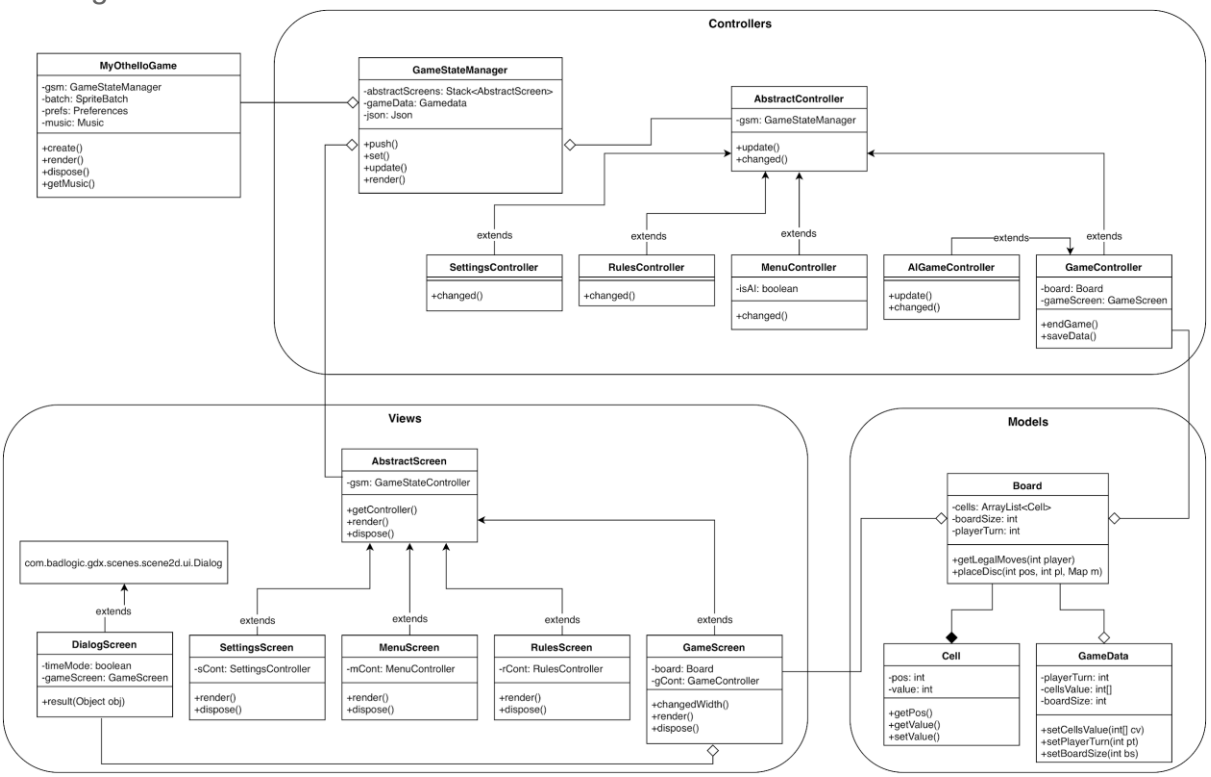

Figure 7.1: Logical view

In figure 7.1 the logical view is shown. We are using the MVC pattern and so the logical view is split into three main parts. The models are where all the game logic is located, here we find

the board and its cells which can be either empty, white or black according to the rules of othello, as well as the game data used to save the principal variables of non-ended game.

The views are what renders all the models and are split up into screens. The AbstractScreen is a superclass and implements some of the most common methods which all screens will need. The MenuScreen handles what happens before an actual game is started and will draw the main menu. GameScreen takes in an instance of the model Board and will use this to render a game of Othello. A screen also takes in an instance of the controller, this is because all user input is collected in the views and then sent to the Controller, which can then make changes to the models accordingly.

The Controller is what orchestrates the game's lifecycle, it tells the models and views what to do and when to do it. It also interprets the input from the views into standard commands and feeds these to the models. This is what decouples the models from the views and makes it easier to make changes in either section. If users are using different devices the input might be captured in different ways by the view, but in the end the Controller's interpretation of all the input will be equal, therefore making the game logic and manipulation of models the same across all devices. The Game State Manager is a bit different from the other controllers as it manages the stack of screens and the game data, used to launch an ongoing game. The GSM is the controller making the link between the ApplicationListener and the pair of Screen/Controller.

7.2 Process view

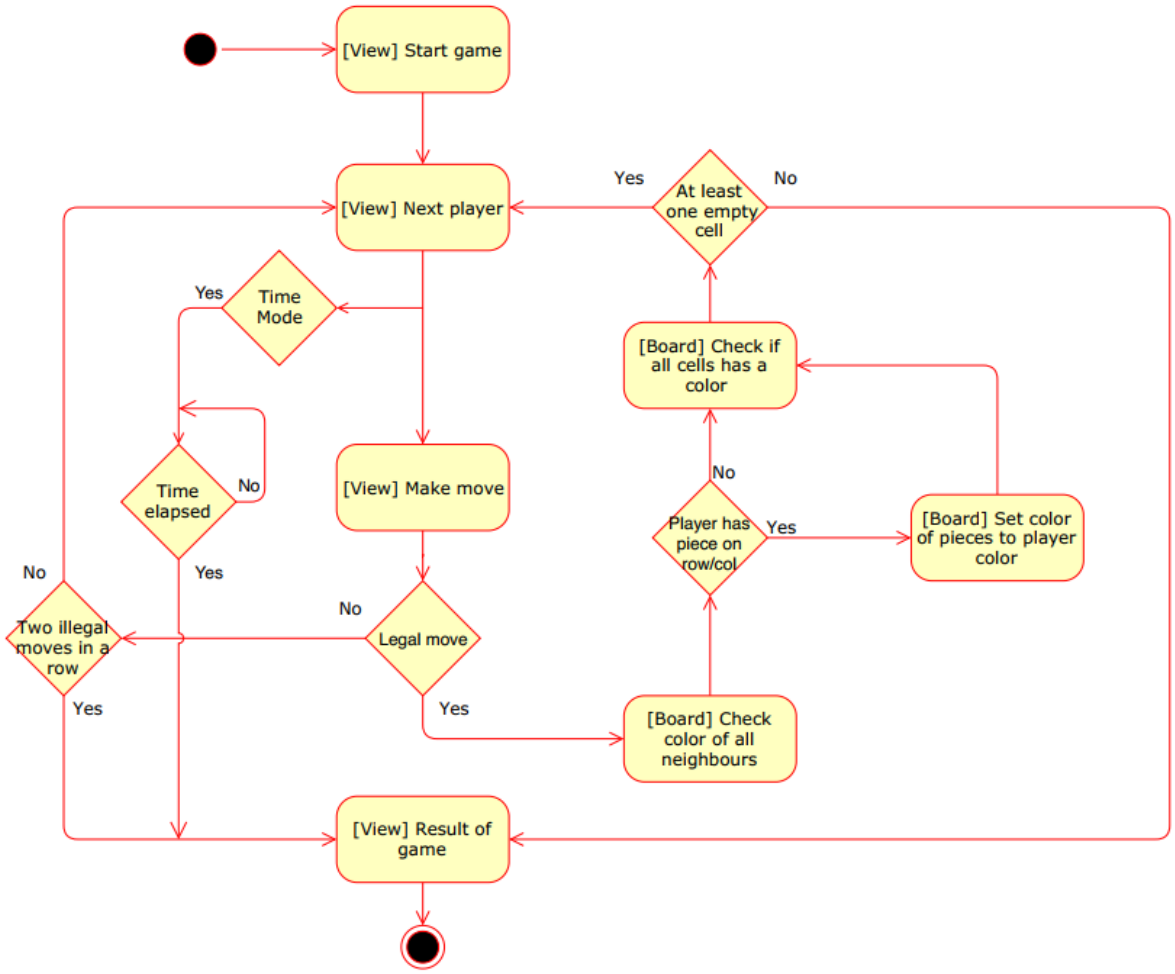

Figure 7.2: process view

The state machine in figure 7.2 shows all the possible states inside a game of othello. The Board class checks whether a player has any legal moves, if a player doesn't have any legal moves the turn goes to the next player. If neither player has any legal moves, the game is over and the winner of the game and its results are declared. After a player makes a legal move, all neighbouring cells on that specific row and column between the piece just placed and the player's already existing piece(s) are flipped to match the current player's color. After this the Board checks whether there are any empty cells left. If there are none, that means that all cells have a piece placed in it and there are no longer any legal moves left, so the game is over, and the results are shown.

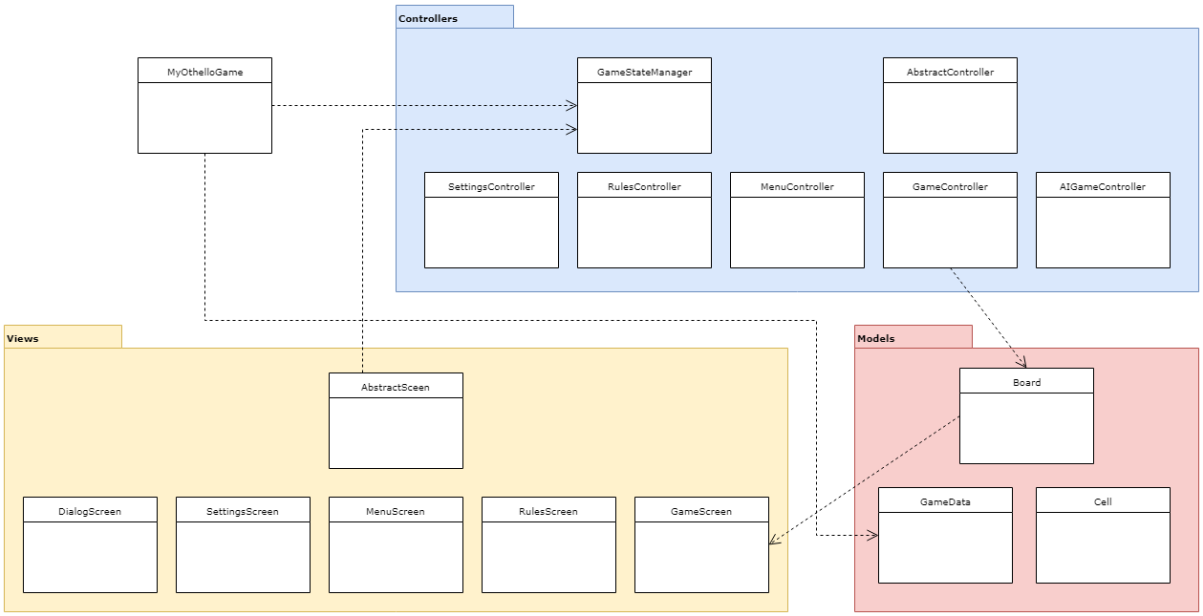

Figure 7.3: development view

In figure 7.3 the development view is shown. The project is divided into three main packages; controllers, views and models, which aligns with the MVC pattern. A lot of dependencies have been left out for the sake of readability, but the diagram mainly shows how the different packages interact with each other. The class MyOthelloGame is not part of any of the packages as it is the starting point of the program. When inside a game of Othello this architecture allows the view to update its controller whenever user input is detected, the controller will there after update the models accordingly and lastly the view will update with the latest information from the models. This allows independent and parallel development of all the different packages and makes both changes and additions easier to implement. The AbstractScreen and AbstractController provides a baseline for all the views and controllers, making development of future classes more efficient.

For the development phase, we used Github, GitKraken and android studio. Those softwares allowed us to work by pair on different task, simultaneously and then merge our code. It has simplified the coding part and allowed us to have a way to communicate the update we did. It also provided us a way to restore previous version if a bug appeared.

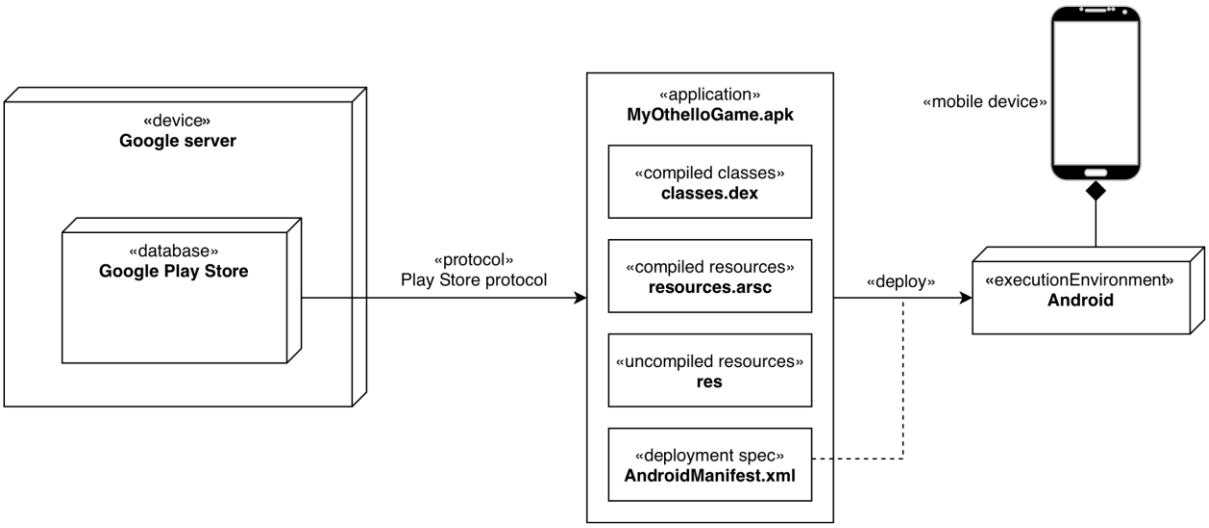

Figure 7.4: physical view

In figure 7.4 view of the physical component are shown. Google play store allowed us to make the game easy to download and update.

7.5 View consistency

We have not detected inconsistencies among the different views. The Logical view does not detail the entirety of the classes, their method and instance variables as it would not be readable but is detailed to understand how the implementation works. The process view describes the progress of a regular game of Othello and the implementation of the game sticks well to what is described here. The development view depicts how the different modules are organized when following the MVC pattern and the different class diagrams attest that we stick to the initial plan. As for the physical view it simply shows how the Google play game services and the Google Play Store are related to our application, namely that our Game is available on Google's platform.

8 Architectural Rationale

The choice in architectural pattern came mostly from the COTS where we chose to use the libGDX framework. Working with this framework there are great advantages with using MVC. This comes from libGDX way of rendering the screens, that makes it easy to split the view from the other parts. This was also one of the patterns we gained experience with from the first exercises and thereby seemed like a natural choice. This is also an architecture most of the group has some experience working in, and we appreciated the common footing from the start.

Our architecture also has advantages for our quality attributes. As for modifiability, the separation of modules in MVC pattern makes it simple to make changes to one of the modules without affecting others. This also aids in the usability as it makes it simple to get user feedback and other user initiative into fruition.

9 Issues

One of the issues with the architecture was the understanding that the design pattern we had chosen to use; pipes and filters wasn't as fitting as firstly assumed and wasn't implemented to the degree that we wanted. The major change to the architecture was removing the Google Play Game Services (GPGS) part. The reason for removing the GPGS was partly an architecture decision and partly because of the lack of experience from the group. From the architectural standpoint, we had difficulty implementing the GPGS library with our desired architecture. This was due to the fact that everything had to be handled through the android module. We knew this, however it proved to be more of a challenge than initially expected.

We had some ideas initially for how we would use the pipes and filters design pattern. Seeing as LibGDX render frames continually but still offers you the chance to update variables and models outside this render loop, it did not seem necessary when it came down to it. We ended up not using this pattern at all.

10 Changes

In this section you will find a collection of changes made to the architecture after initial delivery.

Table 10.1: Changes made to the architecture after first delivery.

Time of change	Changes made	Comment
5th April	Dropped options for online playability	Detailed in the implementation document
23rd March	Added AI agent for single player mode	Required an additional game controller, but did not impact rest of architecture
13th March	Dropped pipes and filters	Did not turn out to be necessary
9th March	Utilized Singleton pattern	Not intended in first de

11 References

[1] https://en.wikipedia.org/wiki/4%2B1_architectural_view_model
[2] https://otter.tech/an-mvc-guide-for-libgdx/
[3] Bass, Clemens, Hazman - Software Architecture in Practise, Third edition