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 this phase

This project was carried out as part of the course TDT4240 Software Architecture, with the core requirement of making a game containing a multiplayer component for either Android or iOS. The goal of the project is to learn how to design, evaluate, implement and test a software architecture through game development.

In this phase of the project we have been implementing and testing the architecture described in the requirements and architecture documents. At the implementation phase, you execute all the physical software development. At this point in the process, you have identified requirements and architectural driver, chosen the design and architectural patterns and are ready to start programming. At this point, changes can still be made, so the team needs to continually make decisions based on a series of constraints, like time, resources, schedule and functionality.

This document in addition to the requirements document and the architecture document tells the entire story of our project from start to finish, and details both where we succeeded and where changes and compromises had to be made.

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.

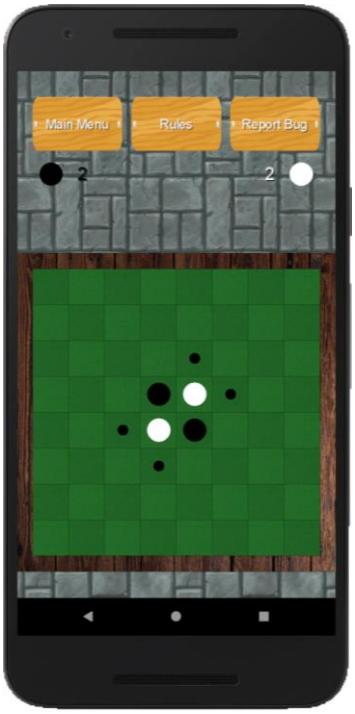

Fig 1.1 - Starting position for every game of Othello.

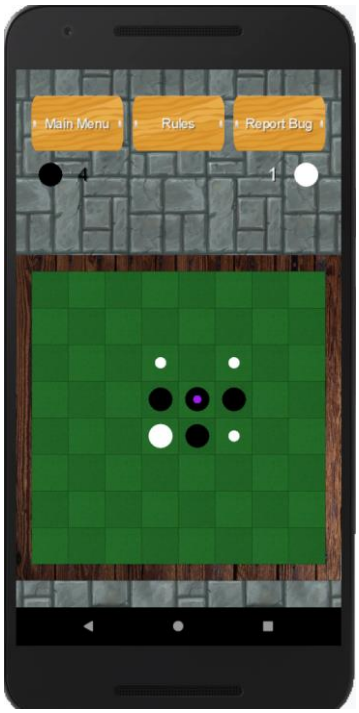

Fig 1.2 - A possible position after black has made one move.

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. Is 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.

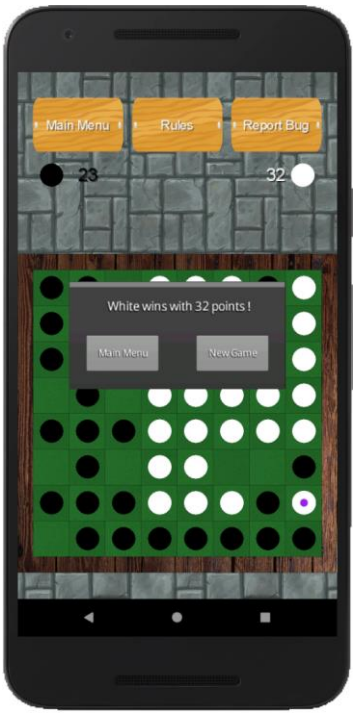

Fig 1.3 - The white player is the winner of the game as there are no more legal moves.

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. The last mode is an online player versus player, this requires both players to have their own device and a working internet connection.

1.3 Structure of the document

This document entails a description of the implementation and testing phase of our software architecture project. This first part is an introduction of the project, particularly as it pertains to implementation and testing. The second part details our implementation of the architecture. This second part does not in detail describe the relationship between our intended architecture and eventual implementation, as this is the focus of a section further down in the document. Part 3 contains a user manual that should tell the user how to install, compile and run the application as well as detailed description for intended use. In part 4, the reader will find a test report as well as detailed tests for all our functional requirements as well as the requirements associated with each of our quality attributes.

At this point, in part 5, of the document we start detailing where our eventual implementation differs from our architectural documentation. This section handles particularly the points in which the difference was of a significant character, but it covers most of the points discussed in the architectural document. The last part, part 6, of the document attempts to describe all the issues and problems we faced during our implementation test and offers some form of reflection over the project as a whole.

2 Design and implementation details

The game is implemented using LibGDX which is a framework that aids in the process of making games. This framework works well with the MVC that we planned in the architectural phase seeing as every frame is being rendered at periodic time intervals, which makes it a natural decision to handle the rendering in separate, specialized classes for the *views.* Because the characteristics of each view is similar, we utilized an abstract class that each screen subsequently extends, the same pattern is also used for the controllers. For a detailed overview of the screens and the structure proper, we have added class diagrams for the different modules as well as an external link to class diagram for the project in its entirety.

The game is being launched by the AndroidLauncher in the android module, and outside of this, the source code is placed inside the core module. The AndroidLauncher generates a new instance of MyOthelloGame. This in turn creates an instance of GameStateManager, basically just a stack of screens, that handles which screen and in turn which controller is being used at every moment. This single GameStateManager is subsequently parsed through each controller when they are instantiated.

2.1 MVC Architectural pattern

Given the nature of the LibGDX, it was useful to design and implement the code with a traditional MVC architecture. This framework generates graphics by systematically drawing each frame to the screen by batching 2D rectangles in specified textures. This makes it easy to maintain the MVC patterns as it gives us the opportunity to use the controllers to continually update the models and subsequently updating the views in real time and draw them on the screen.

2.1.1 Models

The models are split into three different classes that can be seen in figure 2.1. The cell class simply handles the characteristics of each field on the board, mainly the presence or absence of a disc and its respective color. The Board class is a collection of cells arranged in a meaningful order. The GamaData class is used to save the data when one leaves a game and want to resume it after. This class contains the values of the game itself, for instance which player should play next and other game variables.

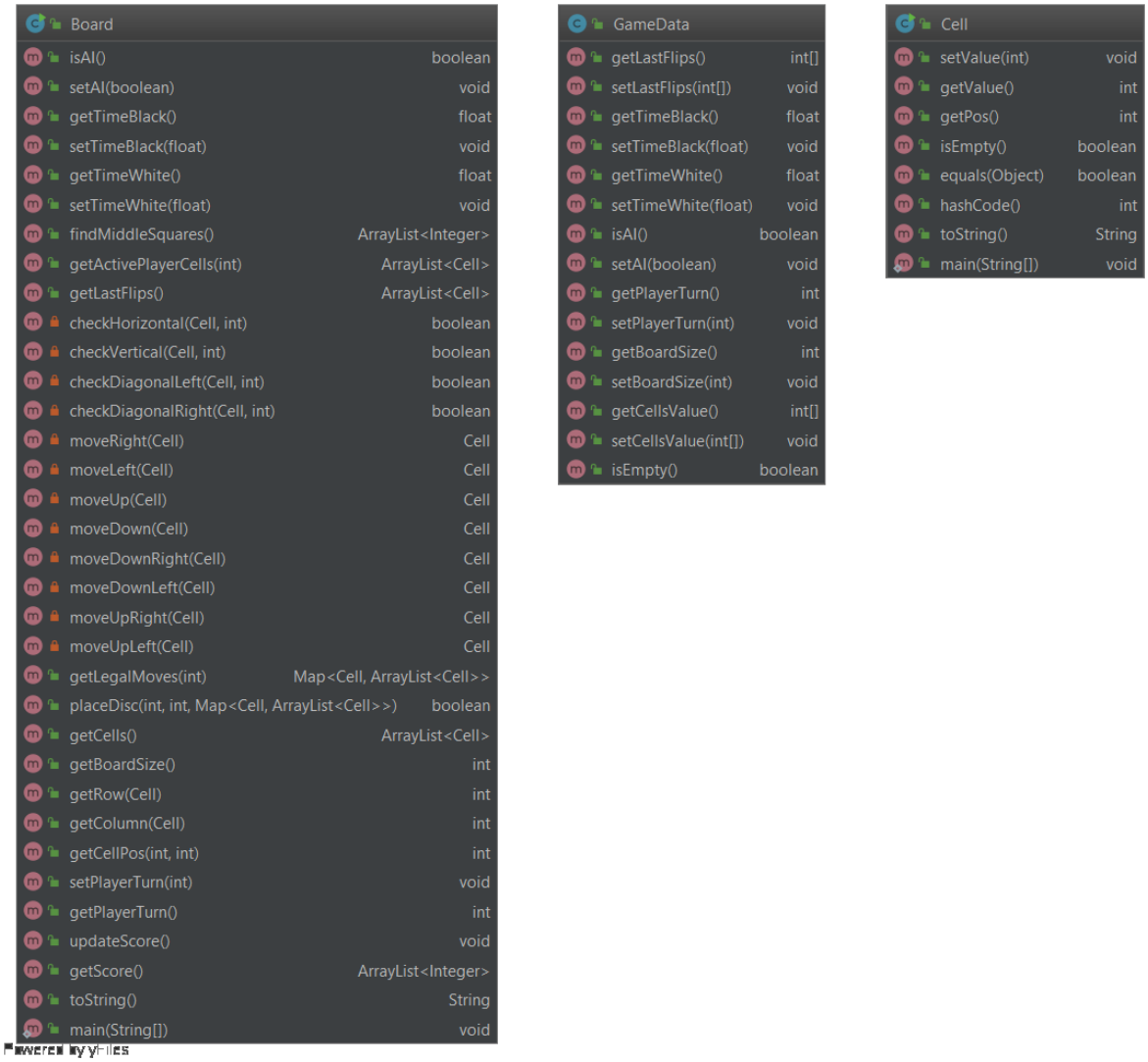

Figure 2.1: This is the class diagram for the models.

2.1.2 Views

The views are shown in figure 2.2. For the views we used the render function in LibGDX. Initially, the required views were the views for MenuScreen which is the first screen you encounter when you boot the application. Furthermore, the game itself required a combination of a representation of the game board, GameScreen, along with required information displayed in a DialogScreen. As one of our functional requirements (FR2) is for the player to be able to read the rules of the game, we also added a RulesScreen with intuitive visuals for ease of usability. And for the quality requirements U2, we implemented a SettingsScreen in order to gather all settings available for our game.

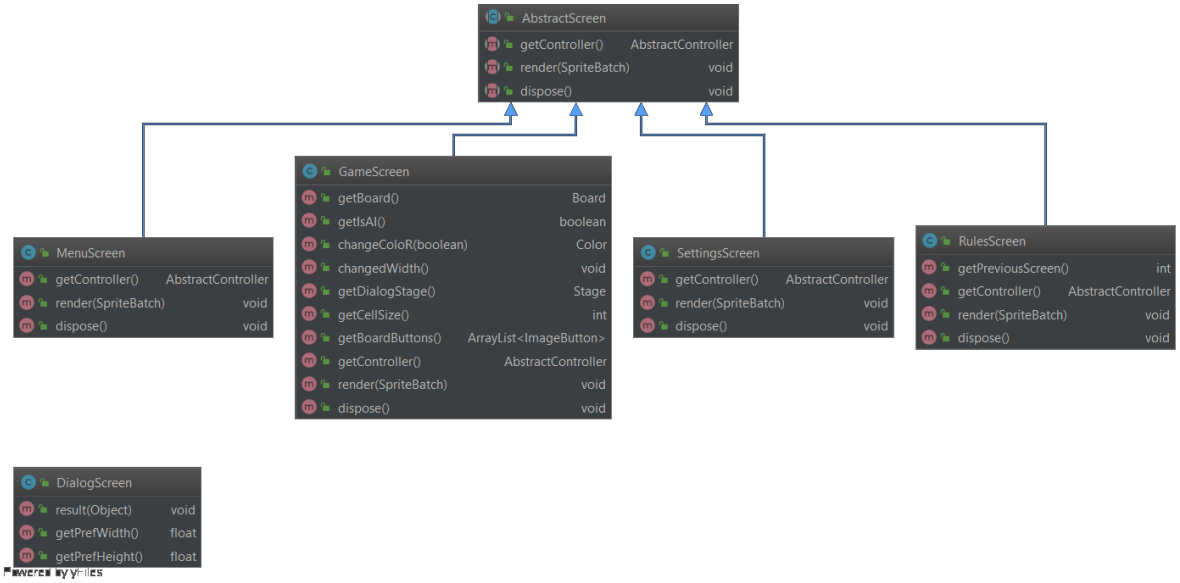

Figure 2.2: This is a class diagram for the views.

2.1.3 Controllers

The controllers are, like the views, split into classes on account of difference in functionality shown in figure 2.3. In reality we have a different controller for each screen, including different controllers for different game modes (i.e. single player and multiplayer). We utilized an abstract controller class that each of the controllers extend. The logistics of this will be covered in part 2.1.5, concerning the Abstract Factory design pattern.

The GameStateManager class contains functionality for the interaction between the ApplicationListener and the different controllers as well as the saving part. When a new screen is requested, it stacks this new screen and this screen generates a new controller corresponding to the screen. At launch, the MenuController is used for handling of the MenuScreen. This can forward you to either the GameController, IAGameController, SettingsController or the RuleController.

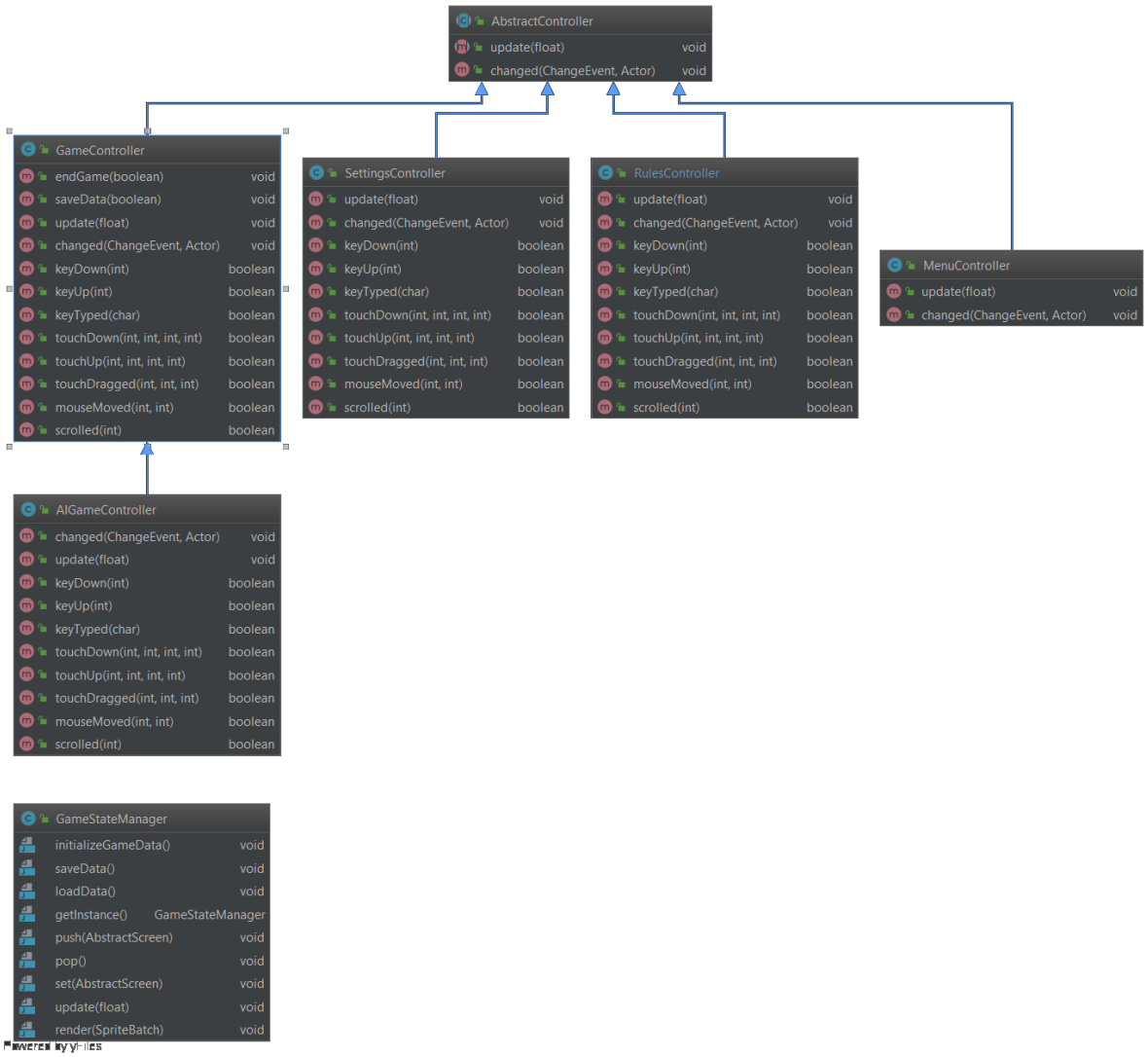

Figure 2.3: This is the class diagram for the controllers.

2.2 Google Play Game Services

As one of our initial functional requirements was to offer online multiplayer, we spent a fair amount of time researching and attempting to implement Google Play Game Services (GPGS) for turn-based online multiplayer. We were unfortunately not able to implement it satisfyingly in accordance with our selected framework and felt compelled to drop the FR in its entirety. This completely removed the necessity for implementing GPGS in the application.

2.3 Usability Requirement

To fulfil the usability requirement visuals to aid in the gameplay was added, as planned in the architectural phase. We also display all available options as soon as you boot the application. To make the user experience as pleasant and intuitive as possible, our application boasts non-offensive graphics and intuitive interfaces for all the screens. This is one of the points that we had to get confirmed in the testing phase. Already from the first screen, the user interface is

constructed with ease of use in mind and displays no hidden functionality. We have also included access to the rules for the game both in the main screen and in the game.

2.4 Modifiability Requirement

To achieve modifiability, it was important to look at how the responsibilities of each module are separated and intertwined with other modules. Amongst other measures, the code is structured with few overlapping sections of code to reduce coupling. The main coupling between distinct modules is limited to the necessary interaction between models, views and controllers. As models, views and controllers are separated on account of large variations in functionality and responsibilities, we achieved a high degree of cohesion. As well as this, so too are each of the screens governed by its respective controller. This in turn reduces coupling and subsequently the chance that changes to one module will affect any of the others.

The code is structured with sufficient documentation to make it readable for the initiated reader. The code is documented with standard format doc comments. Another method that was used to ensure flexibility is abstract common services, in which each of the screens extend our AbstractScreen class. This ensures that we can make changes to all the screens' responsibilities without altering them all individually.

2.5 Abstract factory Design pattern

As described above in the MVC-section, we utilized abstract classes for both controllers and views. This is clearly visible in our class diagram. As the screens and the controllers share much of the key functionality within each module, it was helpful to have the template from the abstract classes to ensure uniformity and limit amount of code duplicates.

2.6 Pipes and filters

Seeing as the application uses systematic and periodic rendering of screens, the pipes and filters pattern became a useful technique for the progression of change between players. This was our initial reasoning; however the eventual complexity of our implementation rendered this pattern useless.

2.7 Singleton pattern

The advantage of the singleton pattern is that it restricts a class to one instantiation. This means that if you have resources that are shared between classes and modules they will always agree on the value of this instantiation. We used the singleton pattern to generate the GameStateManager. This proved to be very useful as this is a static variable that is parsed between all the modules within one single MyOthelloGame instance.

3 User Manual

3.1 Description of how to install, compile, and run the game

The installation of this game requires either an device with Android 21 or higher and access to Google Play Services or Android Studios with an android emulator. To install the game, you have a multitude of options with different qualifications and challenges, so just follow one of the three alternative methods below. They are sorted by degree of preferability from most preferable to least preferable.

3.1.1 Downloading and installing app from Google Play Store

This is by far the easiest way to try out the application. The only thing required is that you have a mobile device that runs on an Android OS. To install the game, just follow this three-step process that should be familiar with most everybody.

1. Open Google Play Store - From your mobile android device, click on the Google Play Store icon and search for “Othello on the Rocks”
2. Install application - As this is a free application, this will be your only alternative action.
3. Open application - When installation is finished, you will get a notification, and you could either open the app directly from Google Play Store or open it from you home screen.

3.1.2 Install application directly to your device/virtual device from Android Studios

This method will still require that you either have a physical android device, otherwise you will have the option to install it on a virtual device on your computer. For your physical device, you will have to unlock developer mode to be able to install application directly from source code. This can be done by following the “Run on real device” section of this this fairly intuitive guide. If you on the other hand would like to use a virtual device, you can do so by following the “Run on emulator” section of the very same guide.

3.1.3 Run desktop version

This is in no doubt the least preferable way to run the game. The project code is implemented with a desktop application that calls the DesktopLauncher class. The app, however, was programmed and designed with mobile functionality in mind, so in addition to reduced quality in graphics you would also get a lower quality version of the intended controls. The advantage of this method is that it is fairly simple and doesn’t require an android device. You simply open the project in Android Studios, press the extending arrow for the configurations panel and choose the desktop application. When this is done, you simply press the “run” button, and the program will run in LibGDX’ default application launcher.

3.2 Screenshots from the game explaining how to play it

When opening the app, the first thing you'll see is the menu screen. This is shown in figure 3.1 and 3.2.

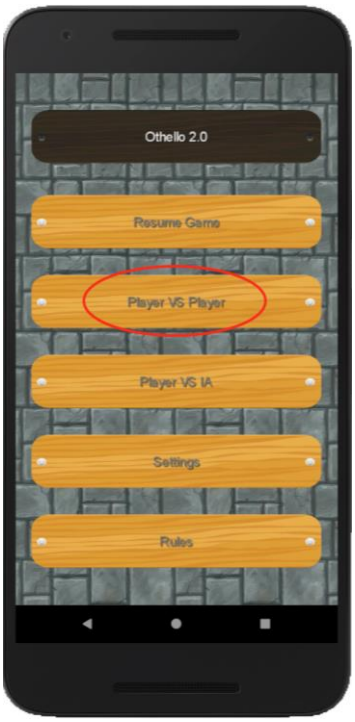

Figure 3.1: Press the 'Player vs Player'-button to start a new multiplayer game

Figure 3.2: Press the 'Player vs IA'-button to start a new singleplayer game.

.

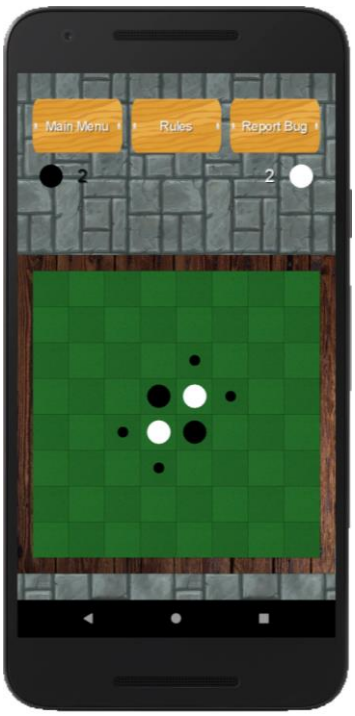

Figure 3.3: Starting position of a game

After you have decided what game mode you want to play, the game screen appear.

Figure 3.3 shows the starting position of the game, on both multiplayer and singleplayer mode. The smaller dots represent all legal moves. Press on one of them to start the game. Black always start.

The game then alternates between white and black players. If playing multiplayer mode, the smaller dots change color representing who's turn it is.

If playing singleplayer mode, an animation shows the AI player's move. When the smaller dots appear again, it's your turn.

When placing a disc at one of the legal moves, the opponent's discs in between your discs, either horizontally, vertically or diagonally, change to your color.

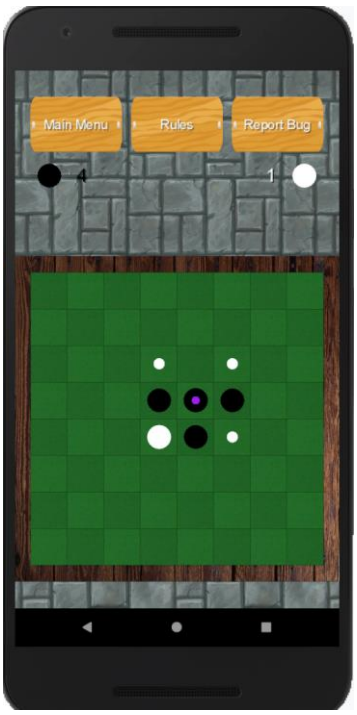

Figure 3.4: Multiplayer mode

In figure 3.4 you see the game screen after the first move, playing multiplayer mode. The purple dot shows what disc(s) changed color.

The overall score to each player is visible above the game board.

Figure 3.4: End state of the game.

When there's no legal moves left, the game ends. The player with most discs on the board, and therefore, the highest score, wins.

A window pops up announcing the winner and gives the options to either start a new game or return to main menu.

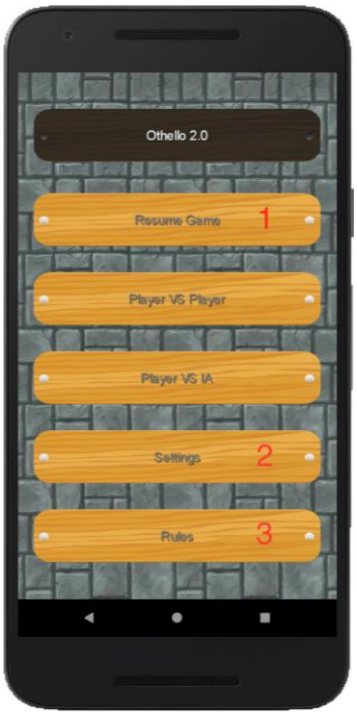

Figure 3.7: Main menu

At any time during the game you safely close the app or return to main menu and still load your current game by pressing the ‘Resume Game’-button (1) in figure 3.7.

Also found in the main menu is settings (2) and rules (3).

Figure 3.8 Settings menu first part

In the settings menu shown in figure 3.8 you can set a handicap (1) by using the slider. The highest handicap is 4, meaning 4 discs are placed in all corners of the game board. This gives an advantage to the player with white discs.

By pressing ‘Show Legal Moves’ (2) you toggle between showing and hiding the smaller dots representing legal moves during a game.

You can turn on or off either the score (3), the purple dots showing last flips (4) or the music (5).

Figure 3.9: Settings menu second part.

By scrolling down the settings menu shown in figure 3.9 you'll find the option for Time Mode (6). This feature is multiplayer-only and adds a timer for each player. You can also erase the current game save (7) or exit to the menu screen (8).

4 Test report

Towards the end of the project, when most of the game was finished, we started doing extensive testing of the functional and quality requirements which were set in the Requirements document.
There were both successes and failures, as not everything was implemented according to plan due to time constraints.

4.1 Functional requirements

In this phase we have tested all the functional requirements that we found in the requirement phase. The test report is shown in table 4.1 with the executer, date, time used, evaluation and comment.

Table 4.1: Test report of functional requirements.

Functionality	
F1: The user should be able to place their disc on all possible positions on the board.	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 20 seconds Success The user had no difficulty placing the discs.
F2: The user should be able to read the rules	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 1 minute Success The user was easily able to find and read the rules.

F3: The user should be able to choose between an online and offline game	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 None Failure This feature has not been implemented.
F4: The user should be able to quit the game and return to main menu.	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 10 seconds Success The user had no problems exiting to the main menu.
F5: The user should see the score at all time	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 5 seconds Success The user located the score without problems.
F6: If there are no possible moves for either players the game should end.	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 1 second Success The game promptly ended when the user was out of moves.
F7: Disc color should change if the disc has been surrounded.	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 1 second Success The intended discs were flipped as the user played through.
F8: The player with the highest score should win the game.	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 2 seconds Success The user had the most the most points when there were no more possible moves and was therefore shown a screen telling them they had won.
F9: The score should increase correspondingly whenever the player flips discs.	

Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 5 seconds Success The score increased as intended and was visible to the user.
F10: The player should be able to see all the possible positions for disc placement.	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 10 seconds Success The user could visually see all the possible moves at any given time.
F11: The player should be able to see who's turn it is.	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 5 seconds Success The user was able to determine which players turn it was by looking at the color indicator of the possible moves.
F12: In offline mode, the player should have the opportunity to leave a game and resume that game later.	
Executor: Date: Time used: Evaluation: Comment:	Robin Finstad 05/05/2019 1 minute Success The user was able to leave the game, close the app and later return to the same state of the game at which they left it at.

4.2 Quality requirements

In this phase we have tested all the quality requirements that we found in the requirement phase. The test report is shown in table 4.2 and 4.3 with the executer, date, environment, stimuli, expected response measure, observed response measure, evaluation and comment:

Table 4.2: Test report of quality requirements

Modifiability	
M1: Add a new view	
Executor: Date: Environment: Stimuli: Expected response	Robin Finstad 03/04/2019 Design time Add a view with rules to help the user understand the game

measure: Observed response measure: Evaluation: Comment:	60 minutes 75 minutes Success As the code as been made to be modular there were no problems worth mentioning, and most of the time went into getting the design of the screen right.
M2: Add a new player	
Executor: Date: Environment: Stimuli: Expected response measure: Observed response measure: Evaluation: Comment:	Robin Finstad 02/05/2019 Design time Add an extra player, beyond the standard two players 10 minutes None Failure After gaining a deeper understanding of Othello, we understood that adding another player wouldn't fit with the rules of the game. This feature was therefore never implemented.
M3: Change board size	
Executor: Date: Environment: Stimuli: Expected response measure: Observed response measure: Evaluation: Comment:	Robin Finstad 02/05/2019 Design time Change board size and see that the graphics respond correctly 2 minutes 2 minutes Success Changing the board size is as easy as changing a single variable inside the GameScreen class. All the graphics and game logic still show up and work as intended as all dimensions where built relatively to the board size.

Figure 4.3: Test report from usability requirements

Usability	
U1: Check the rules	
Executor: Date: Environment: Stimuli: Expected response measure: Observed response	Robin Finstad 01/05/2019 Runtime See if the user can find and understand the rules properly. 2 minutes

measure: Evaluation: Comment:	2 minutes Success The user was easily able to find and read the rules.
U2: Change settings	
Executor: Date: Environment: Stimuli: Expected response measure: Observed response measure: Evaluation: Comment:	Robin Finstad 01/05/2019 Runtime See if the user can find and change the game settings. 1 minute 2 minutes Success The user had no problems finding and changing the settings as they wished.
U3: Be able to start multiplayer	
Executor: Date: Environment: Stimuli: Expected response measure: Observed response measure: Evaluation: Comment:	Robin Finstad 01/05/2019 Runtime Start a game versus another player 30 seconds 15 seconds Success The user quickly found the correct option and started a local multiplayer game on the same device.

5 Relationship with architecture

As we ended up discarding one of our architectural drivers, which was the online mode for turn-based multiplayer, the architecture has suffered a couple of changes across the board. The change is the omission of Google Play Game Services (further referred to as GPGS) in its entirety. This also means that the architectural views look a bit different from our initial thought. We still ended up using the general concept of all our design and architectural patterns as well as the quality requirements chosen

5.1 Modifiability

In general, with modifiability we made it possible both for the developer and the user to easily change settings and parts of the game. This also increases the usability that was another

quality requirement. In the following section we have listed our modifiability tactics and how they relate to the original architecture.

5.1.1 Cohesion and coupling

To the best of our abilities, high cohesion is upheld across all modules. The difference from our initial plan is that we did not end up focusing on semantic cohesion, but we split the responsibilities in as small and specified modules as we found practical. The coupling between the different modules is very low. The main coupling is limited to the necessary interaction between views, models and controllers, and this process is aided for instance by the fact that every time a new view is being instantiated in the GameStateManager, that view will generate a new controller for that respective view. This means that you can change any controller without it affecting the view it corresponds to.

5.1.4 Encapsulation and intermediary

We did not end up using encapsulation as much as we initially planned. However, we used an abstract class for the views. This aided in reducing coupling, or at least parsing the strengths of the coupling for each module itself over to this abstract class. As for intermediaries, it did not turn out to be a necessity.

5.1.5 Refactor

To the best of our abilities, the code structure contains few duplicate classes, as each of the screens and their respective controllers have distinct responsibilities.

5.1.6 Abstract Common Services and binding

This was used for the controllers as well as the views, as described in the implementation and as initially planned. The models on the other hand shares little common functionality and it did not seem necessary to implement for them. We were able to defer from binding for the most part, as most of the game's values are parameterized. This is very helpful for modifiability, as changing a value for one single instantiation can dramatically modify the game.

5.2 Usability tactics

The application was implemented with the option to pause and resume a game. This is a case of supporting user initiative. We also made it possible to change different settings like music, help and board size. This makes it easier for the user to customize the game and this is also connected to modifiability. In the case of supporting system initiative, we maintained a task model. This is executed by showing allowed move and thereby having a model of what the user is supposed to do. We are also showing the last move and, in that way, also having a task model. We made a significant effort for supporting system initiative through having the system displaying valuable information for the user regarding legal moves and previous moves.

5.3 Architectural and design patterns

Not much changed here from our initial plan. Seeing as our biggest change was in the omission of GPGS and this did not affect any of the patterns we chose, we were able to follow through.

5.3.1 MVC

This pattern is described heavily in the implementation details (2.1.1). This went according to plan, however from the architectural documentation the implementation was not detailed to the point where we could anticipate all classes involved.

5.3.2 Pipes and filters

Did not end up using this as much as initially thought. The intention was to parse the output of a previous game state as input in the current game state, however this was done by a more continuous process and by saving data in variables outside of the rendering method. On the other hand, we ended up using this pattern to simulate the motion of pawn flipping.

5.3.3 Abstract Factory

This is described in fair detail in the implementation details and was utilized for screens as well as controllers.

5.3.4 Singleton

We did not initially detail this during our architecture phase, but we used the singleton pattern to generate the GameStateManager, this is fairly detailed in 2.7 Singleton pattern.

5.4 Views

The views have been updated in the architectural document to better fit the reality of the situation after implementation. The GPGS parts of the original delivery had to be removed as we dropped this functional requirement.

6 Problems, issues, and points learned

We spent a fair amount of time trying to implement online multiplayer but subsequently failed. This can be attributed to the initial lack of knowledge and time constraints for the group in accordance with the project. In hindsight, this made it easier to follow through on some of the modifiability tactics. Including the GPGS library as a dependency would have decreased cohesion and increased coupling between modules seeing as a turn-based multiplayer agent would have to be parsed through all classes that would use these functions.

As for problems not significant for the finished product, we encountered a few that took some time to resolve. This was mostly problems relating to the nature of LibGDX and the continuous rendering process. This is a new way of thinking, as all variables must be handled for each frame and brought on some problems during the process between each player's turn. This resulted in the eventual decision of adding a delay at the end of each turn. This can be experienced by a player but should not be of a significant inconvenience.

As a point of reflection, we could have added a more complex AI to the game in order to make it more interesting for single player mode. As it is, our AI will only offer a slight challenge for uninitiated players, while still serving its purpose by making available a single player mode. By virtue of busy schedules, at the time of our ATAM evaluation, our documentation was still

lacking a fair bit of depth. If we had done a more detailed first draft, we could have gotten more valuable feedback from the other group before we started the implementation.