INFO-2051: Object-Oriented Programming on Mobile Devices

Report

Maxime Goffart
180521

Olivier Joris
182113

Academic year 2020 - 2021

Contents

1 Introduction

10 to 15 years ago, every company wanted to have a website because it was the best way to reach the maximum number of potential users or customers depending on the market in which they are/were operating.
Nowadays, people spend more and more time on their smartphones, and web browsing on a smartphone is not always very convenient. Thus, everyone now wants to have a mobile app on both Android and iOS platforms. However, it required maintaining two codebases, one for Android and one for iOS, which is time-consuming and expensive. Nevertheless, this has been solved by cross-platform solutions such as React Native and Flutter.

In this report, we will present our app idea and its structure. We will then take a more in-depth look at its various components and the object-oriented approach we chose. Finally, we will introduce the pattern we chose as long as the technical challenges we faced during the development of the app.

2 Presentation of the app

Our app is a mini-games app. It contains a collection of mini-games, which are all represented by a grid.
Furthermore, our app proposes various auxiliary features. It contains a system of connection and registration so the user's scores can be stored, and he/she can see how well he/she is doing compared to all the other players who have played the same game. The user can add friends or create groups of friends and invite his/her friends to these groups.
Moreover, it is possible to play all the games currently available sequentially in a randomly generated order.
Additionally, we are proposing one challenge per game that the user can try to pass and the user can see who was able to pass a given challenge.
However, if the user is afraid of having some data stored about his/her usage of the app, he/she can use the app as a guest, but he/she will not be able to add friends nor create groups of friends. His/her scores will not be registered, so they will not appear in the different games' leaderboards, but the user can still see the leaderboards.

3 App structure

When the user launches the app, he/she has three possibilities: he/she can enter his/her credentials and connect to the app; if the user is not already registered, he/she can sign in; or, as mentioned earlier, the user can access the app as a guest. After he/she has chosen one of these possibilities, the user will arrive on the home page.
On the home page, there are 4 different panels (see *figure 1*). The first panel allows the user to access the launcher of one game, see the challenges, or start a new shuffle across the different games. The second panel will redirect the user to the different games' rankings, and the third panel will route him/her to the "social part" of the app. Finally, the last panel gives the user the possibility to change his/her email address and password.
Panels three (friends) and four (settings) are not accessible if the user is not logged in.

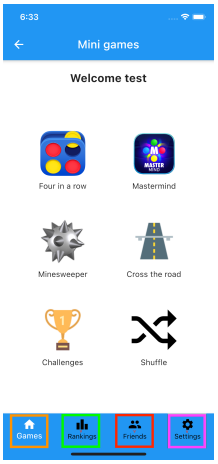

Figure 1: Home page with the first panel being active

4 Code structure

First, we will look at the source code's general structure. Then, we will take a more in-depth look at the different parts.
We will only explain the methods that we consider relevant to be discussed in this report during the more in-depth look. The others are documented in the code when their identifiers are not self-explaining.

4.1 General structure

The source code of the app is split into two main folders, which are `views` and `logic`. As induced by their names, the sub-folder `views` contains the code related to the different views, while the sub-folder `logic` contains the code related to the logic behind each view and other logic components that are not linked to a particular view.

4.2 Connection

The connection is handled by the files `views/connection_view.dart`, which contains the class `ConnectionView` and `logic/connection_logic.dart` which contains the class `ConnectionLogic`. The former allows us to build the connection page's view, which contains a form used by the user to enter his/her credentials to connect to the app. The form uses the class `InputDecorationBuilder`[1], which contains a method used to build an InputDecoration for each field of the form. It also allows the user to route to the page to register or to access the app as a guest.
The latter allows us to verify if the entered credentials are valid or not. It uses the class `DbConnect` (see *4.4*) to access the database and the class `FirebaseAuth`[2] to access Firebase's authentication system. The class `ConnectionLogic` is used throughout the app to access information about the user, such as if he/she is using the app as a user or a guest and his/her username if he/she is logged in as a user. Finally, it contains a method used to sign out the user when he/she decides to do so.

[1] See file `views/custom_elements/input_decoration.dart`.
[2] We did not implement this class. It is part of the package `firebase_auth` used to interact with Firebase's authentication system.

4.3 Registration

The registration is handled by the files `views/register_view.dart`, which contains the class `RegisterView`, and `logic/register_logic.dart`, which contains the class `RegisterLogic`. For registration, we use the authentication functionality proposed by Firebase and a database to store more information about the users (see *4.4*).
The first class allows us to build the registration page's view, which allows the user to create a new account.
The second one is used to check whether the information given by the user can be used to create an account. It verifies that another account is not already using the given email, the username is not already taken by someone else, and the password is strong enough[3]. As for the class `ConnectionLogic`, it uses the classes `DbConnect` (see *4.4*) and `FirebaseAuth`.

4.4 DbConnect

This class is in the file `logic/db_connect.dart`, and it is only an element of logic (no associated view). We are using Cloud Firestore, a NoSQL database as a service proposed by Firebase, to store data.
When the class is instantiated, it connects to the database. This class also contains 4 methods. One allows to update the score for a given game and a given username, and it will verify that the user does not already have a better score for the given game in the database. We decided to store only the best score for each game for a given user because storing every score for a given user and a given game was not useful for our app.
The class also contains methods to get all the users registered to the app, get the maximum score to achieve to pass a challenge, and to retrieve if a given user has achieved the challenge for a given game.

4.5 Home page

The home page is handled by the files `view/home_page_view.dart` which contains the class `HomePageView` and `logic/home_page_logic.dart` which implements the class `HomePageLogic`. The first one is used to build the UI of the home page. It displays the panel provided by the logic component `HomePageLogic`. You can see a screenshot of the home page with the first panel in *Figure 1.*
The second one, which is the home page's logic component, builds the different panels of the home page, and provides the one being active to the view in order to be displayed. It contains a method named `changeView`, which is the callback method for the buttons at the bottom of the UI used to change the displayed panel.

4.6 First panel

The first panel of the UI is implemented in the file `view/games_list_view.dart` in the class `GamesListView`. It is only a view element (no associated logic).
It builds the first panel's UI, allowing the user to choose between the launcher of the available games, the page relative to the challenges, or the page to launch a new shuffle across the different games.
Thus, it is using the classes representing the launcher of each game (see *4.7*). It is also using the method `gameButton` from the abstract class `CustomGameButton` of the file `views/customs_game_`

[3]FirebaseAuth sets the expected format of a password.

`button.dart`. This method builds a button representing one of the previously mentioned possibility given some parameters[4].

4.7 Game launchers

A launcher allows the user to start the game in one of the available game modes. Each game has its launcher based on the different game modes available for that particular game. Each of these launchers uses the abstract class `StartGameView`, which contains methods used to build the game's launcher given some parameters. The launcher for a given game is implemented in a file named x`_start_view.dart` where the name of that particular game replaces x. All the files related to the launchers can be found in the folder `views/start_views`. Class `StartGameView` uses the view of the different games (see *4.14*) in order to be able to launch them.

4.8 Leaderboards

Leaderboards, also known as rankings, are one of the essential auxiliary features of our app. They can be reached through the second panel of the home page. Each user[5] can access these rankings and see the best score by each user who has played a given game at least once. If the user is logged in, he/she will be able to choose between all the scores or only the scores of the user's friends (see *4.9*).
The leaderboards are implemented in 3 different files: `views/leaderboard_view.dart`, `views/leaderboards_game_view.dart`, and `logic/leaderboard_game_logic.dart`.
The first one contains the UI of the menu for the leaderboards. The user will be able to choose one game using a button and will be routed to the ranking of that particular game. It is using the class `CustomGameButton`[6].
The second file is used to build the UI of the leaderboard for a given game.
The last file contains the logic for the ranking of a given game. It allows querying the data from the database given the game for which we want to obtain the ranking. This class is using the class `ConnectionLogic` (see *4.2*) which itself uses the class `DbConnect` (see *4.4*) because it needs access to the user's information and to the database.

4.9 Friends

Friends are another one of the essential auxiliary features of our app. It is offering multiple possibilities, which are accessible through the third panel of the home page.
All these possibilities are implemented in 2 different sub-folders: `views/friends` and `logic/friends`, with the first one containing the views and the second one containing the logic elements. Some of these files are using the class `ConnectionLogic` (see *4.2*) because it needs access to the user's information and to the database.

4.9.1 List of friends

Each registered user has a list of friends. The user can add friends based on a list of registered users to the app but who are not friends with the user already or whom the user has not already requested as a friend. Each friend request can be accepted or denied by the requested user. The list of friends is then visible by itself or can be used in the leaderboards (see *4.8*) to restrict the displayed scores only to the ones of the user's friends.

[4] See code for the explanation of the parameters.

[5] The user can access these even if he/she is using the app as a guest. However, if the user is using the app as a guest, his/her scores will not be registered. Thus, he/she will not appear in the rankings.

[6] See file `views/custom_elements/custom_game_button.dart`.

4.9.2 Groups of friends

The user can create groups of friends. For a given group of friends, the user can see the members or add new member(s) from his/her list of friends.

4.10 Settings

The settings are only accessible if the user is logged in and can be reached by choosing the home page's fourth panel. On the settings page, the user will be able to modify his/her email address and password.
Settings are implemented in 2 files: `views/settings_views.dart` and `logic/settings_logic.dart`.
The first is used to build the UI of the settings page. It is itself using the class `InputDecorationBuilder`[7] and the class `ConnectionLogic` (see *4.2*) to know if the current user is logged in or not.
The second one contains methods used to update the email or the password in the database[8]. Thus, it uses the classes `ConnectionLogic` to access the connected user's information, which itself is using `DbConnect` (see *4.4*) to communicate with the database.

4.11 Shuffle

Shuffle's mode allows the user to play one game of each game available in the app in an order defined by a randomly generated sequence.
The view element defined in `views/shuffle_view.dart` allows to build the UI of the shuffle menu based on the randomly generated sequence. It offers the user the possibility to start the next game of the sequence or generate a new random sequence.
The logic element in `logic/shuffle_logic.dart` generates the sequence of games and returns after a click on the next game button of the UI, the view of the game to play. Thus, it is handling the different views to be displayed and the shuffle's status (not started, ongoing, or finished) based on an enumeration named `ShuffleStatus`.

4.12 Challenges

We are proposing one challenge per game available. To pass each challenge, the user has to achieve a score below a certain threshold. It requires logging in because we need to store if the user has passed the challenge. The user can also see who was able to pass a given challenge.
The views linked to the challenges are available in the sub-folder `views/challenges`.
One of the file in this sub-folder is used to build the UI of the challenge page. For each challenge, it displays a description of the challenge and a red cross if the user has not passed the challenge yet or a green check if he/she has passed the challenge.
The other file in the sub-folder is used to build a list of users who were able to pass a given challenge.
The logic is implemented by 2 classes (one corresponding to the page's logic and one corresponding to the logic of one challenge), both being implemented in `logic/challenge_logic.dart`. They are using the class `ConnectionLogic` (see *4.2*), which uses the class `DbConenct` (see *4.4*), to access information about the user and communicate with the database.

[7] See file `views/custom_elements/input_decoration.dart`.
[8] It is modifying Firestore and FirebaseAuth.

4.13 Game elements

The views and logics of the elements of a game are respectively implemented in the folders `/views/games_elements` and `/logic/games_elements`. The former contains the implementations of the classes corresponding to the views and the latter contains the logic of the elements of each game. In order to take advantage of the object-oriented paradigm, the classes corresponding to each of these elements are all subclasses of abstract classes implementing their common characteristics[9]. The highest classes in the hierarchy are respectively `GameElementView` and `GameElementLogic` which are partially implementing some general methods. Then, the class `GameElementLogic` can be specialized in `StatusBasedElement` corresponding to a game element having a status which is updated while playing such as Connect4, Mastermind, and Minesweeper cells. It can also be specialized in `PositionBasedElement` corresponding to a game element having a position which is updated while playing such as Cross the road cells.

4.14 Games

The views and logic elements of games are respectively implemented in the folders `/views/games` and `/logic/games`. The former contains the implementations of the classes corresponding to the views and the latter contains the logic of each game. In order to take advantage of the object-oriented paradigm, the classes corresponding to each of these games are all subclasses of abstract classes implementing their common characteristics[10]. The highest classes in the hierarchy are respectively the `GameLogic` and `GameView` with `GameLogic` using matrices of `GameElementLogic` and `GameElementView` and partially implementing some general methods. Then, the class `GameLogic` can be specialized in `TurnBasedLogic` corresponding to the logic of a turn taking game such as Connect 4, Mastermind, and Minesweeper. It can also be specialized in `RealTimeLogic` corresponding to the logic of a real time updating game such as Cross the road. Each game logic is composed of subclasses of `GameElementLogic` and `GameElementView` corresponding to elements of the corresponding game.

4.14.1 Four-in-a-row

The view and logic of this game are respectively implemented in the `connect4_view.dart` and `connect4_logic.dart` files. When the launcher of the game is launched via the menu, the user can choose between two game modes : playing against someone else on the same device or playing against the computer. After having chosen a game mode, the user can be redirected to the game and play by pressing the column in which he/she wishes to place a token when it is his/her turn to play.

4.14.2 Mastermind

The view and logic of this game are respectively implemented in the `mastermind_view.dart` and `mastermind_logic.dart` files. When the launcher of the game is launched via the menu, the user can choose between two game modes : playing with someone else who will choose the combination he/she has to guess on the same device or let the computer generate the combination randomly. After having chosen a game mode, the user can be guided by the application : the offer can choose a combination by pressing on the color buttons and the guesser can make proposition by clicking on the same buttons and choose the column in which he/she wishes to place this color. Then, the computer returns clues to the user : a black pawn indicates that one

[9]This can be observed in section 5.
[10]This can be observed in section 5.

of the proposed color is well placed and a white pawn indicates that one of the proposed color is part of the combination but not in the right place.

4.14.3 Minesweeper

The view and logic of this game are respectively implemented in the `minesweeper_view.dart` and `minesweeper_logic.dart` files. When the view of the game is launched via the menu, the user can play by clicking on the cell he wants to reveal. The first cell on which he/she is clicking cannot contain a bomb. Then, when a cell is discovered, a number indicating the number of bombs in the 8 neighbor cells of the discovered cell appears to help the user to reason about the places of the bombs.

4.14.4 Cross the road

The view and logic of this game are respectively implemented in the `cross_road_view.dart` and `cross_road_logic.dart` files. When the view of the game is launched via the menu, the user can move the character by tapping on the arrow corresponding to the direction he/she wants to go.

4.15 Player

This abstract class is implemented in the `/logic/player/player.dart` file. It contains useful information about a player and is specialized for each one of the game we propose.

5 Object-oriented approach

The object-oriented structure of our application can be observed on *Figure 2* and *Figure 3* which are respectively representing the view and logic part.

6 Pattern

We chose to use the Provider pattern to handle states of the application because this pattern seemed quite intuitive to us. Each view of our application which needs state management is linked to its logic using a provider. So, it is updated when this logic notifies the associated widgets of the view.

7 Technical challenges

One of the principal challenges we encountered while developing this application was to make the Connect 4 computer algorithm. This algorithm is associating a score to each move the computer can play and then it plays the move with the highest score among all these moves. This algorithm can be improved by exploring a certain depth of the game tree before evaluating a score: this would in fact amount to use an H-minimax algorithm. Indeed, with the current implementation the computer can give the win to the player because the move he plays is only taking into account the actual state of the game and not some other deeper in the game tree. We decided to stick with the current implementation because the algorithm is running on the mobile device and some mobile devices do not have powerful processor nor a decent amount of memory. We could imagined a new version that would send the state of the game to a server

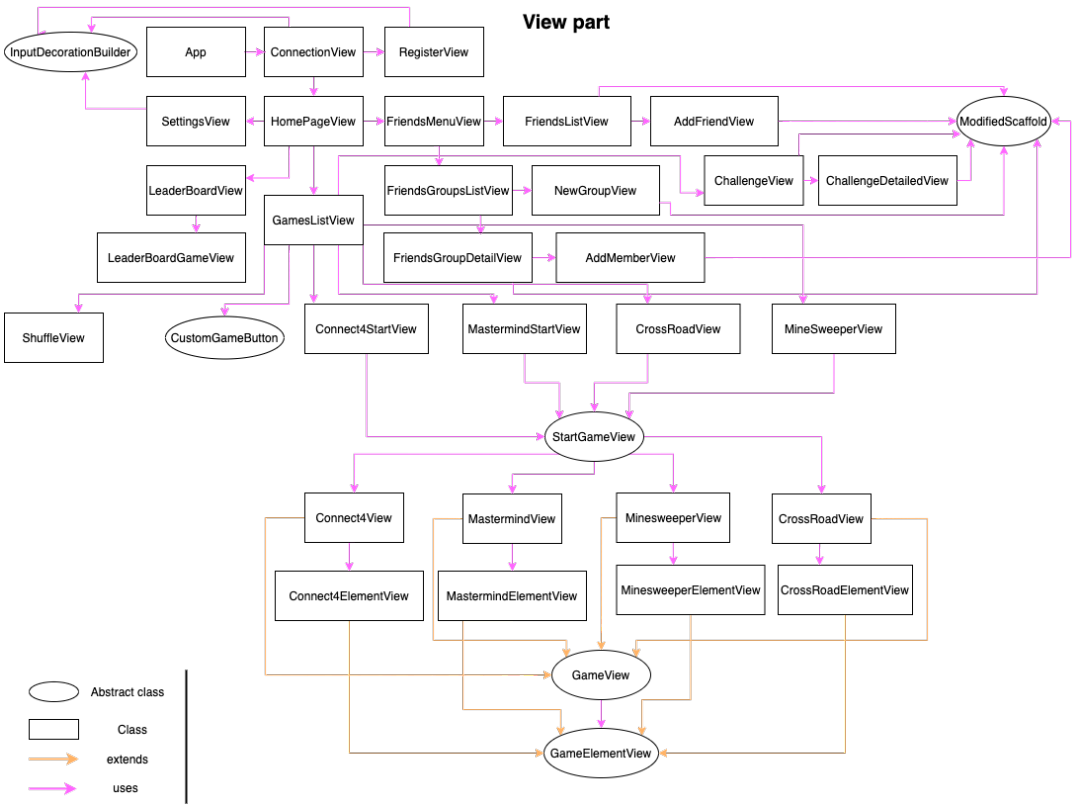

Figure 2: View part of the OOP diagram.

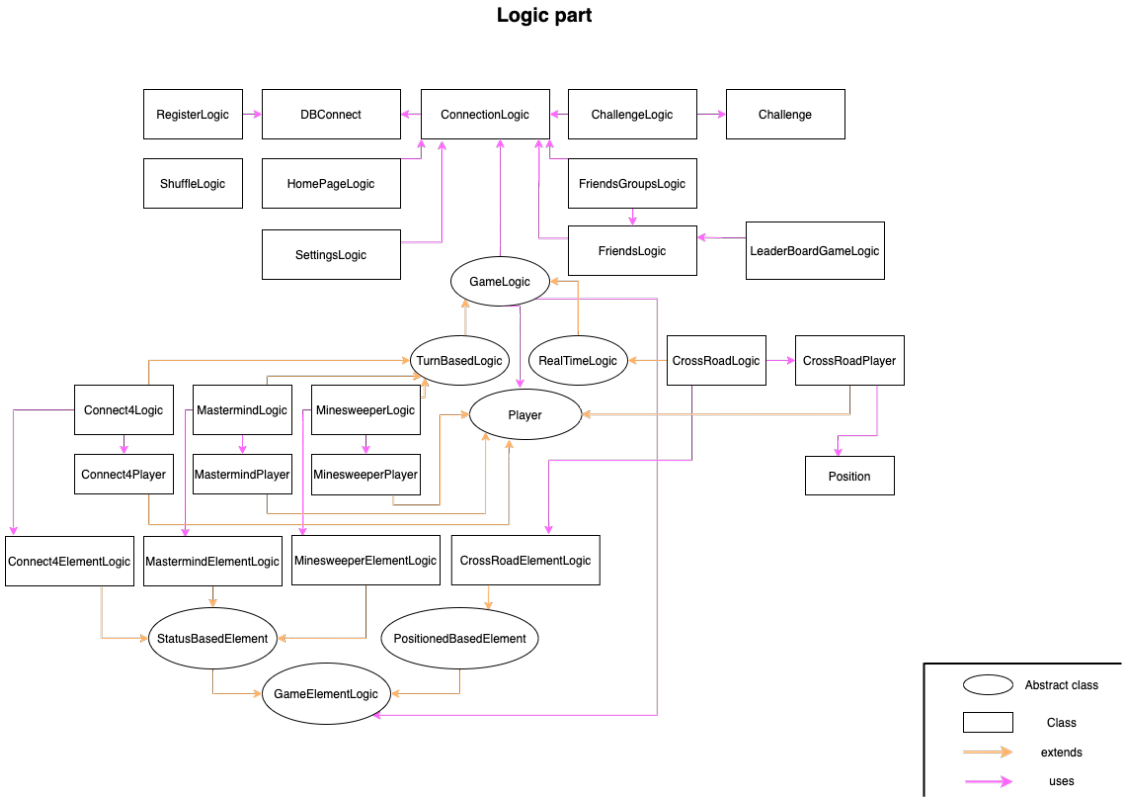

Figure 3: Logic part of the OOP diagram.

which would performed the H-minimax algorithm and would send back the best move to the mobile device but it would require an internet connection for communicating with the server.

Another challenge we encountered was dealing with Firestore because it is a NoSQL database which is different from the relational databases we are used to. This adaptation was nevertheless quick.
An alternative would have been to set-up and run our own server to handle HTTP requests with a relational database. However, it would have taken some (maybe plenty of) time to configure it and ensure a minimum of security which was not the case with Firestore. Since we were only 2 for majority of the time, we did not choose this option. Yet, it could be an improvement and could also be used to improve the Connect 4 computer algorithm.

Moreover, it was the first time we had to deal with asynchronous operations, so we had to practice a bit to feel comfortable with them.

Finally, implementing all the features we planned for this application while being two instead of three was not easy but we did it.

8 Conclusion

In conclusion, this project allowed us to apply a lot of concepts we saw in our curriculum in order to achieve a rather large application using the object-oriented paradigm. Moreover, it allowed us to learn new concepts that sometimes required self-training, which is important as computer science students.