Requirements and Analysis Document for Boardbook

Aaron Sandgren, Alexander Ohlin, Carl Holmberg, Dino Pasalic, Rasmus Rosengren

2019-10-25
Final Version

1. Introduction

The application Boardbook is a Social Media-esc android app that allows users to save matches of different board games. These matches can then be viewed later and contain information such as what players were in the match and who won. Furthermore, users can select game specific details such as what team or role a player might have been. Boardbook also allows users to add each other as friends to easier save matches. This allows users to also view what board games their friends have played recently. Additionally, the application tracks statistics on your performance and can show your win rate both overall and per board game.

The primary audience of Boardbook is people who frequently play one to many different board games. Specifically, it caters to users who want some of the functionality gained from digital board games while still playing the analog version of said board game. The purpose of Boardbook is to provide tools to the user so that they can easily store matches of board games together with who they played with at the time and what game was played. Since users can also see their statistics they will be able to easily see their performance over long periods of time and track whether they have gotten better or not.

1.1 Definitions, acronyms, and abbreviations

MVP: Model-View-Presenter
SDD: System Design Document
Activity: A specific screen in the user interface
Fragment: A subsection of an activity, used for some specific functionality

2. Requirements

2.1 User Stories

Starting the application

Story Identifier: US001
Story Name: Starting the application
Status: Done

Description

As a user, I want to be able to start the application so that I can see if it works.

Requirements

Functional:

- There exists a prototype of the app that can be run.

Non-functional*:*

- Everyone has their development environment set up and are able to run the app.

Availability:

- The app can be started 24 hours a day, 7 days a week, every week of the year.

***Security*:**

- Don’t wipe the harddrive of the device starting the application.

Interacting with an interface

Story Identifier: US002
Story Name: Interacting with an interface
Status: Done

Description

As a user, I want to be able to see an interface and be able to interact with it so that I see the application is running.

Requirements

Functional:

- There needs to be some sort of interactable component.
- There needs to be some sort of response to the interaction.

Non-functional*:*

- Everyone has their development environment set up and are able to run the app.

Availability:

- The app can be started and interacted with 24 hours a day, 7 days a week, every week of the year.

***Security*:**

- No security requirements

List of games

Story Identifier: US003
Story Name: List of games
Status: Done

Description

As a user, I want to be able to see the games that I can log my matches for so that I can know what games are supported.

Requirements

Functional:

- Load games from some sort of persistence solution.
- There needs to be a way to list games in a list of some sort.
- There needs to be a way to scroll if there are too many games.

Non-functional*:*

- Some sort of persistence solution needs to exist.

Availability:

- No availability requirements.

***Security*:**

- No security requirements

Save a match

Story Identifier: US004
Story Name: Save a match
Status: Done

Description

As a user I want to be able to save a match of a game so I can see the match later.

Requirements

Functional:

- A Match must be able to be stored in some sort of persistence solution.

Non-functional*:*

- There needs to exist some sort of persistence solution.

Availability:

- No availability requirements.

Security**:**

- No security requirements

Create account

Story Identifier: US005
Story Name: Create account
Status: Done

Description

As a user, I want to be able to create a personal account, to be able to have individual and personal interactions with the app.

Requirements

Functional:

- A User with password and email needs to be saved.

Non-functional*:*

- There needs to exist some sort of persistence solution.

Availability:

- No availability requirements.

***Security*:**

- No security requirements

Login

Story Identifier: US006
Story Name: Login
Status: Done

Description

As a user, I want to be able to log into an account so that I can interact with my personal information.

Requirements

Functional:

- Can I see information about my personal account?
- Can I log in now and remain logged in later/get logged in when I start the app?

Non-functional*:*

- There are no non functional requirements.

Availability:

- Can I access my account information from the home page?

***Security*:**

- Are other people prevented from accessing my account when I am not online?

Detailed match view

Story Identifier: US007
Story Name: Detailed match view
Status: Done

Description

As a user, I want to be able to see a detailed view of a match so that I can get more information about the match.

Requirements

Functional:

- Can I open a detailed view of a match?

Non-functional*:*

- Are players sorted by winners or losers?

Availability:

- There are no availability requirements for this user story.

***Security*:**

- There are no security requirements for this user story.

Match History

Story Identifier: US008
Story Name: Match History
Status: Done

Description

As a user, I want to be able to see my match history so that I can see what I have played recently.

Requirements

Functional:

- Can I see if I won or lost the match?
- Can I see when the match was played?
- Can I see how many players were in the match?
- Can other people that were in the match see it?

Non-functional*:*

- Is the information displayed in an “easy to digest” way?

Availability:

- No availability requirements.

***Security*:**

- No security requirements.

Compare Statistics

Story Identifier: US009
Story Name: Compare statistics
Status: Not done (Will not be finished)

Description

As a user, I want to be able to compare my statistics with my friends so that I can see who's better.

Requirements

Functional:

- Can I click on a friend to see my statistics compared to them over all games played together?
- Can I see specific game statistics for games we played together?

Non-functional*:*

- No non functional requirements.

Availability:

- No availability requirements.

***Security*:**

- No security requirements.

Friends

Story Identifier: US010
Story Name: Friends
Status: Done

Description

As a user, I want to be able to see my friends and add more of them so that I can easily select them when adding new matches.

Requirements

Functional:

- See all friends of a user in a list.
- Add more friends to my friends list.
- Search friends both when adding new friends and existing friends.

Non-functional*:*

- Only show non friends when adding new friends.

Availability:

- No availability requirements.

***Security*:**

- No security requirements.

Profile Page

Story Identifier: US011
Story Name: Profile Page
Status: Done

Description

As a user, I want to see my profile with statistics of the games I've played so that I can see if I'm improving.

Requirements

Functional:

- Overall statistics on the profile.
- Match history for the player.

Non-functional*:*

- No non functional requirements.

Availability:

- No availability requirements.

***Security*:**

- No security requirements.

Add new games

Story Identifier: US012
Story Name: Add new games
Status: Will not be started.

Description

As a user, I want to be able to add new board games to the application that previously was not there.

Requirements

Functional:

- No functional requirements.

Non-functional*:*

- No non functional requirements.

Availability:

- No availability requirements.

***Security*:**

- No security requirements.

Achievements

Story Identifier: US013
Story Name: Achievements
Status: Will not be started.

Description

As a user, I want to get achievements from the app to reward me for my accomplishments.

Requirements

Functional:

- No functional requirements.

Non-functional*:*

- No non functional requirements.

Availability:

- No availability requirements.

Security**:**

- No security requirements.

Chat

Story Identifier: US014
Story Name: Chat
Status: Will not be started.

Description

As a user, I want to be able to chat with my friends so that we can play future board games.

Requirements

Functional:

- No functional requirements.

Non-functional*:*

- No non functional requirements.

Availability:

- No availability requirements.

***Security*:**

No security requirements.

Friends profile pages

Story Identifier: US015
Story Name: Friends profile pages
Status: Done.

Description

As a user, I want to be able to see what my friends have played so that I can get ideas for new games.

Requirements

Functional:

- No functional requirements.

Non-functional*:*

- No non functional requirements.

Availability:

- No availability requirements.

Security**:**

- No security requirements.

Home feed matches from friends

Story Identifier: US016
Story Name: Home feed matches from friends
Status: Will not be finished

Description

As a user, I want to be able to see my friends recently played matches in the home feed, so that I can stay up to date with what my friends are playing.

Requirements

Functional:

- My friends games are put in the home feed when they are added.

Non-functional*:*

- No non functional requirements.

Availability:

- I want to see my friends matches every hour of the day.

***Security*:**

- No security requirements.

2.2 Definition of Done

- The user story has been graded as done in a meeting with the group.
- The user story should have an identifiable impact on the application.
- The user story should be tested a satisfiable amount and be without known bugs.

2.3 User Interface

Boardbook primarily contains three different activities: the Account Manger, the Home Page and the Match Wizard. Since the application is based around a specific user the first activity to start is the account manager.

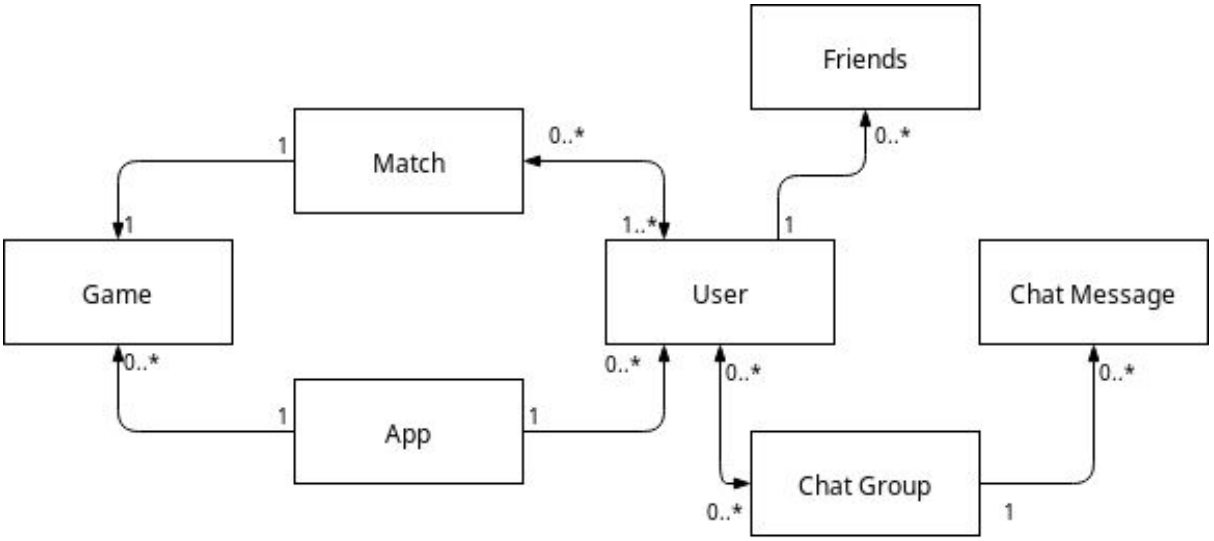
The account manager contains two different fragments that can be switched between, one for login and one for registration. Here, the user can either make a new account or log in with an existing one. The home page is then started.

The home page is the primary screen of the application and has most of the functionality. It has some static elements in the form of a toolbar and a bottom navigation bar. It also has a different fragments that can be switched in between with the usage of the bottom bar. These fragments display information such your recently played matches, your friends and all board games that are supported by Boardbook. The bottom bar also has a button for starting the match wizard.

The wizard is a series of fragments that guide the user through adding a new match. These fragments lets the user select all the necessary information about a match and then add it to their account.

Furthermore, a profile screen can be started by clicking on a button in the top toolbar of the home page or clicking on a friend on the friends page. The profile screen is modular and can be used both for the users profile and used to display other users, for example the profile of friends. The profile page displays some base statistics, and shows the latest played matches that the user has played.

3. Domain Model



3.1 Class responsibilities

Game

A game represents a board game, for example Monopoly. This object groups matches by different games and allows games to have different roles/teams.

App

Central point for the code.

User

User is a user of the application. There can be many users, but only one logged user per app. Users can have friends, and user can have played matches.

Friends

A user can have zero to many friends.

Match

A match is a match played of a specific game. It hold the users that played the match and it stores what game it was. It also binds users to roles.

Chat Group

A chat group that hold chat messages and users.

Chat Message

A message in a chat group.

4. References

Firebase: https://firebase.google.com/
Android Studio: https://developer.android.com/studio
Android: https://developer.android.com/

System design document for Boardbook

Aaron Sandgren, Alexander Ohlin, Carl Holmberg, Dino Pasalic, Rasmus Rosengren

2019-10-25
Final Version

1 Introduction

This document will describe the Android application Boardbook by explaining its design and structure. This explanation will include the architecture of the program and the different components that interact within in. Furthermore, it will both show and talk about the design of the application and the design patterns that have been implemented. It will also give a basic rundown on how a user would from start to finish interact with the application. Lastly, the document will detail how the application tests itself and the account security is handled.

Boardbook is an Android application where users can enter results of matches played for supported board games. This allows the application to generate statistics such as overall board game win rate or win rate for specific games. Boardbook will also allow the user to add friends and view their matches as well. Boardbook will be able to show what game was played, roles if there are any and if the person won or lost said match. The user will also be able to get achievements by fulfilling certain criteria like playing ten board games and so on. In order to make Boardbook an encouraging platform for board games, the application will allow users to chat with one another.

1.1 Definitions, acronyms, and abbreviations

MVP: Model-View-Presenter
SDD: System Design Document
Activity: A specific screen in the user interface
Fragment: A subsection of an activity, used for some specific functionality

2 System architecture

Top level design

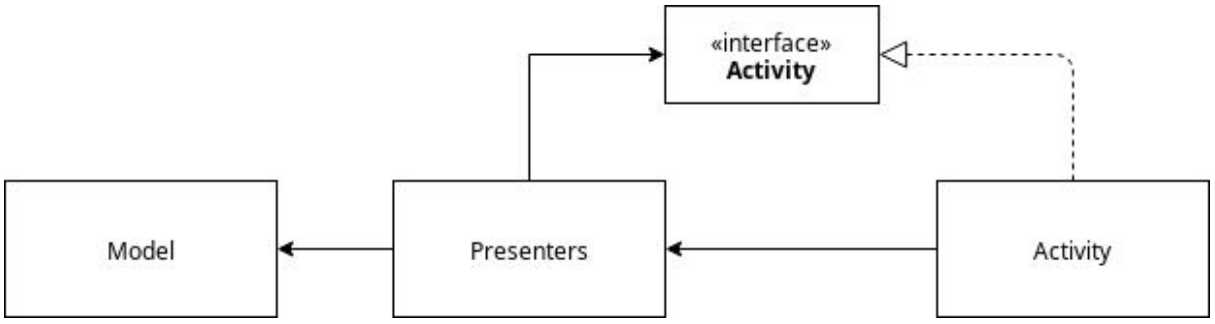
Boardbook only has one component, the application itself. It manages every feature that the application is able to perform.

Flow

When the user first start the application they will be prompted to either log into Boardbook or create a new account. Whichever option they choose will then take the user to the home screen. Here they will be presented with matches that they have previously added to their account or matches other people have added that include their account. It is also here that the user can find the “Add match” button which can be used to add the presumably recently played match of a board game to the users account. The user will then have to select what board game was played, choose the players who played the match, divy the players to potential different teams or role in the boardgame and finally choose which players won or lost. This process exemplifies the most common interaction with the application.

3 System design

Top level:



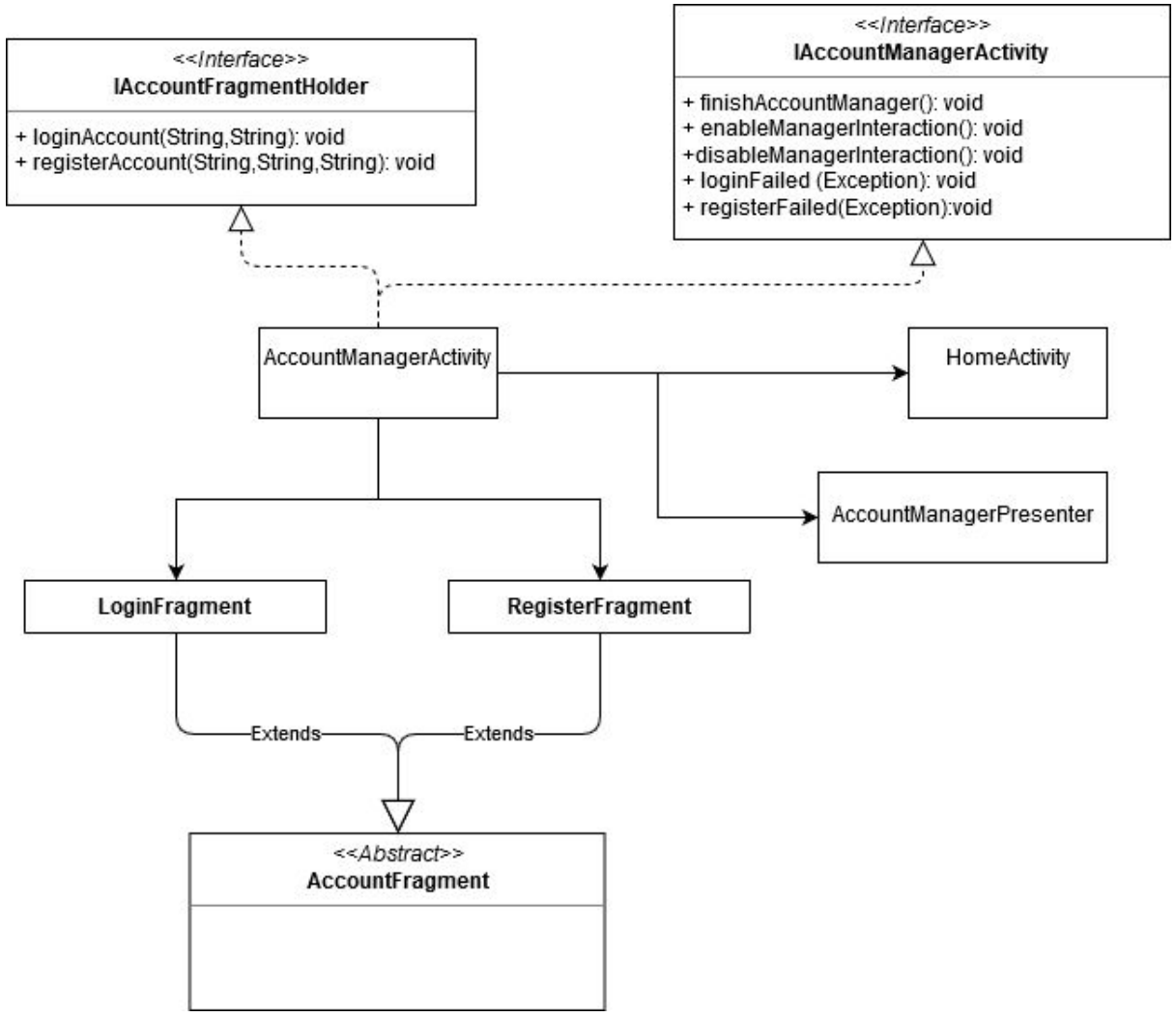
Boardbook has been developed using the MVP design architecture in mind (See image above). In this application, the responsibilities goes as following.

Activities and their fragments are dumb views with as little logic as possible. The activities delegate user events to presenters, classes in charge of communication between the model and the view.

Presenters are created by their respective views and respond to their events. They then interact with the model and then tell the activities how to respond.

The model is in charge of handling and storing data of games, matches, users and other entities in the application. It does this through the usage of various data handlers whom then in turn interact with repositories. The repositories interact with a database and require a specific implementation, which in our app is done through the usage of Google firebase.

Authentication:

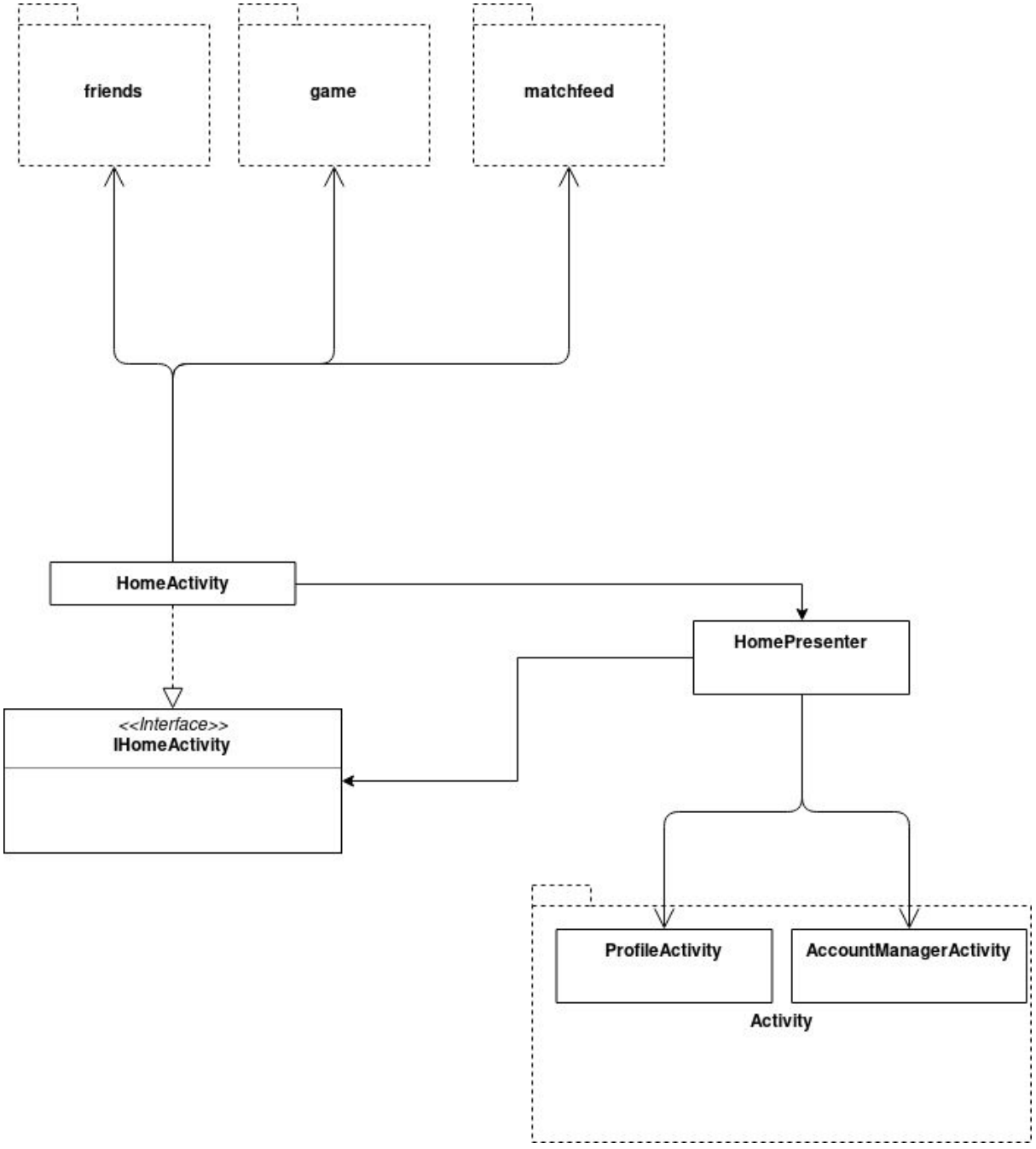


The authentication page consists of one activity, the AccountManagerActivity. The activity has a AccountManagerPresenter which it delegates user input to. It also has two fragments that handle registry and login of accounts respectively. Additionally, the activity implements IAccountFragmentHolder and IAccountManagerActivity. The fragment holder interface defines communication between the activity and its fragments. Meanwhile, the IAccountManagerActivity interface defines methods required to interact with the presenter of the activity.

Both fragments of AccountManagerActivity extend the class AccountFragment which contains behaviour shared by both fragments, such as a reference to their parent activity. When a user clicks on the register or login buttons the fragments send the information in their input boxes to the activity. The activity then informs its it's AccountManagerPresenter to either login or register with the information it receives from the fragments.

The AccountManagerPresenter interacts with the backend and tries to either log in or register a new account. If successful it will instruct the activity that it should switch to the HomeActivity and afterwards close itself.

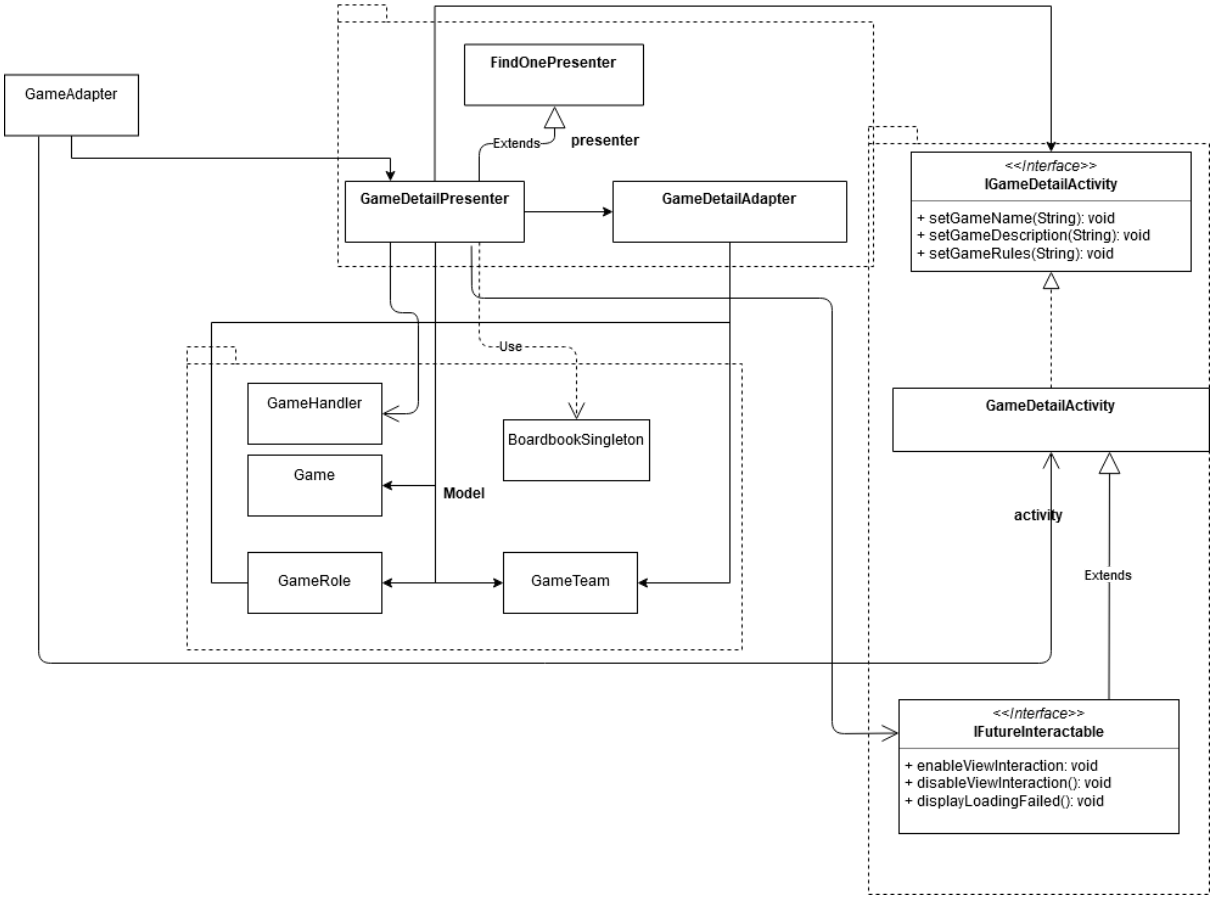
HomeActivity:



The home activity is simplest activity in the application. After authentication, this activity starts. It contains a top bar, a bottom navigation bar and a container for fragments. The top bar is very simple and only contains a drop down menu where

the user can view their profile or log out of the app. The bottom bar contains buttons for starting fragments in the container. When a user clicks on a bottom button a new instance of that respective fragment will be created a put in the container. The (implemented) fragments a user can start are Matchfeed, Games and Friends. By default, the fragment that is in the container is Matchfeed fragment.

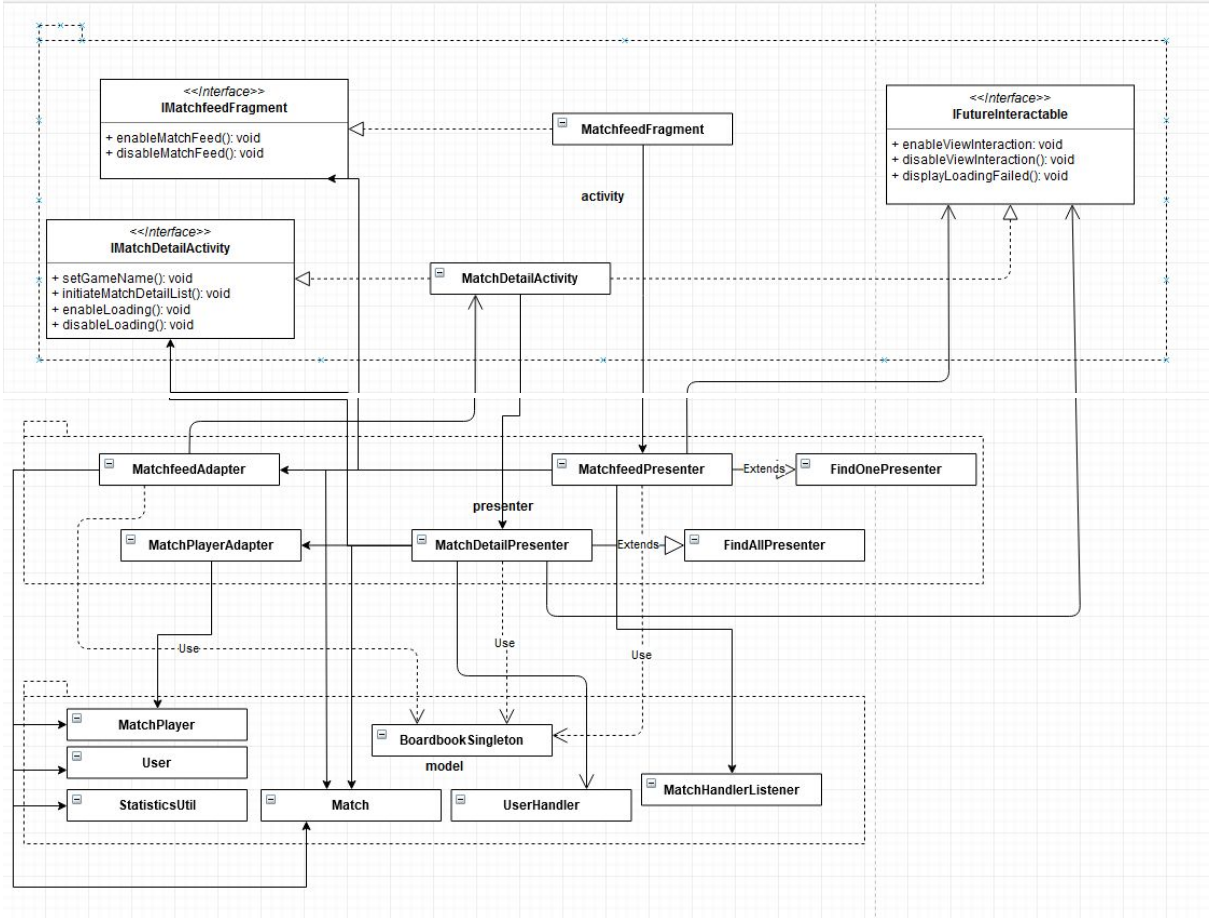
GameDetailActivity:



The GameDetailActivity describes the page shown when the user clicks on a game in the game feed. The page shown is managed by GameDetailActivity which is a dumb class with very limited functionallity. When the activity is created it creates a GameDetailPresenter with itself as the argument to enable the presenter to interact with the view.

The GameDetailPresenter handles managing the activity and communication with the model and database. It inherits logic from FindOnePresenter which it uses to find the game to be displayed. When found, it makes an GameDetailAdapter which translates the game to a view object that can be displayed on the screen. It then tells it’s activity to display said view object.

MatchFeed Fragment + MatchDetailActivity:



This above diagram describes the system and pages that manage the match feed and the match detail views. Match Feed displays a list of matches. When one of these matches is pressed, the match detail view is opened.

MatchFeed:

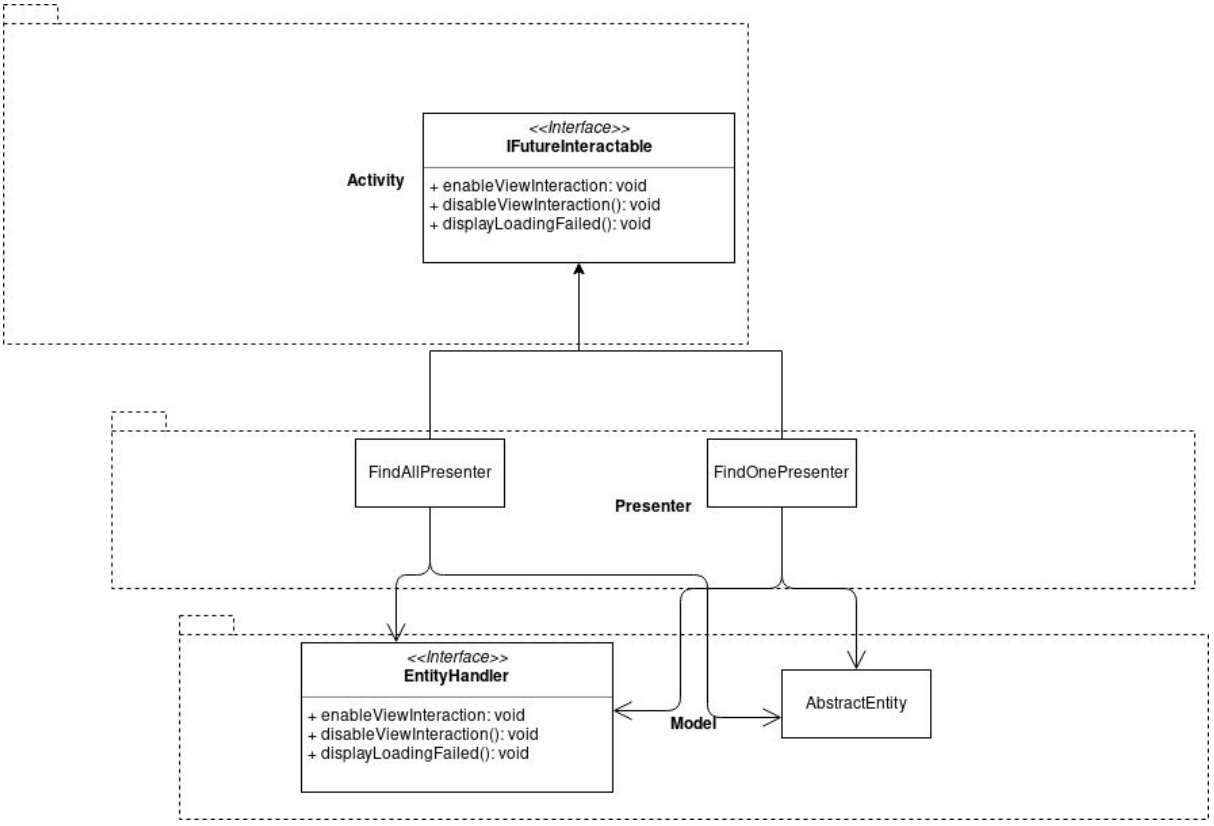
HomeActivity creates a MatchFeedFragment. The fragment then creates a MatchFeedPresenter which in turn creates a MatchFeedAdapter. MatchFeedPresenter also subscribes to MatchHandlerListener. MatchFeedFragment implements the IMatchFeedFragment interface. It binds the MatchFeedAdapter to the view. MatchFeedPresenter creates an adapter to convert a list of matches to a view object. It extends the FindOnePresenter. It uses the boardbook singleton to get data from the model.

MatchDetail:

MatchDetailActivity implements the IMatchDetailActivity interface. It also implements the IFutureInteractable, which makes the view able to interact with futures from the

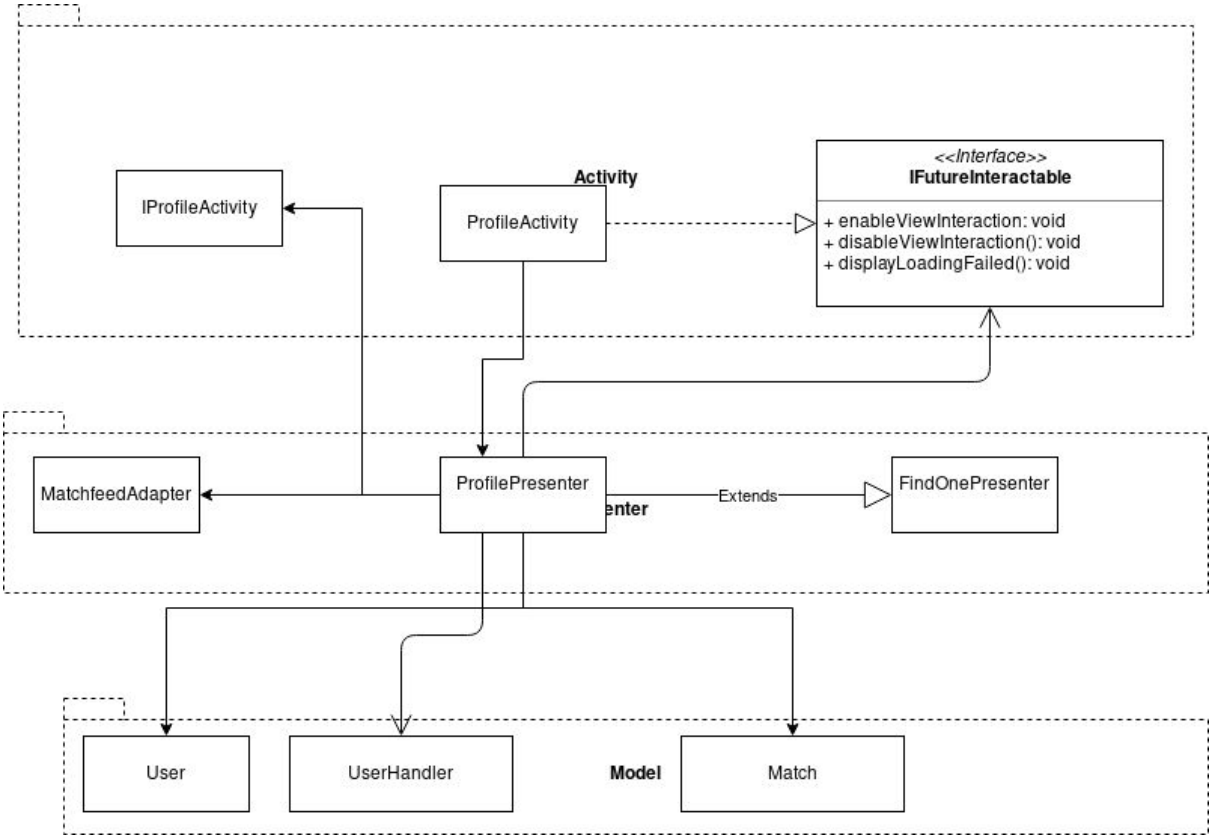
backend. MatchDetailPresenter holds a reference to a IMatchDetailActivity. It also extends the FindAllPresenter. The presenter interacts with the model via the user handler to get all the users in the specified match and listing them via the MatchPlayerAdapter.

AbstractPresenter:



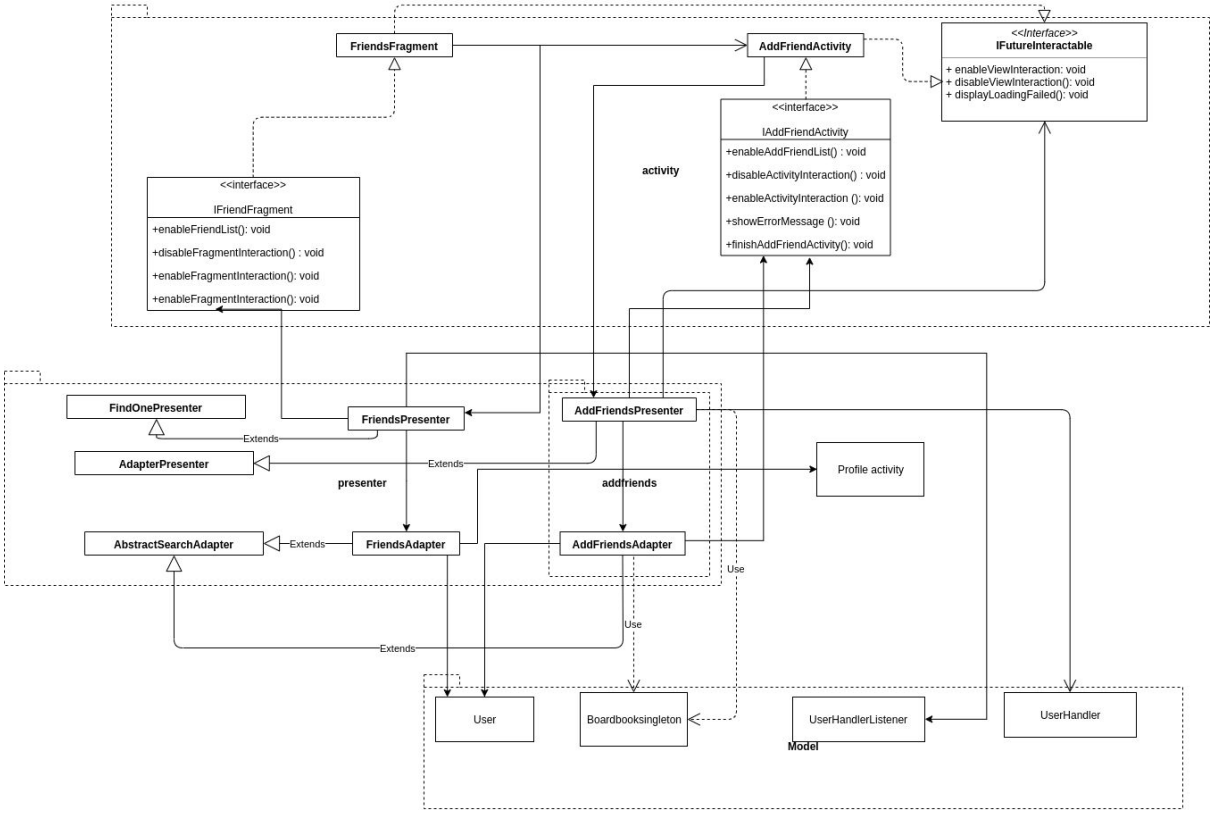
The classes FindOnePresenter and FindAllPresenter generalizes behaviour that all the other presenters use, namely to find one or all of a specific entity in the database. They are both abstract classes and are inherited if finding one or many entities is required. On creation, both take a AbstractEntity and EntityHandler as type parameter. This enables them to search for any type of Entity via the EntityHandler. Both presenters have an IFutureInteractable and use it to enable loading while searching.

ProfileActivity:



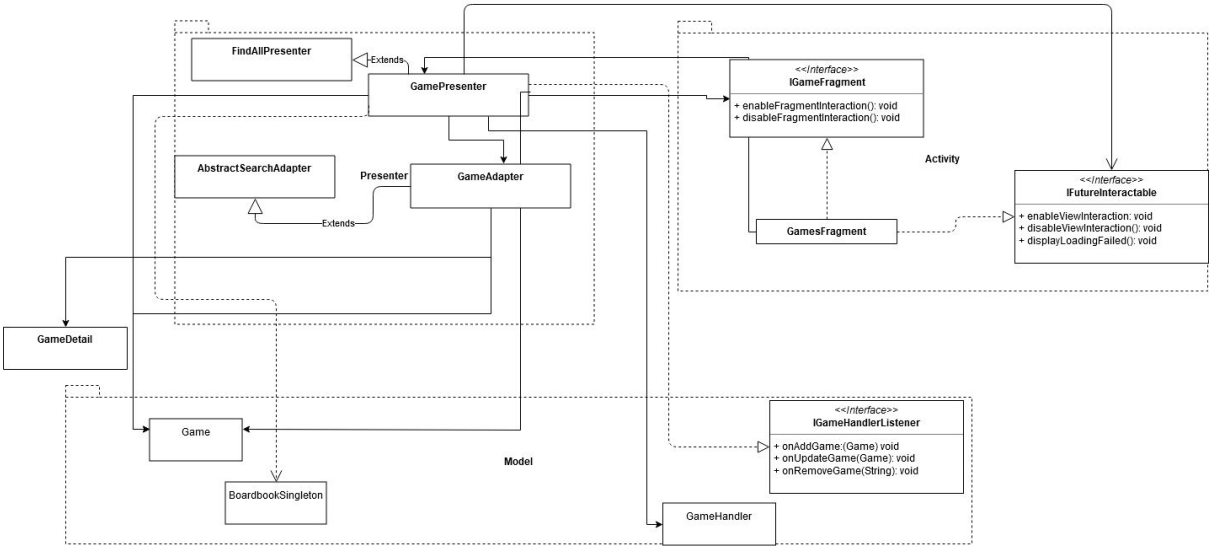
This diagram shows the profile screen of the app which you can access by pressing the icon in the top right corner, alternatively finding a specific users whose profile you want to access. In the profile screen, you can see win percentages and play records of a specific user. The profile activity is the class that takes care of showing the visuals with the help of the presenter to have proper logic. The presenter makes use of the User, UserHandler and Match classes in Model as its all the information it needs.

AddFriends + Friends:



In the homescreen when the friends icon is pressed the app changes the fragment from whatever else is active at that moment to FriendsFragment. The FriendsFragment utilizes the FriendsPresenter and FriendsAdapter to display the current user's friends. Then the user can press the “add friend button” and a new AddFriendActivity is started where the user can search for and add other users as friends. The activity uses its Presenter and Adapter to display users and take input from the user. The activity also implements IFutureInteractable to enable and disable its inputs while getting data from the Repositories inside of BoardBookSingleton.

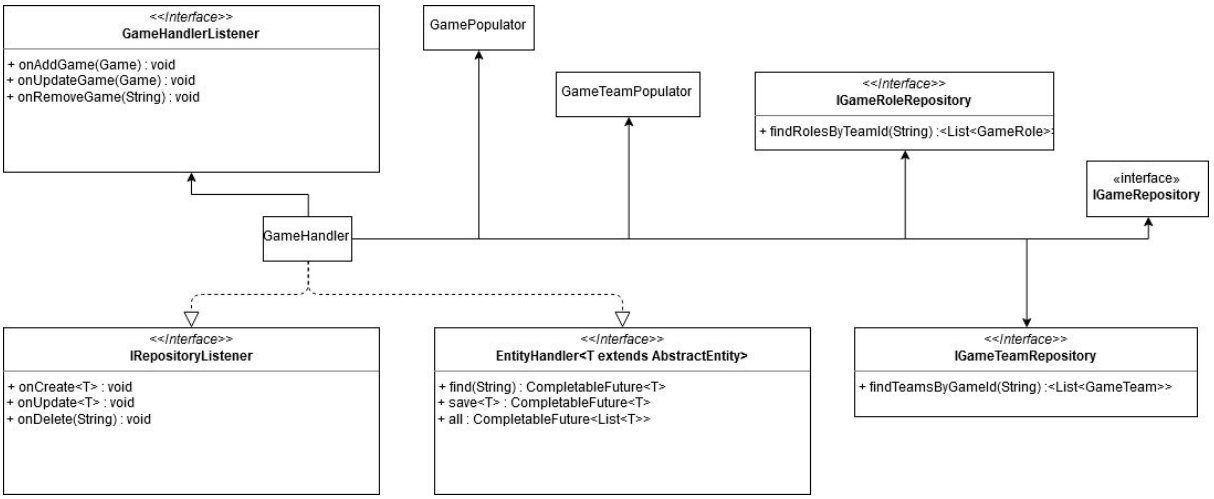
Game Fragment:



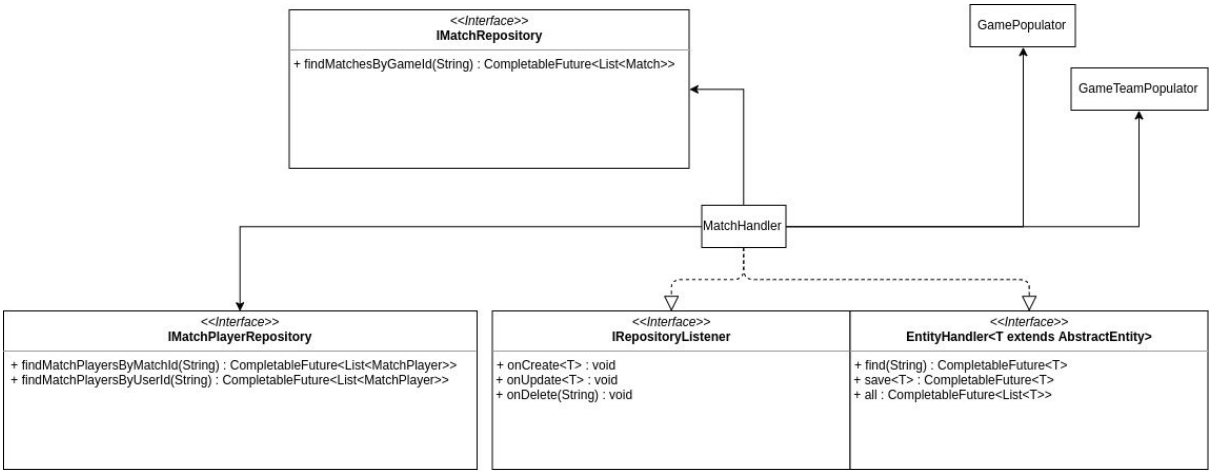
This is a diagram that shows the system that shows and handles games. GamesFragment implements the IGamesFragment interface. It also implements the IFutureInteractable interface to be able to interact with futures from the model. The Game fragment interacts with the GamePresenter. The GamePresenter talks to the model and turns data from the model into view elements that the activity can serve the user. It turns the data from the model into a view element by using the GameAdapter. The GameAdapter extends the AbstractSearchAdapter which has functionality for filtering and searching in the adapters games. The presenter extends the FindAllPresenter class to inherit searching for all games in the database.

Handlers:

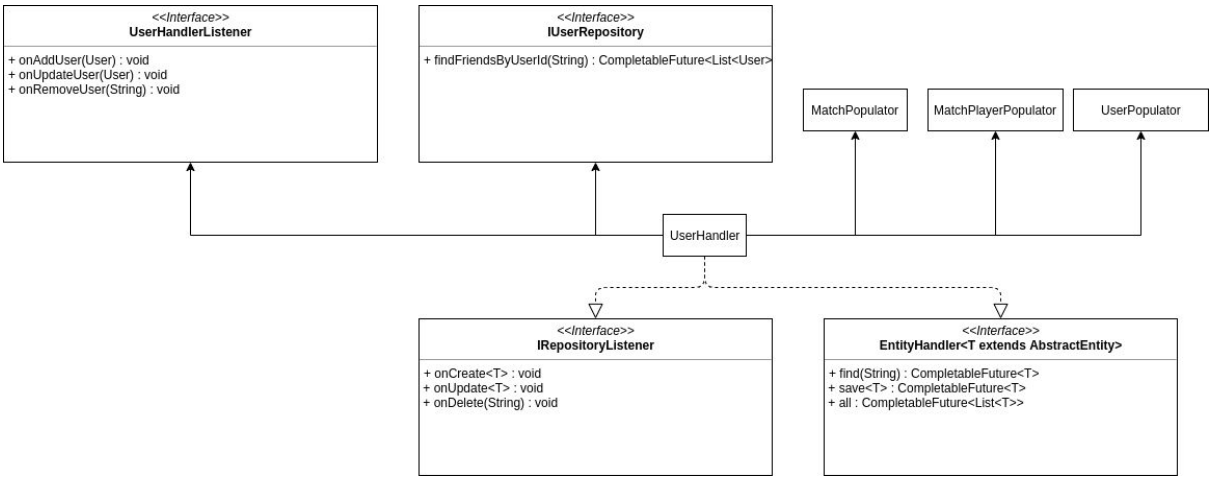
The following three diagrams show the structure of the different entity handlers. They all have a very similar structures, only differing in their dependencies. The purpose of the handlers are to provide a good way of fetching the relations of the entity (games, matches, and users), to reduce code duplication throughout the codebase. Each handler has a dependency on one or more repository interfaces (DIP). It also has some populators, each populator's purpose is to fetch all the relations for the given entity type, e.g. *MatchPopulator* fetches *MatchPlayer*s and the *Game* related to a *Match*, so that the remote data is properly synchronized to the local device. Each handler also implements a repository listener of the same entity type, so that each handler can listen to the changes that happen to the repository.



A UML diagram showing the dependencies of the GameHandler class

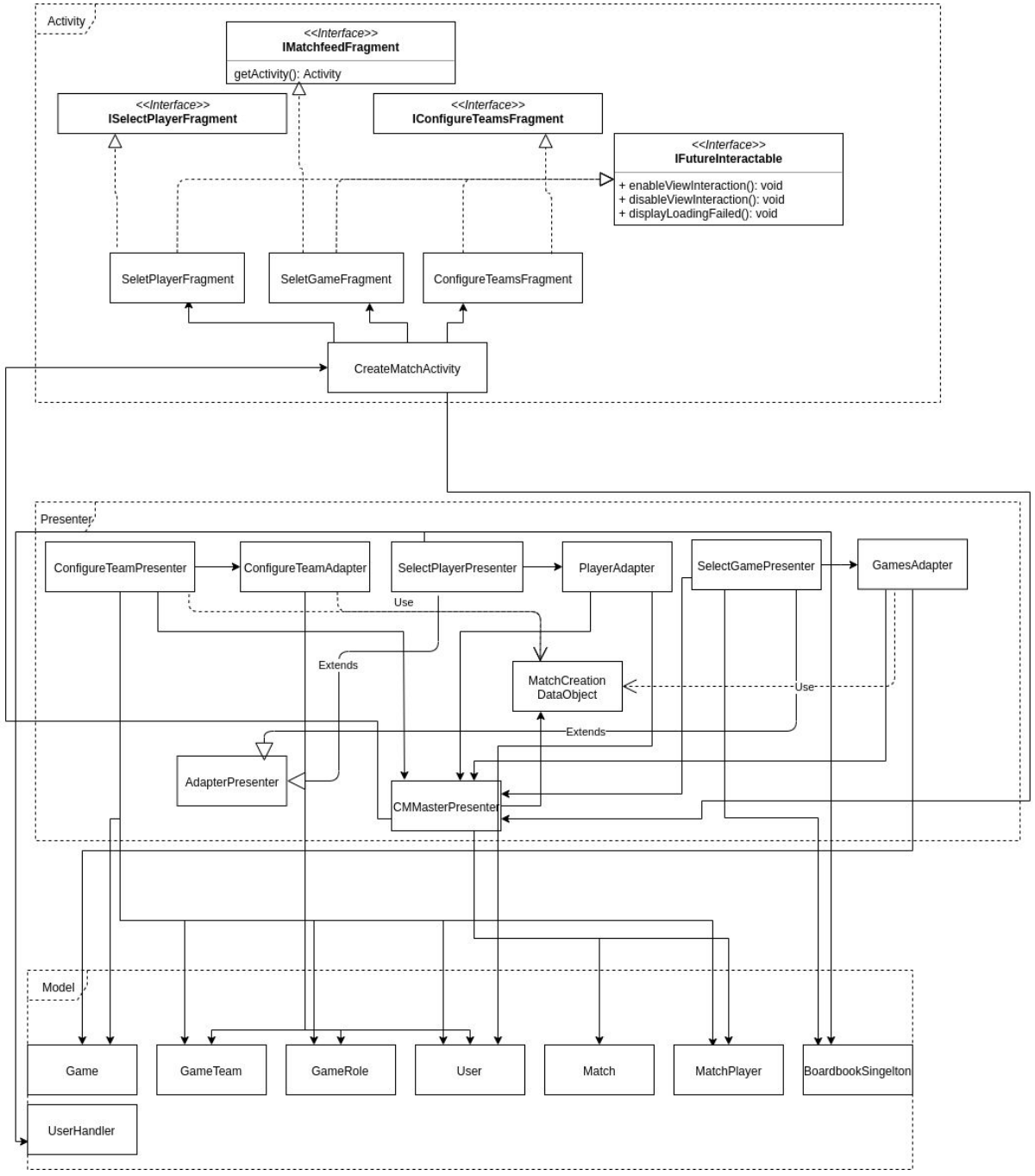


A UML diagram showing the dependencies of the MatchHandler class



A UML diagram showing the dependencies of the UserHandler class

Creating Matches:



This diagram shows how Match Creation is built up. CreateMatchActivity is the central point which contains all the Fragment and servers the Fragments with their presenters. The Activity itself contains a MasterPresenter which holds logic which is required by all the Fragments, like switching Fragments. The Fragments themselves only hold the instance of their own Presenter provided to them by the MasterPresenter and instantiates/binds GUI objects.

All logic is in turn inside of the Presenters and their respective Adapters. These are the Classes that use and interacts with the model. The Presenter also holds onto a

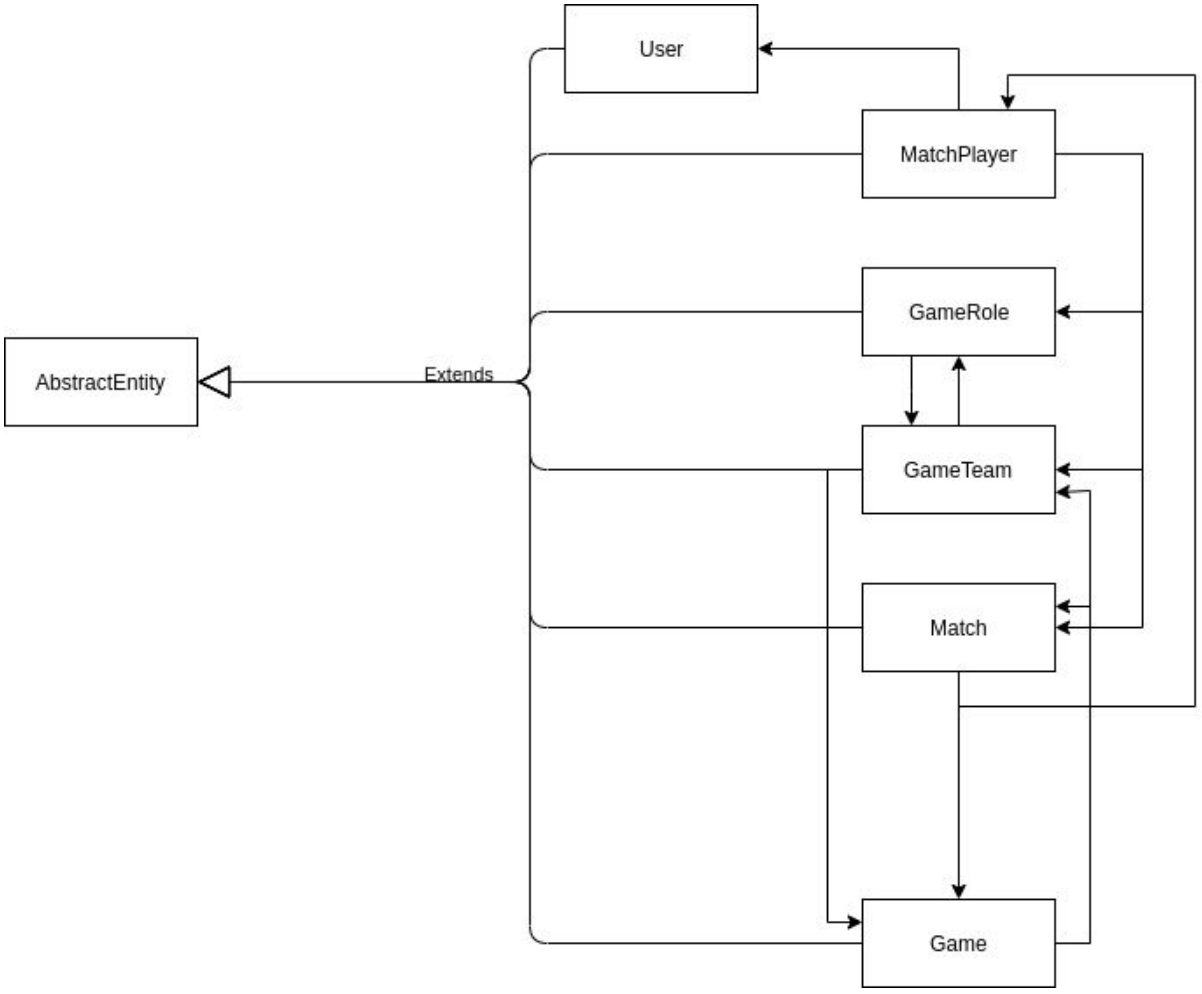
reference to the MasterPresenter.

The MatchCreationDataObject is held onto by the MasterPresenter and is used as a way to store the data provided by the user throughout the process of creating a match. It only holds onto data and provides no further functionality.

When the process is done the MasterPresenter gets the data from the DataObject and saves the match.

Entity:

The following diagram shows how the different entities of the model are related to each other. They all extend a base abstract entity class, which contain information that all entities much have, such as a unique identifier.



4 Persistent data management

Boardbook uses firebase to store all its data. Therefore, all necessary data is requested when the application starts and no data is stored on the users device.

The model is completely abstracted from the database by using the Repository pattern. This ensures that the database could completely be swapped out for another storage solution. There are currently two implementations of the repositories, the firebase version and the mock version primarily used for testing.

5 Quality

Code Quality

Firebase can be slow on initial load. The speed has been improved by limiting cascading loads and caching loaded data and relationships, but it is still slower than we would like it to be.

Sometimes people are shown as added to a match when they are actually not added. This is a visual bug and we can't find a solution to this problem before the deadline.

Testing

For testing we use unit tests. They are located in the test directory in the project. These are run with travis ci to ensure that the code will build and that the code is tested.

5.1 Access control and security

Boardbook enables users to create their own account with an email and a password. The application uses the authentication service of Firebase in order to save and compare this data when it is necessary. Therefore, Boardbook does not handle any of its own login authentication.

6 References

Firebase: https://firebase.google.com/
Android Studio: https://developer.android.com/studio
Android: https://developer.android.com/

Peer Review - Group 6

- **Do the design and implementation follow design principles?**

  From our perspective, there exists no actually references between different data objects. For example, there is a movieGenreJoin class that joins ids but there is no actually references between the objects. If the data holders are the actual model implementation then the model is not entirely object orientated since no actual references between classes exist.

  - **Does the project use a consistent coding style?**

    - It's very consistent.

  - **Is the code reusable?**

    - Because much of the code and design is specifically built to interact with it's libraries it becomes difficult to reuse the code for components not built for these libraries.

  - **Is it easy to maintain?**

    - It's difficult as it depends on a lot of libraries which could become unsupported with future versions of java. If the libraries becomes unsupported is out of the developers control.

  - **Can we easily add/remove functionality?**

Because the code is highly dependant on it's libraries it will become very difficult to add new functionality that is not supported by the libraries. Furthermore, if such functionality is needed then new libraries with said functionallity will need to be implemented and large parts of the code would need to be refactored.

On the other hand, functionallity that is supported by used libraries will be much easier to implement since the libraries already do a lot of work for by themselves.

- **Are design patterns used?**

  - Yes

- **Is the code documented?**

  - There are some very helpful comments but some of them are kind of confusing
    :
    CastAdapter row 29 "// Inflate our RecyclerView with the items it shall contain?"

- **Are proper names used?**

- No.
  Interfaces names vary a lot.
  Classes have confusing and unclear names.

- **Is the design modular? Are there any unnecessary dependencies?**

  - Since we are not quite sure how the code works we cannot tell what dependencies are not necessary

- **Does the code use proper abstractions?**

  - They could use more abstractions in their code

- **Is the code well tested?**

  - They don’t seem to run which means we cant see if they are well implemented

- **Are there any security problems, are there any performance issues?**
  - Can’t comment on security issues. There was no glaring issues with the performance

- **Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?**

- Its isolated in a good way although some parts of the code are unclear with odd names
- There is not a clear “model” folder which makes it difficult for us to understand what is a part of the model.
- The code depends on a lot of other libraries which makes it difficult to understand.
- Frameworks makes this code impossible to give feedback on without understandning the frameworks. We do not have the time to read what all these frameworks does, so we cant do a proper peer review.

- **Can the design or code be improved? Are there better solutions?**

  - Yes, the main problem that we find is the lack of a concrete model implementation. The current model that exists implicity depends on external libraries which makes it hard to both understand and to separate from the rest of the application.