TDA367: Object-oriented programming project
Chalmers University of Technology

Grey Matter
RAD-documentation

and I OOPP:
ALINE EIKELAND
HUGO GANELIUS
VIKTOR JOHANNESSON
EMIL LUNDQVIST
FELIX OLIV

1 Introduction

1.1 Purpose

The project aims to construct an Android application for mental training. The application is to be competitive and rank users in relation to others as well as own past performances.

With an exploding population getting smartphone access the mobile app market is, if not brand new and unproblematic, still growing and developing. Strong influences seeking more meaningful usage of their devices can be attracted to a scientific, low weight and healthy game app.

1.2 Application specifications

The application is to be an android application compatible with android 8.0 and upwards. It will be adaptable for various screen sizes.

The application will work as a hub for multiple mental training mini games. These games can be reached from a common menu. Common for the games are also a leaderboard which ranks players internationally and amongst their friends as well as a personal progress analyzation tool. Both the leaderboards and the analyzer tool will be scoring via the Neurån scoring system. This competetive point system will have a scientific aspect as it is based on the normal distribution of the player base.

The user will be able to add other users as their friends. They can show their friends in a list and is able to remove or block existing as well as search for new ones.

Additionally the app will be able to notify the user when they are passed on the friend leaderboard, when they have not used the application for a specific amount of time and The notifications can be modified by the user. The user will also be able to modify their username and picture to be shown at the international leaderboards.

1.3 Scope of application

The application will include at least four mental training mini games. It will host an international leader board and a friend leaderboard for each of the games. *Grey Matter* will store the users player history in a personal profile which can add other profiles as their friends.

1.4 Definitions, acronyms and abbrevations

Gamification - applying gameplay principles on non-gaming activity to increase attractiveness
JSON - JavaScript Object Notification, easily readible file format for data storage

JSON-server - JSON module to easily mockup servers without having to program server part.
GSON - Google´s JSON server service.
MVC - Model View Controller pattern.
MVVC - Model View ViewModel pattern.
Normal distribution - Natural probability calculation, looks like a bell curve.
Standalone application - A application that runs locally on the device and does not require anything else to be functional.
Game/test - a short activity that challenges a player in some way, rewarding a score based on performance.
Neurån - a scoring system as well as a score unit. Based on normal distribution.

2 Requirements

2.1 User Stories

> **US01 Score History**
> As an esports professional, I want to be able to see my score history, to know how well I've done previously.

Confirmed by following functional requirements:

- Can I view my history for different games separately?
- Can I see my highest score in each game?

And by nonfunctional below:

- Availability
  - Can I view and update my score history without an internet connection?
- Security
  - Are unauthorised people prevented from viewing my score history?

This user story is not yet fully satisfied.

> **US02 Score Graph**
> As a e-sports professional, I want to be able to see a graph of my score over time, to see if I'm improving.

Confirmed by following functional requirements:

- Can I select time period to be drawn on the graph?
- Does the graph show one value for each day played?

And by nonfunctional below:

- Availability
  - Can I select if the graph shows daily averages or daily top scores?
- Security
  - Are unauthorised people prevented from viewing my graphs?

This user story is not yet fully satisfied.

> **US03 Leaderboard**
> As a competetive person, I'd like to see my score in a leaderboard, so I can compare myself to others.

Confirmed by following functional requirements:

- Can I select to view a leaderboard for friends and worldwide separately?
- Can see a leaderboard for each game separately?

And by nonfunctional below:

- Availability
  - Can I view the leaderboard without an internet connection?
- Security
  - Can I choose who can view my scores from worldwide/friends?

This user story is fully satisfied.

> **US04 Score Rank**
> As a competetive person, I want to rank my score against everyone in the world.

Confirmed by following functional requirements:

- Can I see which percentile of all player's scores my own performance reaches?

This user story is not yet fully satisfied.

US05 Daily Reminder
As a senior citizen, I'd like a daily reminder to test myself, so I can keep a regular routine.

Confirmed by following functional requirements:

- Can I choose to receive a daily reminder?

This user story is not yet fully satisfied.

US06 Notification options
As a busy person, I'd like to customize how and when the application notifies me

Confirmed by following functional requirements:

- Can I select which weekdays I receive a notification?
- Can I select which time of the day I get notified?

And by nonfunctional below:

- Security
  - Are unauthorised people prevented from changing my settings?

This user story is not yet fully satisfied.

US07 Short-term Memory
As a user I want to play memory games to improve my short-term memory

Confirmed by following functional requirements:

- Can I play games that work on my short-term memory?
- Can I play memory games on the public transport?

This user story is fully satisfied.

US08 Problem Solving
As an employee I want to improve my problem solving abilities to advance

> my performance as an employee

Confirmed by following functional requirements:

- Can I play puzzle games to improve my problem solving?
- Can I get a score to compare previous results?

This user story is fully satisfied.

2.2 Definition of Done

The following criteria must be met for all user story implementation before they can be considered complete.

- All features should be tested with jUnit tests.
- All features should work as expected by the user, with no major bugs.
- All coded features must be added to version control, in our case git.

2.3 User interface

The first iterations of the GUI:

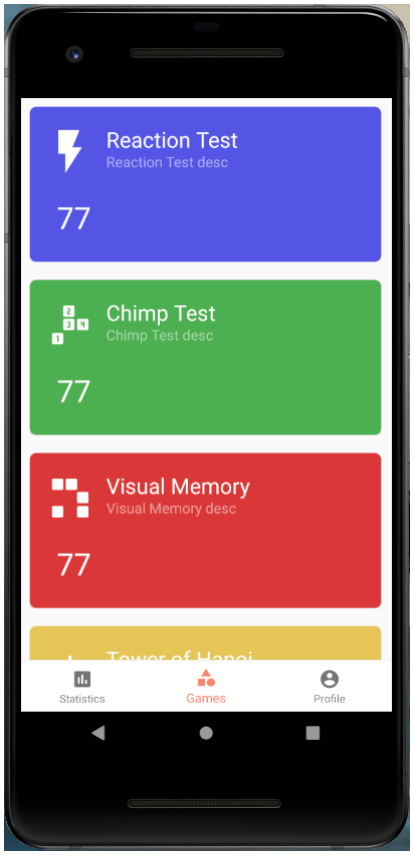

Figure 1: The scrollable list of the games, where the user can press on one of the cards and the corresponding games starting screen comes up as seen in figure 2

Figure 2: The starting screen of the game ”Chimp Test”. It contains some information about the game and a button to start the game with

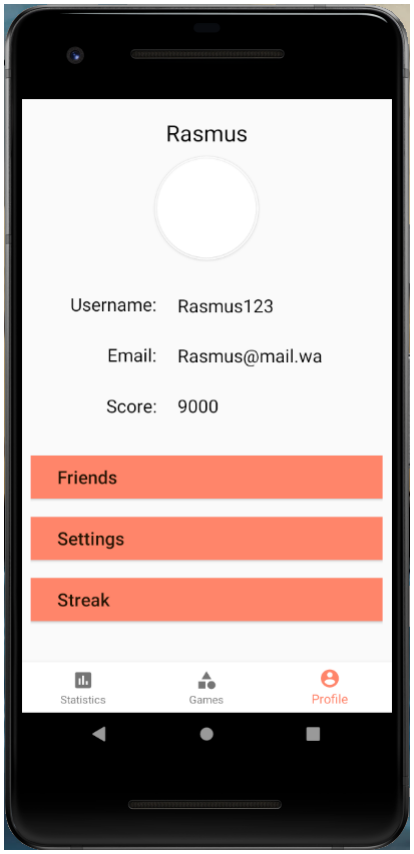

Figure 3: The profile page where we want the user to be able to change some settings and check on their friends

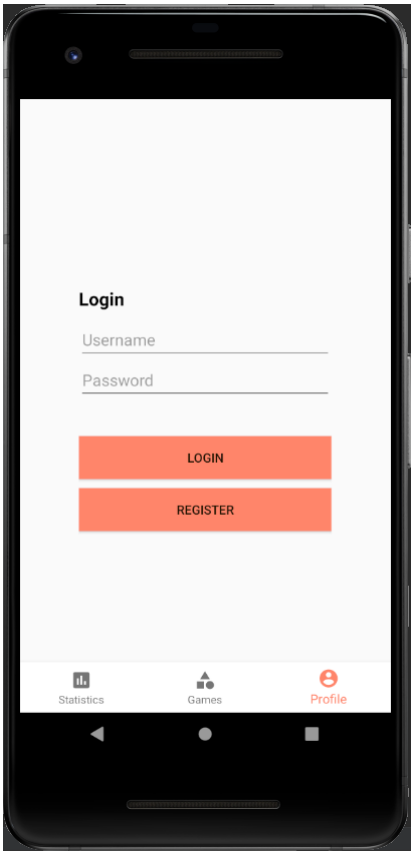

Figure 4: A dialog window that shows the log in and register option

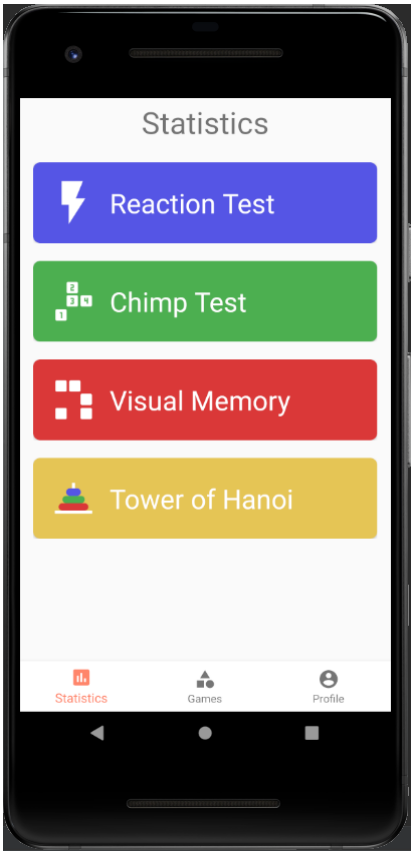

Figure 5: A scrollable list of statistics on different games. When a game is pressed it shows the users score in each game and how it compares to the rest of the world

Figure 1, 3 and 5 are all accessible by the bottom navigation bar. Figure 6 pops up when you press on a game in figure 1. Figure 4 is accessed when you press on profile but you are not logged in.

3 Domain model

Multiplicity - How many instances exist of the object
Relationships - How the classes relate to eachother
Mutability - If the object is able to change/mutate
Persistence - If the object will save the data after execution

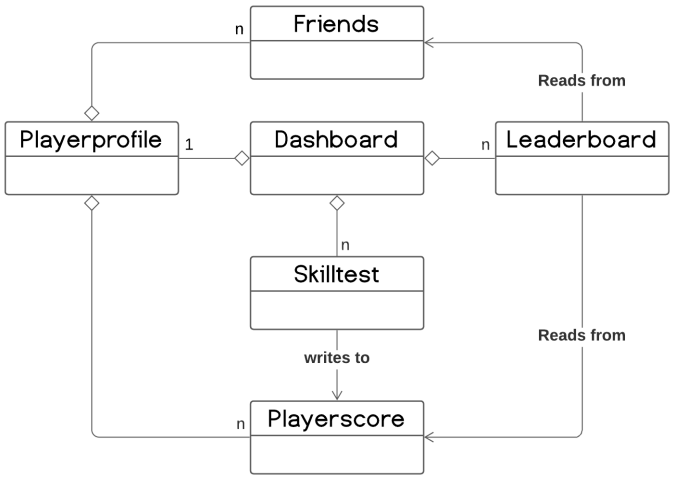

Figure 6

3.1 Class Responsibilities

Dashboard - Represents the first screen the user sees when entering the program, from which they can select games to play and navigate the app. It is not mutable.

Player - The signed in user currently using the app.

Skilltest - A test the player can perform and receive a score on.

Playerscore - Data for the player's previous performances, mutable through the player's actions.

Leaderboard - Shows the user's high score among other users' high scores. It is mutable as users will improve their score.

Friends - other users whom the player can compare their scores to

TDA367: Object-oriented programming project
Chalmers University of Technology

Grey Matter

System Design Document

and I OOPP:
ALINE EIKELAND
HUGO GANELIUS
VIKTOR JOHANNESSON
EMIL LUNDQVIST
FELIX OLIV

1 Introduction

This system design document intends to introduce the reader to the brain test mini game android application that is Grey Matter. The document will develop on design choices and process, system structure and qualities as well as the finished product.

1.1 Definitions, acronyms, and abbreviations

Gamification - applying gameplay principles on non-gaming activity to increase attractiveness
JSON - JavaScript Object Notification, easily readible file format for data storage
JSON-server - JSON module to easily mockup servers without having to program server part.
GSON - Google's JSON server service.
MVC - Model View Controller pattern.
MVVM - Model View ViewModel pattern.
Normal distribution - Natural probability calculation, looks like a bell curve.
Standalone application - A application that runs locally on the device and does not require anything else to be functional.
Viscosity - Refers to how easy it is to add code to the program while maintaining the design.
AVD - Android Virtual Device, simulation of an Android device.
Android - Popular mobile operating system.
JUnit - Unit testing framework
Local data - Information on your local device
Repo - Short for repository, a location to store software packages

STAN - Program that analyses the structure of the code and creates dependency maps.
PMD - Short for Programming Mistake Detector, analyzes the source code and reports issues.
P of EAA - Patterns of Enterprise Application Architecture, a book by Martin Fowler about design patterns.
GUI - Graphical user interface, what the user sees when using the application
RecyclerView - a is a simple way to display large sets of data while minimizing memory usage.
Instance - a concrete occurrence of any object Firebase

2 System architecture

GrayMatter is a mostly standalone application, and all the logic is therefore built into the app. However to be able to compare your score with other players, the app needs a internet connection to download other players data which is located on a server.

Grey Matter application contains all necessary logic and GUI. The social interaction and leaderboard features however requires the need of a database and server. If the player experiences infrequent internet connection, the offline features can still be used although the results will not be updated until the player retains internet access.

2.1 Flow of the application

When the application is started the method on-Create in the Main-activity class is called. This method will instantiate the main-page XML file which displays the main-page to the,

3 System design

3.1 System packages

3.1.1 Social domain

The social

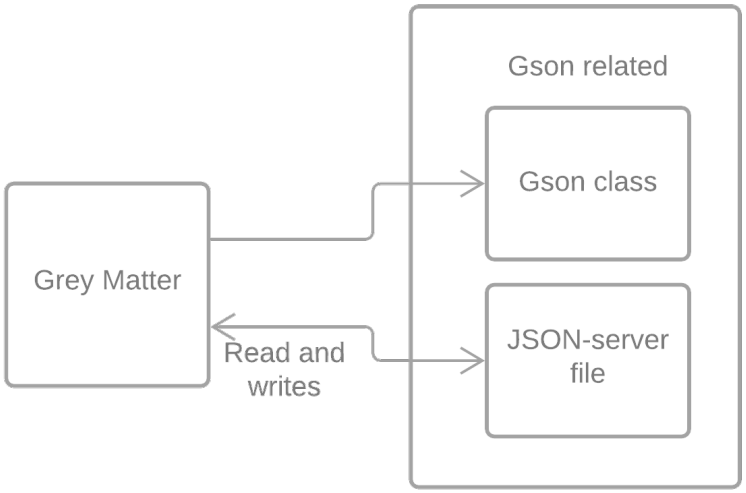

Figure 1: External use of Gson in social package

3.1.2 Datamapper packages

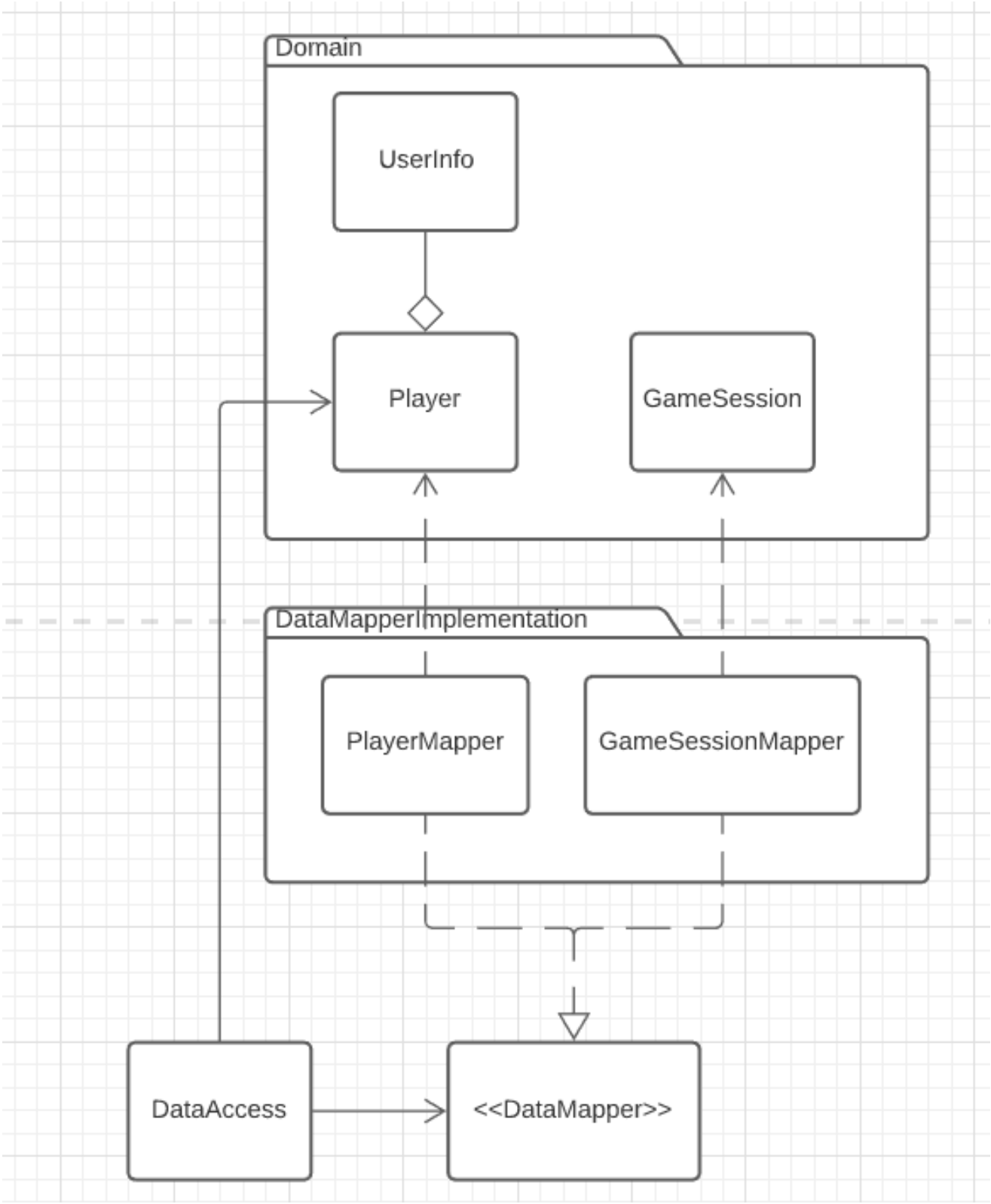

Figure 2: The package model of the application

3.1.3 Progress

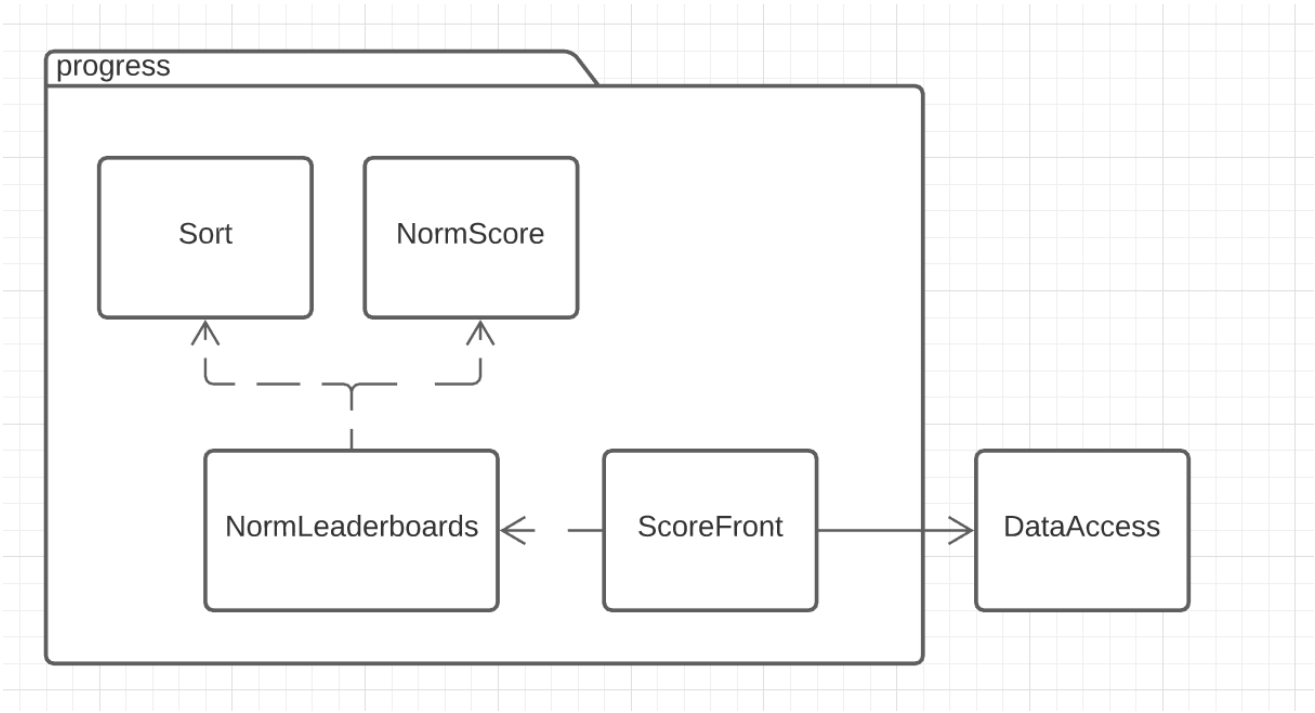

Figure 3: The package model of the application

The structure of the application has been implemented using an architectural design pattern similar to MVC, but instead of the controller we use a viewmodel, called MVVM. Both principles are ways to separate the calculations and data into the Model and just focusing on presenting the data inside the View. The difference between the MVC and the MVVM pattern is that a controller is more of a way to display data from the model to the view. Since we do not need to display data from the model for the user to interact with the game we use a viewmodel that control the communication from the view and model.

The viewmodel keeps track of what the user does with the GUI, and changes the view accordingly, showing the correct screen and data when needed. The view communicates with the viewmodel that something has happened and the viewmodel in turn communicates with the model to access the correct data and then the view can fetch the data from the viewmodel to display it for the user.

For the View we use fragments to display the data, which is like a sub-activity to the application which has its own input events and it is possible to add or remove it during the application run-time. While the application is running you can create and modify each fragment independently, which is why we chose to use them. We have

various screens that use its own fragment and to manipulate them independently is a must for us. Each fragment get added to the stack, which enables the user to use the back button to navigate to the previous fragment, for example if you are in a game fragment you can press the back button and arrive at the scrollable list of all the games.

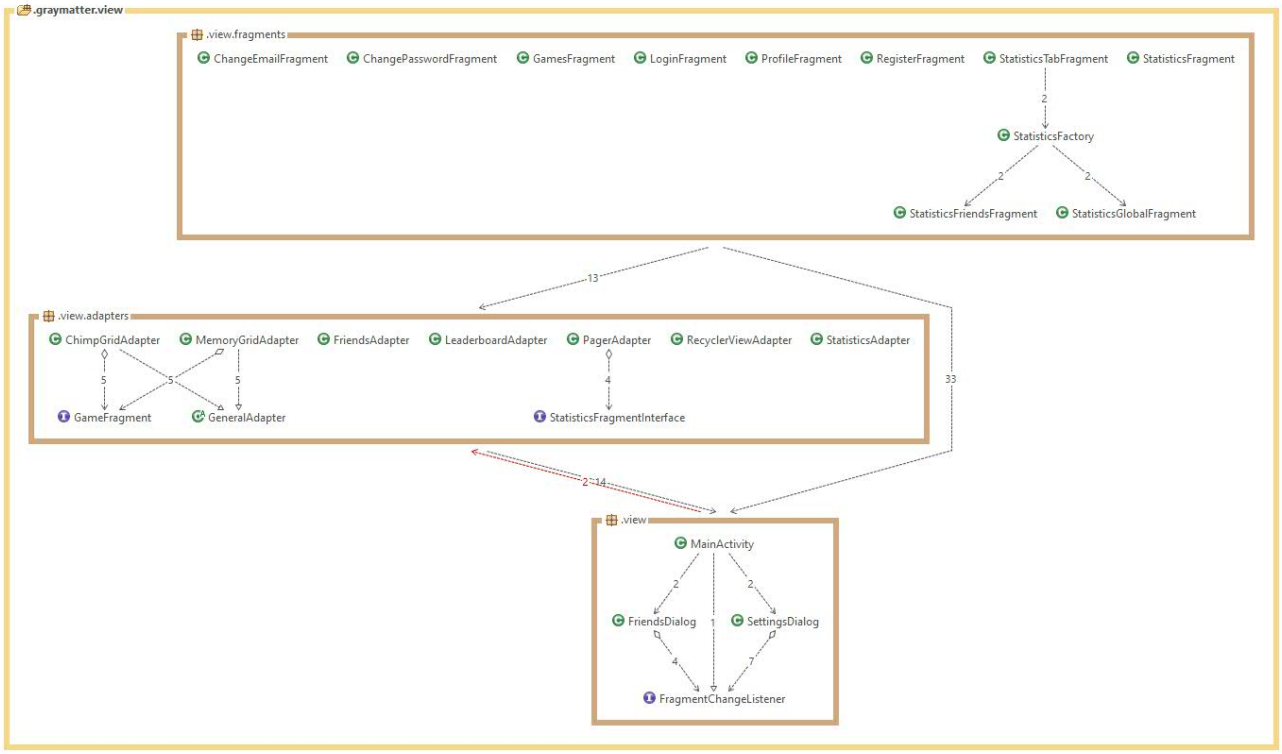

Figure 4: The package model of the application

The general workflow of the View package starts with a main activity that creates fragments and displays the main screen first, where you can scroll by the games. From there you can navigate to the different fragments, for example a game. The list of all the games uses a recyclerView to get a scrollable list with cards that represents the games, each card is also fragment.

When you click on a game you arrive at the game, the game can depend on a gridadapter to make a grid in an efficient way. This is to separate functionality as much as possible, where the adapter is another class that the fragment can use. The adapter draw a grid on the fragment and let the application use each grid as a component, making it easy to control.

3.2 Modular

3.2.1 MVC-pattern

3.2.2 MVVM-pattern

When the View is created, onClick events are also created, to get a response from the user. If something is pressed the View signals to the ViewModel that something has been clicked, which signals to the model that some values have changed. Then the model tells its observers to update with the new directives, and then the same cycle repeats. This is to not make the model communicate with other parts of the program except itself.

We did use Singelton in the game class but removed it, because it is only worth using if you only can have one instance and need to manage that single instance. The problems is that the pattern does not follow the single responsibility principle since it control its own creation and lifecycle. Also that it does not follow the open/closed principle, to change the behaviour or add functionality the code inside the class needs to be modified. To modify the code is much riskier than adding code since it can introduce undesired side effects to other parts of the application.

Template method pattern is a pattern we use to reuse code instead of have redundant code. To make that possible the code that is similar are moved to a separate abstract class that the classes can extend. This is used in for example Game and GeneralAdapter where some methods where almost identical.

3.3 One purpose

Assigning a single responsibility to a package or class goes hand in hand with the last subsection on modularity. It makes it possible to add and remove functionality without other classes getting affected by the change. Since the requirements change throughout the time, the functionality needs to be updated. If each class is separated with its own functionality the change is easy to adjust. It is also easier to understand and read the code since each class have one single functionality, reducing the amounts of bugs and time spent on explaining the code in a project.

3.3.1 Data Mapper

Data Mapper is not a traditional Object oriented design pattern. It is however based on the principle of abstraction. The Data Mapper design pattern is P of EAA pattern intended to keeping database and domain objects seperated. It orders all possible actions with the database to five different methods. Using these methods does not require any other class outside of the datamappers to have an understanding of how the database works.

3.4 Foolproof

By giving copies of instances and variables to other classes we can assure that mutation does not happen in places we do not want it to happen, for the same reason attributes and variables get assigned private and final. But the application is still open for new functionality, an example it the usage of an interface. Interface dependence is a more loose type of coupling compared to for example inheritance, which makes it easy to make extensions to the application without changing the functionality.

3.5 High abstraction

The gameType string thingy makes great abstraction, follows ”Design by contract” and is highly questionable. - String is given by game. Game has only reference (not ensured, fix) ¡explain when we used gamType String and when we constructed different methods¿

3.5.1 String usage

Using Strings for global interactions may seem odd considering the risk of mistakes such as miswriting. This is to simplify adding new games. If new games are added nothing about the dataaccess package needs to change.

3.5.2 Facade pattern

Facade pattern is used in multiple parts of the application. DataAccess is a facade, simplifying client use and preventing other parts of the program from breaking the database structure. The client does not need to rely on dependencies on datamappers and weakens its dependencies on the domain social package.

ScoreFront is another facade, helping the client apply the NormLeaderboard functionality on data from DataAccess.

3.5.3 Repository layer

DataAccess also functions as a repository layer in this application, another P of EAA pattern. This limits possible mistakes in data mapper interactions. Allowed data mappers interactions are defined in the repository layer, preventing clients from breaking the database contents with illegal interactions.

An example: a client wishes to store a game result in database. Client uses the GameSession constructor and then the insert method in GameSessionMapper in an

attempt. It misses the need to store the gameID with the player owner. PlayerAccess method storeGameSession is easy for the client to use and makes sure the DataMapper and Object interations are correctly executed. It is low in viscosity since even though data mapper interactions are possible, the correct usage through the repository layer is more intuitive and easier to use.

4 Persistent data management

Grey Matter uses a mockup Json file as server. This is intended to mimic a real database connection as closely as possible. A Json-server could be hosted at a service such as Firebase for an easy transference to uses a JSON server for storing user data and game records. It uses a local text file for login information.

5 Quality

5.1 Testing

The application is tested with jUnit tests and Travis automatic build testing.The project can be viewed continously at this github repo. The model achieved a 90% test coverage before refactorisationing explained in 5.2. Model coverage is now lower. ViewModel and View tests has been written. Following custom testing has been written:

- Tests on user profile and security
- Tests on social interaction such as adding and removing friends
- Tests on minigames
    - Building the game

      Playing successfully
    - Playing and failing or ultimately losing
    - Breaking the game through unusual play patterns

5.2 Issues

The code has mostly healthy dependecies.The social package has a file, PlayerMapper, which needs to be seperated into smaller areas of responsibilities. It is too long and it not following the DataMapper pattern as intended.

When losing in memory game one does not...

5.2.1 Data management

The data management has caused many issues, causing a stop to all development during the last crucial days before deadline. This resulted in mainly two issues: The database connection being less than ideal and the parts of the project that had to be put on hold to make a function app for deadline.

5.2.2 Database connections

When attempting to implement the usage of backend data-domain mapping packages the program was not able to find the specific path. This was due to the group not understanding Android usage of data. Original design intended for mapping package to only receive a String and solve remaining connection in devoted DataMappers. In current design they receive a Context from where they pull path to local files.

While the projected was never intended to implement a real hosted server, the original design included a Json-server stored at proper locations in app directory. As of now, ”global” data is stored as local app data at users device. It is not part of the project repo, but of the players device or AVD. Login details were meant to be stored in a local cache, when also these implementations failed another local file was devoted to storing these records.

Obviously this is a safety concern, since one could hack another players account by simply exchanging a number in a public file for a public userID. As security was never to be the focus of the project it should have used an external service for login verification such as the ones provided by Google and Facebook.

This results in higher viscosity for changing the connection to real connection, if one wished to start hosting a real game server.

There is dead code in DataMapperImplementation package being a parameterized DataMapper, meant to further abstract the database communication. There were no time for implementing this. It would work fully parameterized or as an abstract class to inherit.

5.2.3 Adding and removing friends

Because of time constraints the function of adding and removing friends were not implemented. Although this is important for functionality, if one wishes to fully develop the application the design would be similar to the process of adding a game.

5.2.4 Testing consequences

JUnit testing coverage on model before the first attempt to connect front-end to database related parts of the model was over 90%. When the database connection method was changed, all database related tests stopped working. Some were converted to Android tests and used when resolving the issues, but some were lost in the process. The original test design was too dependent on nested testing, here as in some domain functionality was only tested through the data mappers.

5.2.5 STAN results

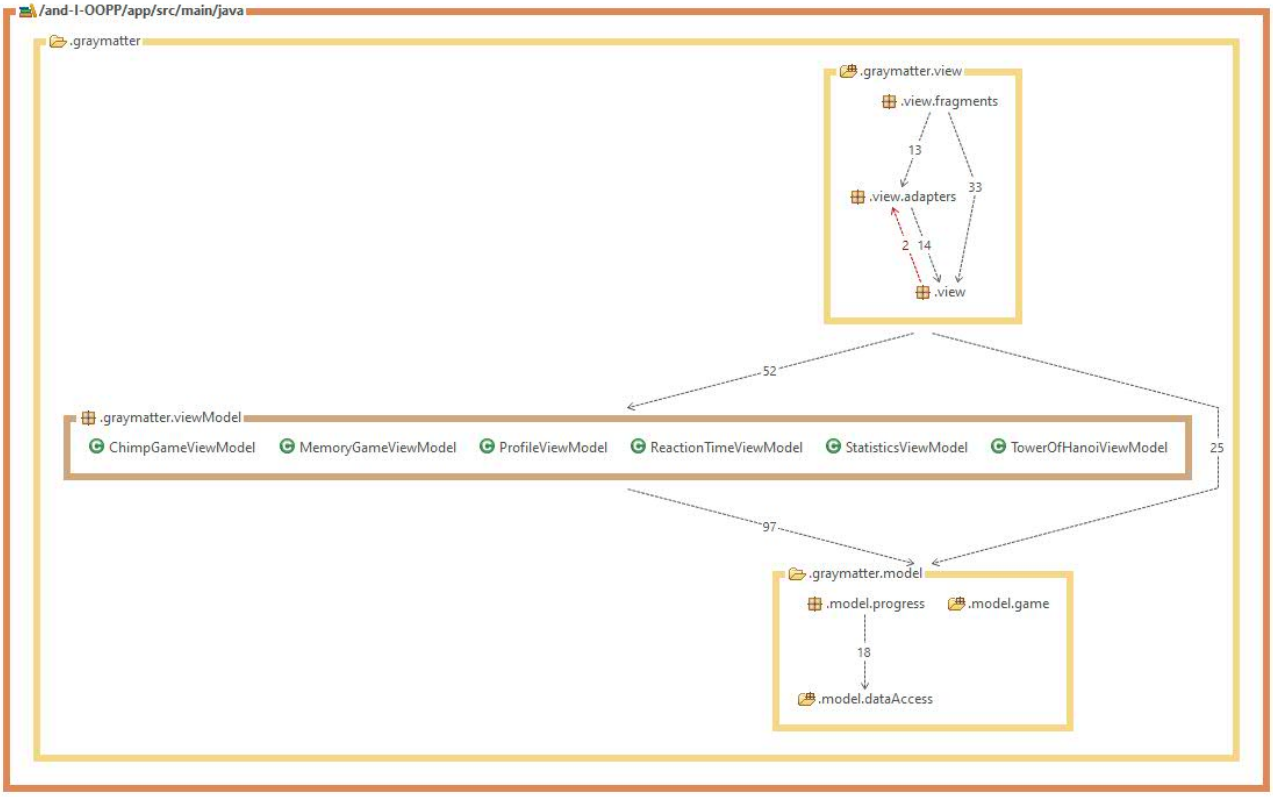

Figure 5: The dependencies in GrayMatter from the application STAN

Diagrams
The STAN diagram shows the dependencies between the Model, View, and ViewModel.

There are no circular dependencies except for the one between adapters and the ViewFragments. These dependencies is necessary because the adapter needs to

know what to print on the screen when the grid is initiated. The View is the object being created first, and then needs a grid that needs to communicate with the Fragment when a grid is pressed.

The VievModels takes care of the communication to the model, and uses the functionality available. It then stores the values and directives for the Fragment to use. The dependencies between Fragments and the ViewModel is also one-way, the Fragments can use the ViewModel without the need to be seen. This makes the the coupling low and enables extension with ease.

The games all have dependencies to the model, to use the functionality. These dependencies are necessary to use the functionality, and the important part it that the model does not talk to any other parts of the program, which it does not.

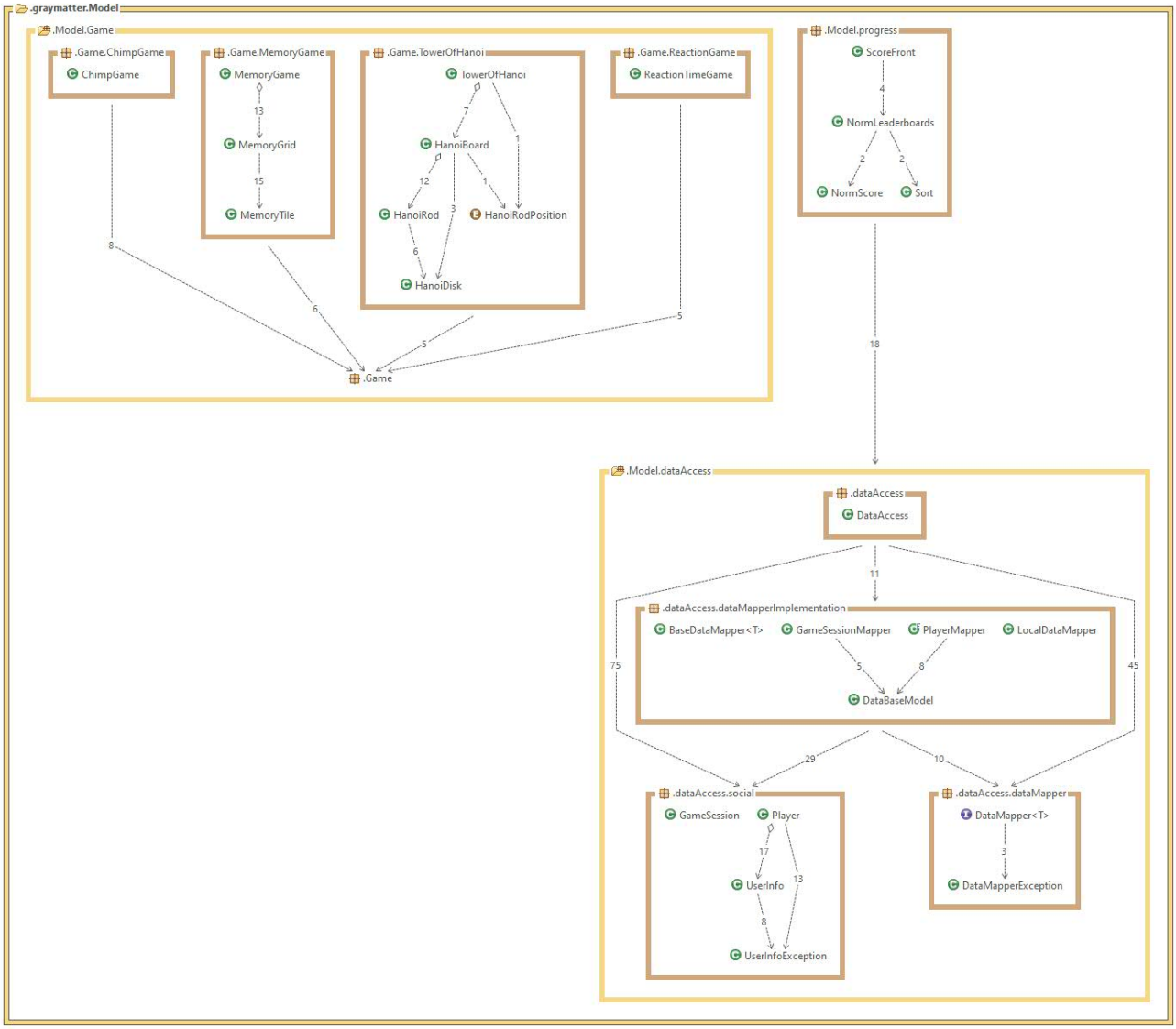

Figure 6: Stan results for model package

5.2.6 Stan model

The model package is divided into three smaller packages. One for the games, which includes each specific game as well as the abstract class Game which all of the for mentioned games inherits from. The game package does not communicate with anything else in the model and can be easily reused in other use cases. The progress and data Access packages however are somewhat connected because the progress package needs a database to to get data from which in this case is the data access

class. The package data Access does however not need any other packages to work and can therefore as the game package be reused in any other application. In its entirety the model package has low coupling and high cohesion which contributes to reusability and ease of maintenance. The package also follows the open closed principle since it is easily extendable by other modules without any changes to the existing code.

5.2.7 PMD results

PMD test found problems mostly related to small fixes as failure to remove unused code and imports, extra brackets of different kinds and misadaption to Java conventions, such as capital letters in the beginning of a package name.
It did however also highlight following mistakes that was deemed more frequent or serious:

- Using unnecessary local variables where one could immediately return result - this clutters the code and uses additional storage.
- Commenting required - lacking commenting or Javadoc.
- Method argument could be final - arguments to different methods could be final.

5.3 Access control and security

Grey Matter uses personal accounts for storing results and social interaction. The system is developed for this application. The user verifies protected actions with a secure password.

6 References

Gradle documentation
Gson documentation https://www.martinfowler.com/eaaCatalog/index.html