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.

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

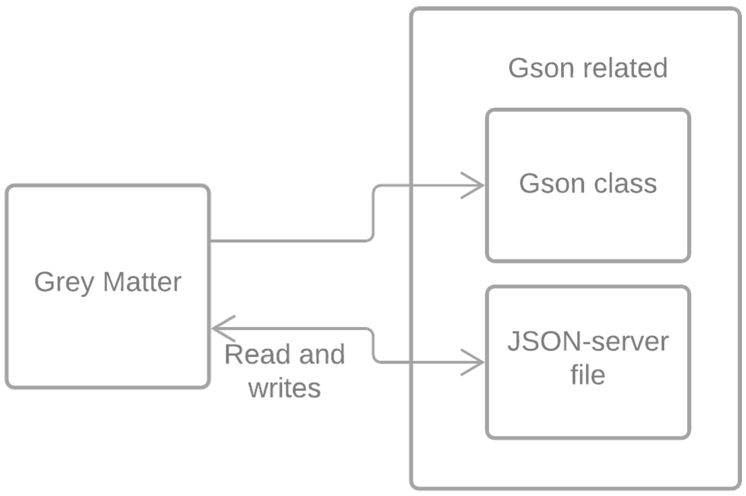

Figure 1: External use of Gson in social package

The social

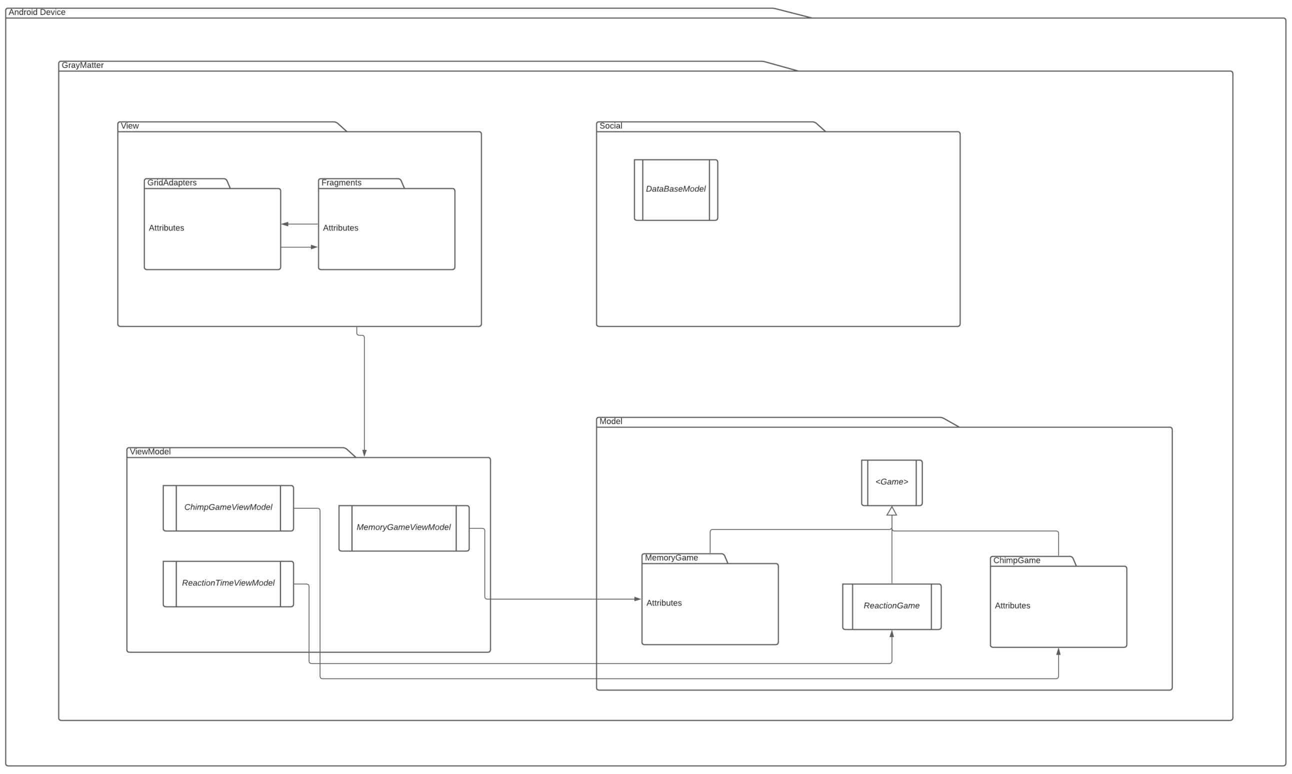

Figure 2: The package model of the application

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 screen, 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 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.

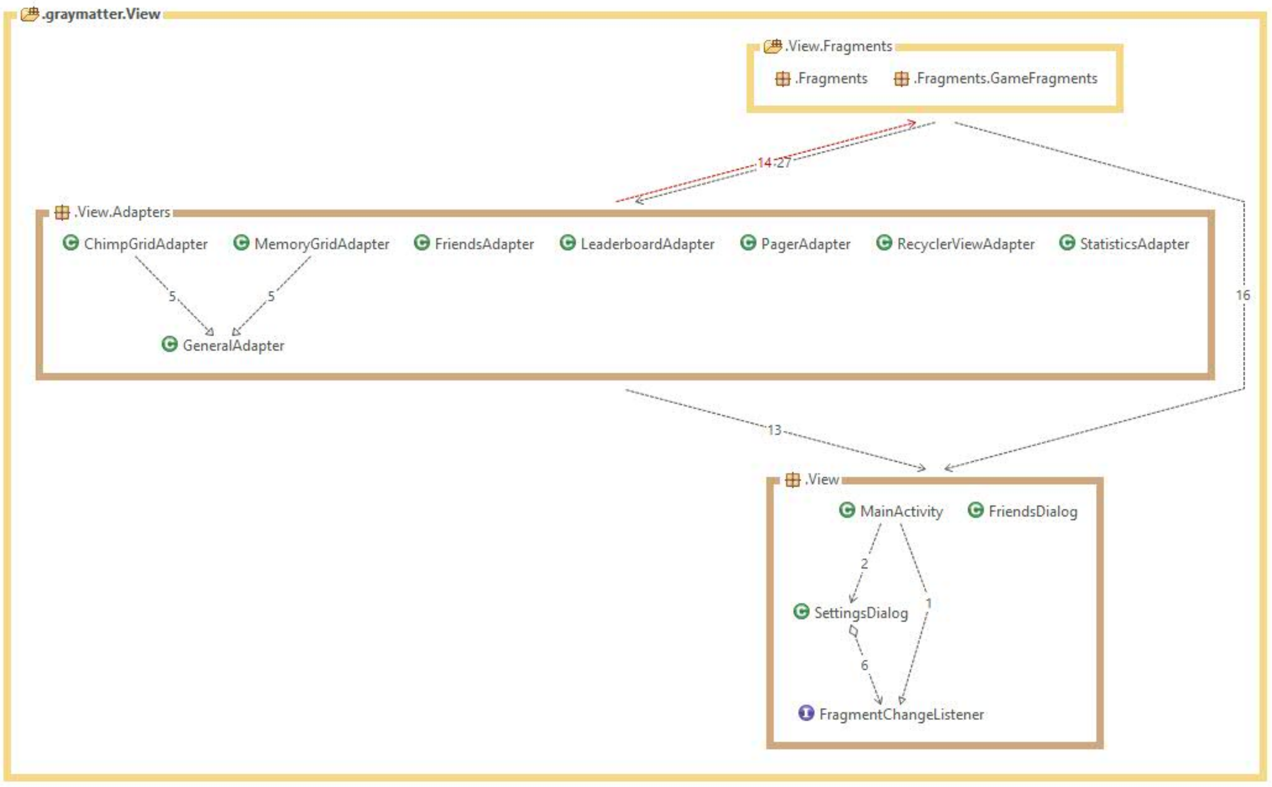

Figure 3: The package model of the application

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.

Interface fragmentChangeListener???

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.
Viktors tankar: Vi använder oss inte av singelton för att inte begränsa oss något (?)
Vi använder inte trådar för det var ett externt bibliotek (?)
I möjligaste mån försöker vi att använda oss av kopior eller andra metoder som private eller final för att minska muterbarhet
Adapter patten? (göra annars inkompatibla komponenter kompatibla) för att kunna ha ett gridview i ett fragment
Facade Pattern (öka abstraktionen genom att gömma intern komplexitet): Detta gör vi med game klassen för att minska beroenden mellan VM och modellen
Composite pattern (använda en grupp av objekt på samma sätt som ett objekt)
Iterator patter (vet knapopt vad det innebär)
State pattern: vi använder oss av states i form av starta och stoppa spel
Template method pattern (när en del av kod är gemensam, bryt ut den del som är gemensam i en abstrakt klass): i game som är en abstrakt klass har vi brutit ut observers som alla spel ska använda sig av
Slut på Viktors tankar

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 it is easy to update the application. It is also easier to understand and read the code since each class have one single functionality, reducing the amounts of bugs and time to explain the code in a project.

3.3.1 Data Mapper

Data Mapper is not a traditional Object oriented design pattern. Although it is based around the principle

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 Every edge case noticed

3.6 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¿

3.6.1 Facade pattern

Facade pattern is used in multiple parts of the application. DataAccess is a facade, helping the client minimize their dependencies on datamappers and domain social package.

3.6.2 Repository layer?

Basically facade in this situation...

4 Persistent data management

Grey Matter uses a JSON server for storing user data and game records.

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.
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 STAN results

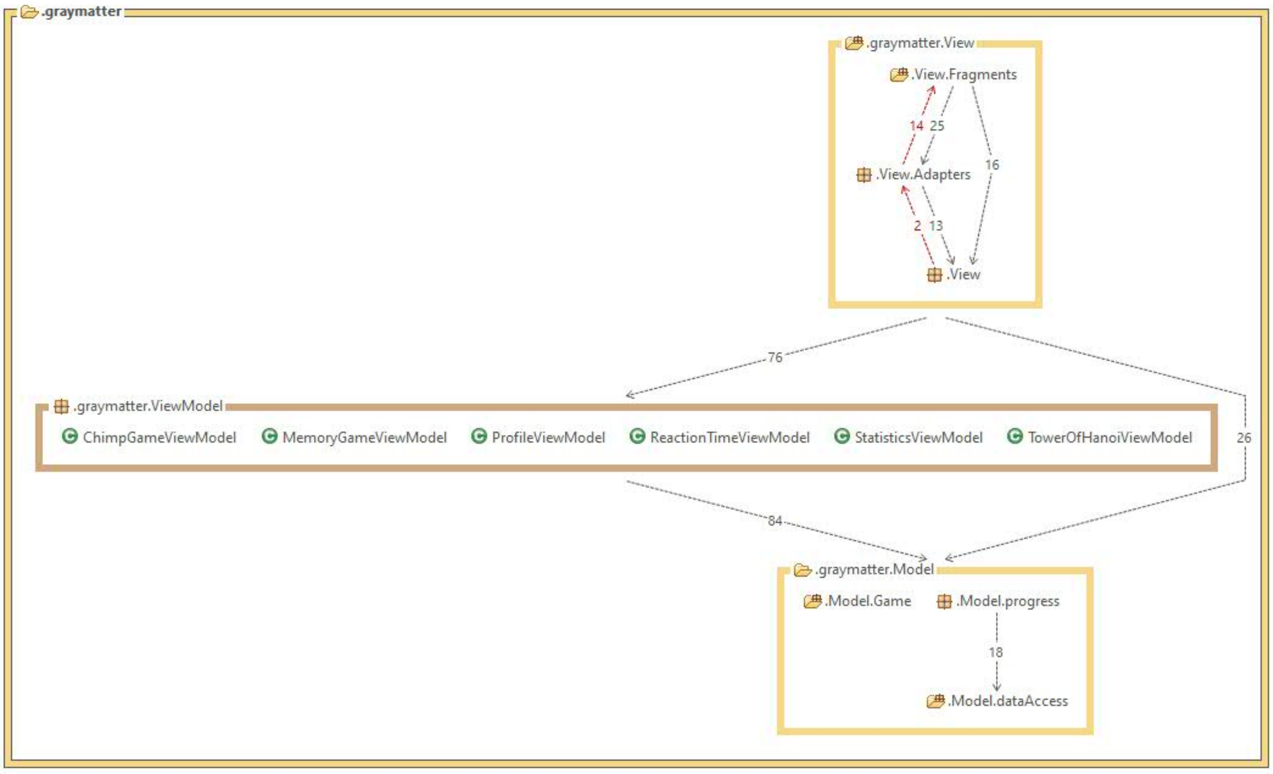

Figure 4: The dependencies in GrayMatter from the application STAN

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 (varför gridAdapter inte finns i fragment vet jag inte (Felix)) 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.
EventBus (ingen aning hur det funkar)

5.2.2 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 in the code:

- Typing to implementation class instead of interface - this limits flexibility.
- Using unnecessary local variables where one could immediately return result - this clutters the code and uses additional storage.

For full PMD analysis see appendix.

5.3 Access control and security

Grey Matter uses personal accounts for storing results and social interaction. The system is developed for this application and the user

6 Appendix

7 References

Gradle documentation
Gson documentation