# System Design Document for Marc

Alexander Sandberg, Victor Johansson,
Mathias Forsman, Gustav Albertsson,
Thomas Li

2018-10-26

Version 1.2

# 1 Introduction

This is an app which is for helping people study with the help of flashcards and quizzes. The user creates their own decks with notes that generates cards that can be used to play different types of learning games. The user can also see his/her own statistics, for example how many games that has been ran, average amount of correct answers etc. Through these stats the user can get different achievements that will help with the feeling of being rewarded. This document describes the architecture behind the application and what how the application is built.

## 1.1 Definitions, acronyms, and abbreviations

- “app” = Shorthand for “application”
- “cards” = Refers to entities containing two sets of information. One of the sets should contain the information that the user can see while reviewing, while the other should contain the information that the user should memorize, with the help of the first set. These sets will often be referenced to as the “front” and the “back” of the card.
- “notes” = Refers to the data collections used to generate cards for usage within the app.
- “game mode” = Different kinds of exercises in the application for practicing the information contained in the cards.
- “deck” = Refers to the collection of notes, which are used to play the different game modes.
- “MTT” = Abbreviation of “Memorization Training Tool”, which holds every user the app contains. This means every deck, every card, all statistics etc.
- “Java” = Specific programming language.
- “MVP” = Model-View-Presenter: A pattern for GUI handling. It’s a way to separate the model (application data), the presenter (what handles the presentation logic) and the view (generates the GUI).

# 2 System architecture

The application is made to run on a Android smartphone running android version 5.0 or later (API level 21 or later), the project is developed in Android Studio. The project does not support Java 8, this is because we have developed the app for devices with API level 21 or later and some of these API levels are too old to support Java 8.

The application runs locally and does not have any connections to any other machine. Everything that the system needs will be automatically started when the application is launched, there is no need to

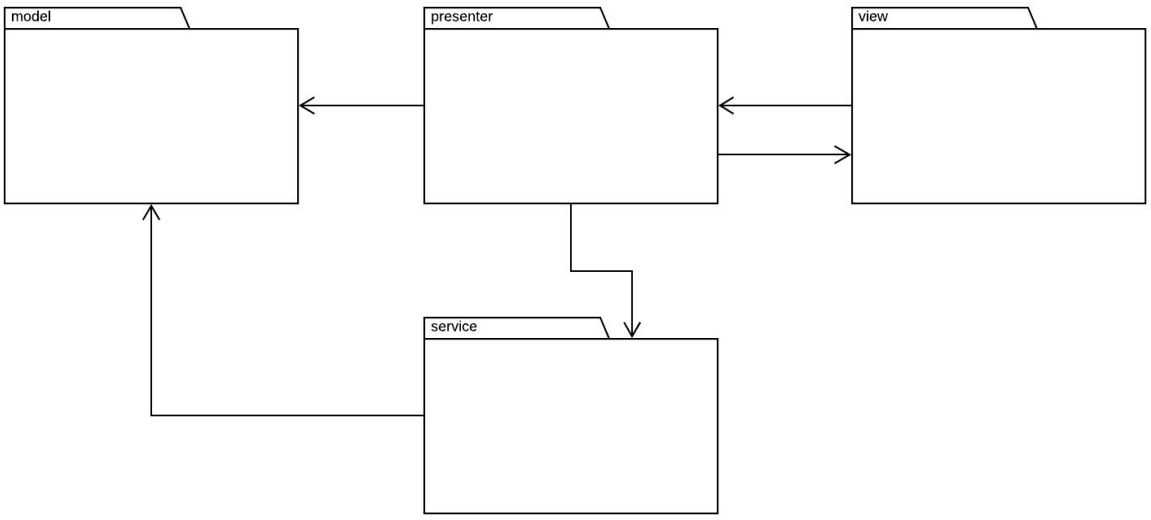
start anything external such as servers. The same procedure goes for when you want to stop the system, i.e the system can be stopped as any other android application at any given time. The application is made of a single component, which contains all of our packages and classes.

## 2.1 Subsystem decomposition

The system only contains one component. There are no multiple subsystems. Our implemented component will be described below, in the following sections.

## 2.2 How MVP is implemented

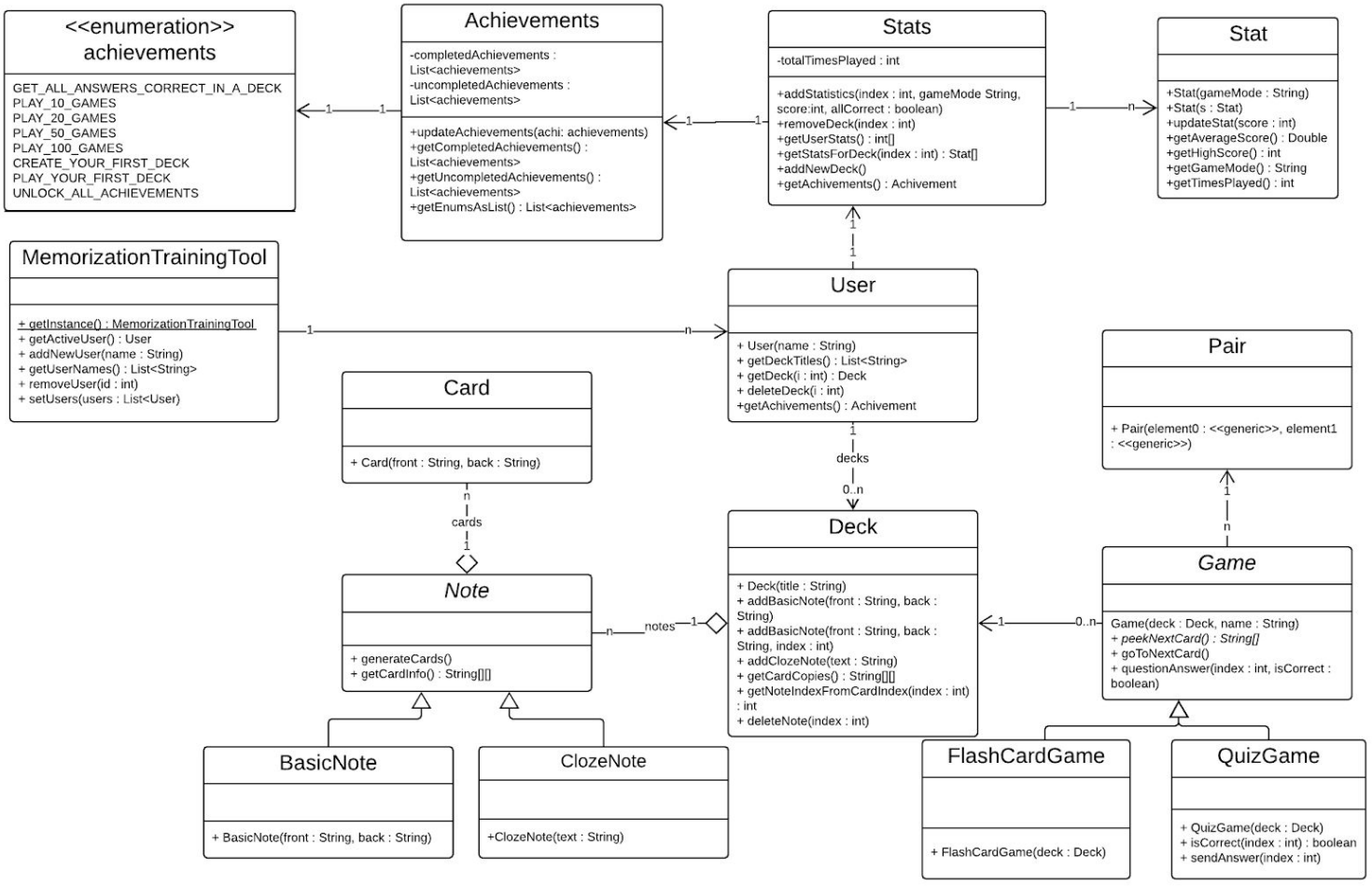
The app is built using the MVP architectural pattern with one model, views and presenters [2]. Every separate screen has an activity which handles everything that should be shown on the screen and detects when the users interacts with the screen. Each activity has a presenter which handles different user inputs and communicates with the needed bit of the model (fetching and updating information in the model). The view package contains different interfaces as well as activities, one for each screen. For example, for the StartMenuActivity there exists a StartMenuView interface. This is done to keep the presenter from being dependent on an actual implementation. The view is responsible for showing things on the screen and detecting inputs. The presenter package holds the presenters which are responsible for setting up the view with information from the model and handles the communication between the view and the model. The service package contains things that are external to the application(such as persistent storage). The model package contains all the rules, logic and data for the application.




*Picture 1. A high-level component diagram*

Picture 1 shows the main concept of how how our app is built. Presenters get and set the data in the model, while the view sets up and handles the GUI. The relation between presenters and views is shown in picture 1 with a circular dependency; this is how MVP is supposed to work. The presenter talks to the service when something needs to be stored on the device, such as Users and their Decks.

## 2.3 Design Model/Model package




*Picture 2. The design model*

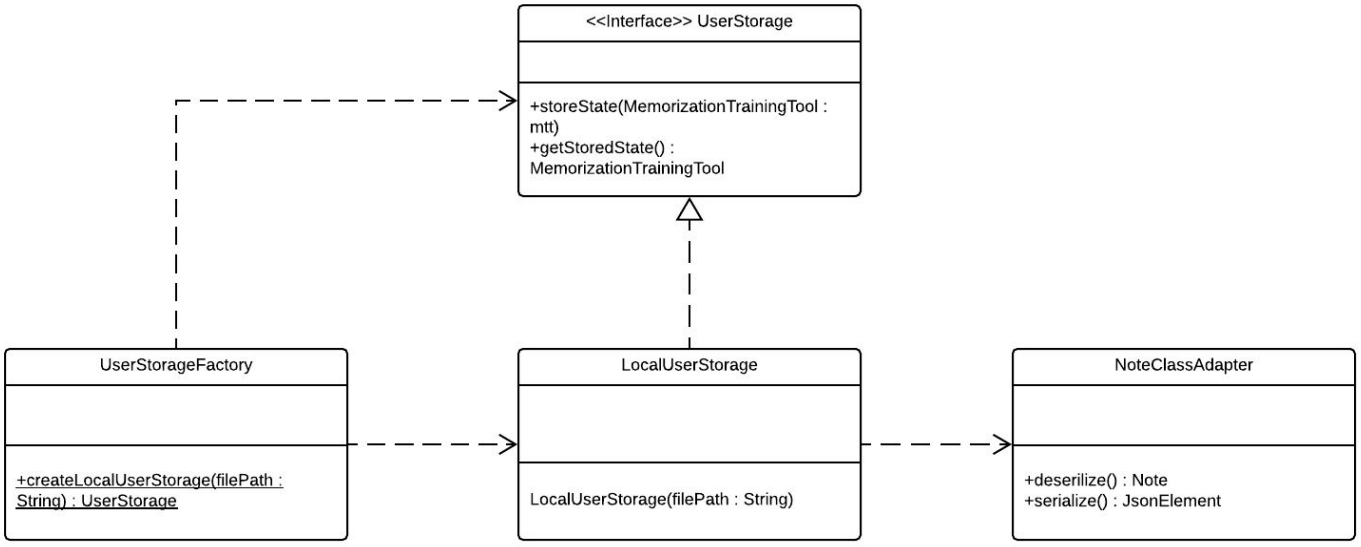
The design model is derived from the domain model. Where each of the “Boxes” in the domain model has been turned into individual classes, there are also additional classes that has been added to better handle things in an object oriented way. The model package holds the data for the application as well as logic for doing things such as playing a Game or creating Decks.

The classes in the design model has the following responsibilities:

- The Deck class has the responsibility of holding Notes that can be used in a exercise.
- The Card class holds the information for a card, which includes text for the the front and the back side of the card.
- The Note class is an abstract class that makes it so that different types of notes can be handled in a general way, and to support different types of Notes without the need to change other parts of the code.
- The BasicNote class holds all the information for a Card, IE just one Card with a front and a back.
- The ClozeNote class holds all the information needed to generate Cards with cloze deletions. This is done by parsing specific syntax in text of the Note.
- The User class holds the information about a user, their name, which Decks they have and how they have performed on these Decks.
- The Stats class holds the statistics for a User, how they performed on their different Decks in different game modes.
- The Stat class holds statistics for a single game mode/Deck combination for a User. It holds statistics about high score, number of times played and average score. The class is also responsible for calculating the new statistics, given the results of a Game.
- The Achievements class holds the achievements that the User is able to receive and are responsible for keeping track of which one are completed and which one are not.
- The Game class is an abstract class which holds code which is the same for different game modes, it also contains abstract methods to force different game modes to behave in a similar way.
- The QuizGame class contains rules needed for the Quiz game mode, using this class one should be able to play a game of Quiz on a Deck. The idea then that the player wont be able to flip the card to see right answer but instead get four alternatives where one is right answer and the other 3 is randomized answers from another questions. The quiz game mode should only use BasicNotes, however currently it can use different Notes which sometimes yields strange quizzes.
- The FlashcardGame class contains rules needed for the Flashcard game mode. If you use this class, these rules will be applied to the Game.
- The MemorizationTrainingTool class has the responsibility of holding references to all the Users and to store which User is active at the moment.

- The Pair class is an implementation of the tuple concept. Tuples are used to hold values, while wanting to have some key that can identify that value. In our case, we use the Pair class in lists, where we hold what questions have been answered and if the answer was correct or not.

## 2.4 Service Package

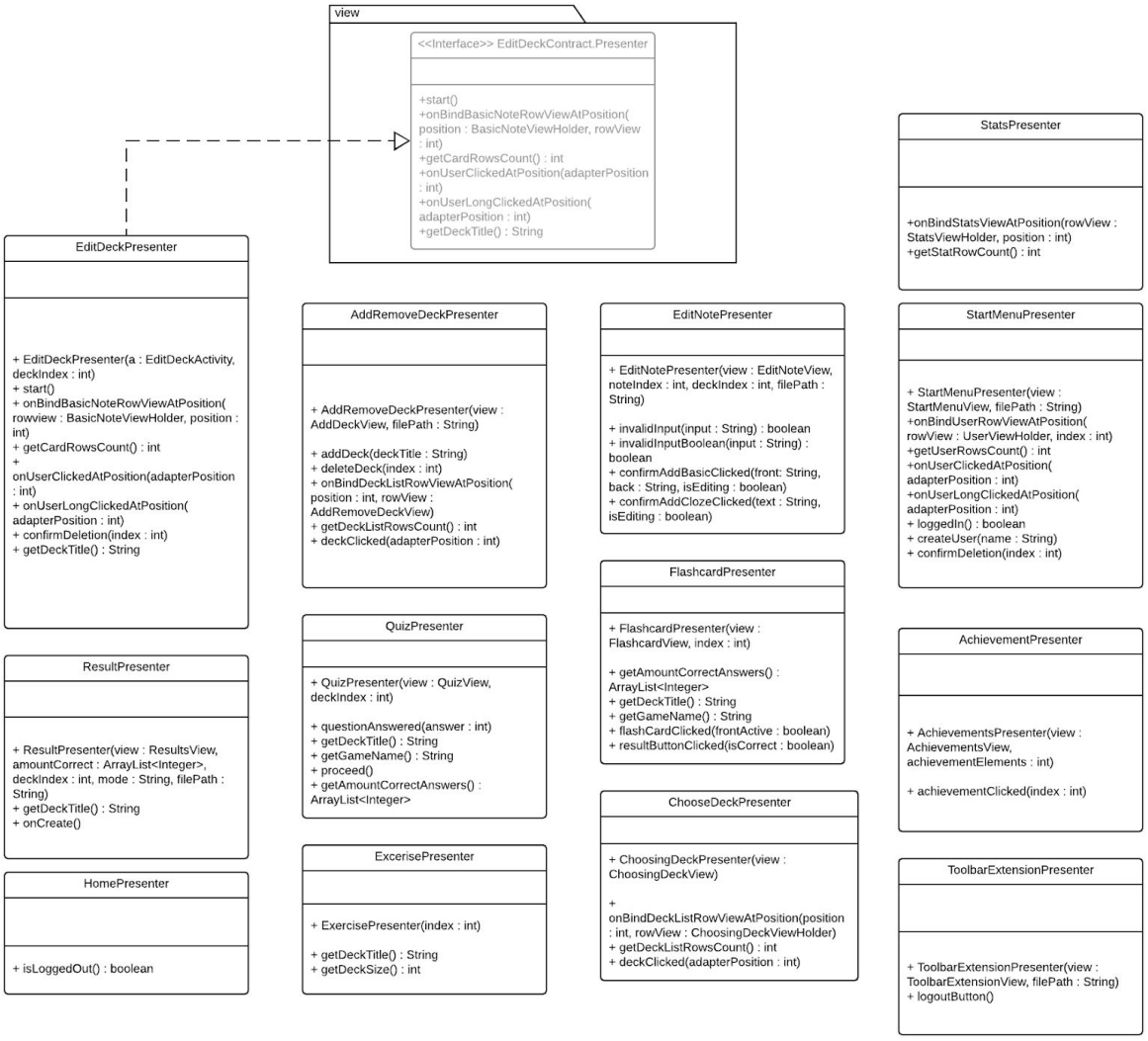



*Picture 3. The service package*

This package contains classes needed for communications with parts that are outside of the application. In this case this package only serves one purpose, persistent storage of user data.
This package only exposes the UserStorage interface and the UserStorageFactory, everything else is abstracted away and not accessible from outside of the package.

- The interface UserStorage contains methods that are needed to have a generic way to save data. This is done to be able to easily swap out how the data is stored as long as we can use the same methods to save and retrieve data about the user.
- LocalUserStorage is an implementation of UserStorage, which saves the data about the users in a local JSON file.
- NoteClassAdapter is a class needed for GSON to be able to handle converting of subclasses of Note to JSON format which can the saved on a file on disk.
- The UserStorageFactory class is a factory class which serves to create objects of the type UserStorage while not exposing the concrete implementations.

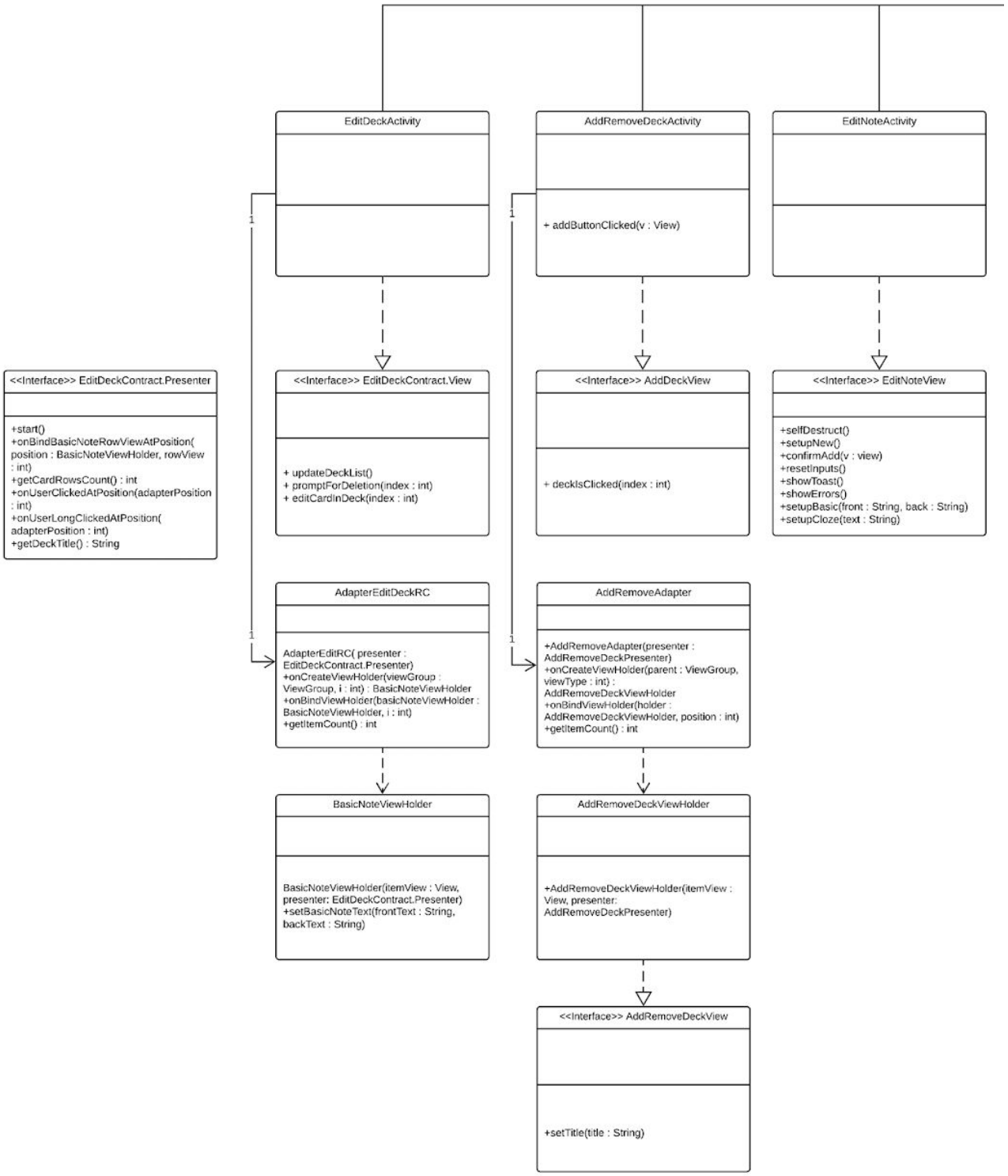
## 2.6 Presenter Package

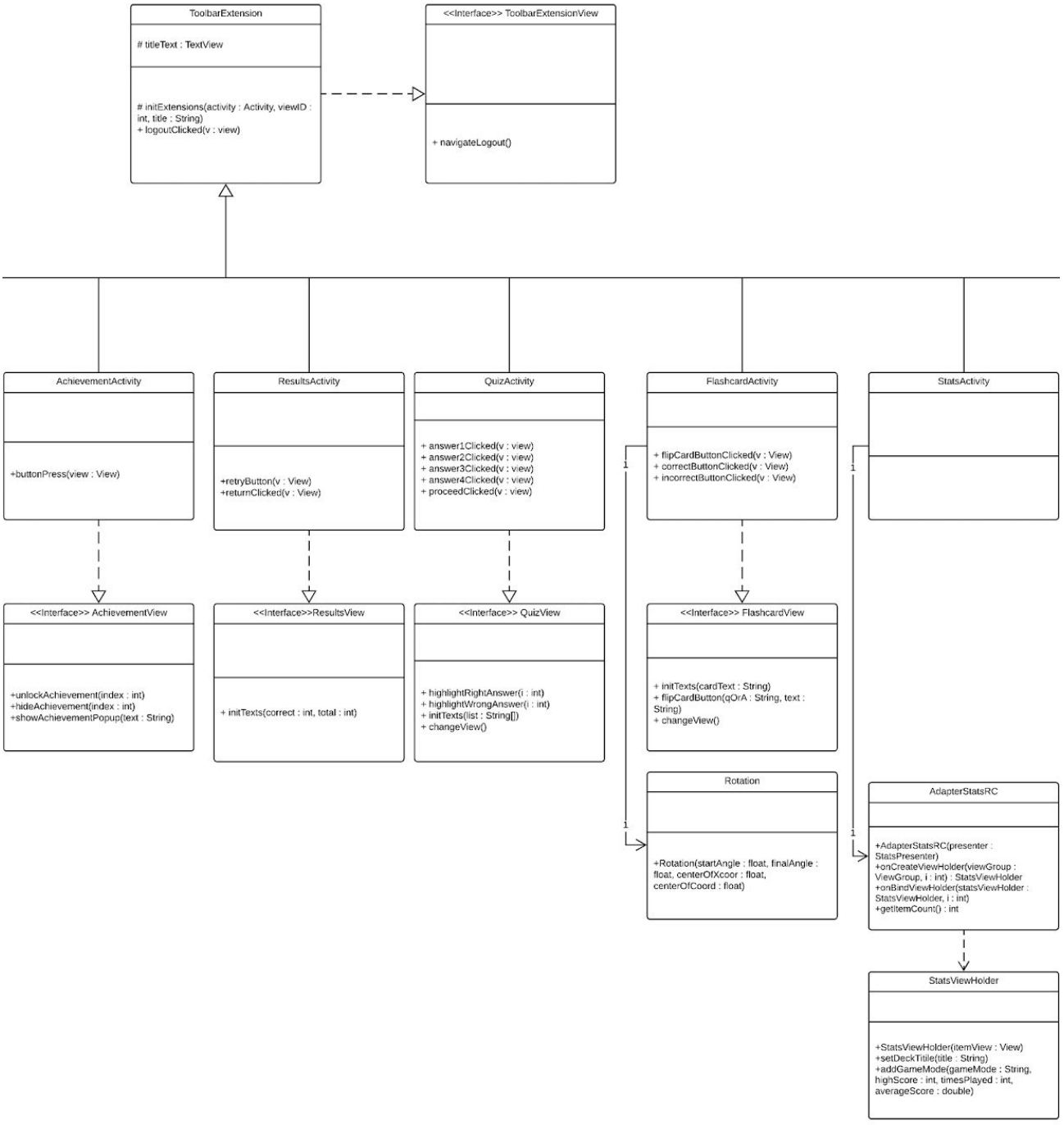


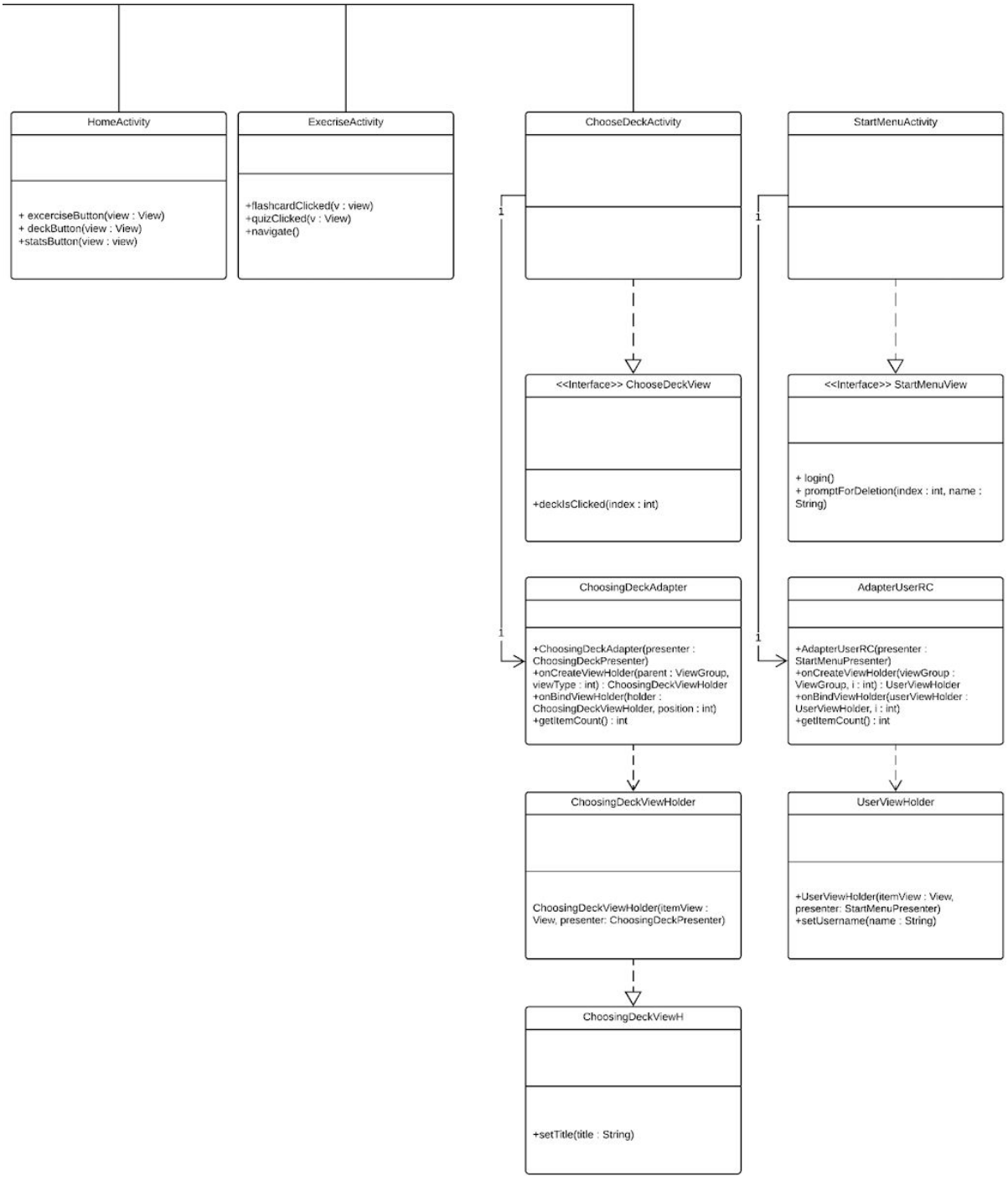

*Picture 4. The presenter package*

The presenter package contains all the presenters that are used in the application, none of the presenters are dependent on each other. The presenter serves to handle the communication between views and the model. The naming convention has activities named "*[name]*Activity" have a related presenter named "*[name]*Presenter". Presenters also serve to communicate with different services to store and receive information from external sources. In the case of this application presenters talk to services which handles persistent storage of user data.

## 2.7 View Package




ToolbarExtension
# titleText : TextView
# initExtensions(activity : Activity, viewID : int, title : String)
+ logoutClicked(v : view)
<<Interface>> ToolbarExtensionView
+ navigateLogout()
AchievementActivity
+buttonPress(view : View)
ResultsActivity
+retryButton(v : View)
+returnClicked(v : View)
QuizActivity
+ answer1Clicked(v : view)
+ answer2Clicked(v : view)
+ answer3Clicked(v : view)
+ answer4Clicked(v : view)
+ proceedClicked(v : view)
FlashcardActivity
+ flipCardButtonClicked(v : View)
+ correctButtonClicked(v : View)
+ incorrectButtonClicked(v : View)
StatsActivity
<<Interface>> AchievementView
+unlockAchievement(index : int)
+hideAchievement(index : int)
+showAchievementPopup(text : String)
<<Interface>>ResultsView
+ initTexts(correct : int, total : int)
<<Interface>> QuizView
+ highlightRightAnswer(i : int)
+ highlightWrongAnswer(i : int)
+ initTexts(list : String[])
+ changeView()
<<Interface>> FlashcardView
+ initTexts(cardText : String)
+ flipCardButton(qOrA : String, text : String)
+ changeView()
Rotation
+Rotation(startAngle : float, finalAngle : float, centerOfXcoor : float, centerOfCoord : float)
AdapterStatsRC
+AdapterStatsRC(presenter : StatsPresenter)
+onCreateViewHolder(viewGroup : ViewGroup, i : int) : StatsViewHolder
+onBindViewHolder(statsViewHolder : StatsViewHolder, i : int)
+getItemCount() : int
StatsViewHolder
+StatsViewHolder(itemView : View)
+setDeckTitile(title : String)
+addGameMode(gameMode : String, highScore : int, timesPlayed : int, averageScore : double)
1
1
1
1




*Picture 5, 6, 7. The view package UML diagram (the full diagram can be found in the repository in Documentation/uml)*

The view package contains all activity classes, displayed in the UML-diagram above. Each activity is connected to a unique XML-file. It is through the activity, that you are able to manipulate the UI displayed by the XML-file. An activity has got no relations to the model, directly. To get and set data

in the model, the activity will create an instance of a presenter, introduced in the section “Presenter Package”, above. Methods implemented in this instance(of the presenter) will be called by the activity in order to get and set data in the model. To visualize these changes, methods in the activity will be called by the activity’s presenter, through an interface implemented by the activity, called a view.

A view is an interface for an activity. This interface is needed when the model’s data changes need to be updated in the activity. The presenter will have an instance of this interface, rather than the activity itself. Thereby, the presenter is merely able to call methods that are contained in the interface, rather than methods contained in the activity. This also means that the activity that implements the interface can be replaced without doing any modifications to the presenter.

A unique activity in this package is the activity called ToolbarExtension. It is an abstract class, made to simplify the implementation of a customized toolbar and quick menu for any activity that wishes to utilize it. This class hierarchy allows for less copy-paste coding, as all functionality for setting up the toolbar and side menu is handled in ToolbarExtension. ToolbarExtension has got three XML-files of itself, and a presenter.

The XML-files in the layout folder are used in the view package, but are not a part of the package, nor are they displayed in the UML-diagram. These files are responsible for the UI that is to be displayed for our users. It is through this UI that the users interact with the application. When a button is clicked in the UI, the XML-file will possibly call a method that exists in an activity connected to it.

## 2.8 Code Quality

### 2.8.1 Tests

The application is tested with unit tests written with the library JUnit. The unit tests for the app can be found in the directory: *Marc/app/src/test/java/com/example/ohimarc/marc/*. The project is setup to use *TravisCI* and will compile the code and run all unit tests every time a commit is pushed to the repository.

### 2.8.2 Coverage

The model and service packages has been tested with unit tests with ~95% line coverage. The presenter and view packages have no unit tests, the tests on these are purely from a user’s point of

view how the app should behave and react to the user's inputs. These could be tested with some mocking framework, but due to lack of time this was not done.

### 2.8.3 Quality Tools

The project has been checked by the built in tool in Android Studio called "Inspect code" (Analyze>Inspect Code). This is a tool built upon IntelliJ code analyser which is a static code analyser which detects language and runtime errors.[1] The tool gives suggestions on changes based on the results of the test. This tool gives different categories of problems with the code Android, Java, Spelling and XML. This project has focused mainly on the Java and XML categories but also Android and Spelling has been looked into and major problems has been resolved.

## 2.9 Model usage in the application

Since Android does not support passing object references between different "Activities" the model that is used by the app is a singleton that can be accessed via the MemorizationTrainingTool class. Altho it would be preferable to only pass the correct references around and not having to use a static method in MemorizationTrainingTool to get the model, it is not possible. Another possible implementation that would avoid using a static method would be to use the persistent storage to read and write the user's data to disk when changing activities. However this would require reading and writing from disk and converting objects to a format which is able to be stored in a file. All this could in theory slow down the application when dealing with large amounts of data or if the phone's hardware is slow. This is something that the singleton approach circumvents. Since the model is only handled by the presenters changing the way that the model is handled in the application would only need a minimal amount of changes in the presenter classes.

## 2.10 Dependencies

The system is constructed with the architectural pattern MVP as explained earlier. Usage of this architectural pattern provides some clear directives to whom communicates with whom. As seen in appendix 2, the code is written in a way such that the model is dependent on nothing but itself, presenters and views are dependent on each other. None of the presenters are dependent on each other since they only serve to communicate between the views and the model/services.(see appendix 1)
It is usually bad to have a circular dependency but in the case of MVP it is required to have it between views and presenters. The things that the model exposes are often limited, when using the model via the MemorizationTrainingTool class one can only access a handful of classes, classes such as Notes and Decks are never exposed to presenters or views, they are instead represented by Arrays of strings

which serves the same purpose. This has been done in order to hide away much of the internal class structure and complexity to classes using the model. Even within packages some abstractions have been made, for example neither FlashCardGame nor QuizGame knows about the concept of Cards and Notes, they only know about Decks. This means that other types of Notes could be added without affecting the different game modes. See appendix 5 and picture 2 for the exact dependencies between the classes in the model package. By limiting what is exposed the system that is created is one that has low coupling which means that it is prone to less problems when expanding different parts of the system. To further reduce the coupling and create abstractions between packages most of the communication from the presenter to the view package are done via interfaces. Most presenters don't know about a concrete implementation they know about an interface. The service package is implemented in such a way that it exposes very little of its internal complexities to the outside of the package, it only exposes a factory class and an interface. The exact dependencies within the service package can be found in appendix 4. The dependencies within the view package can be found in appendix 3.

## 2.11 Build Tool

The application uses Gradle version 3.1.4 as its build tool. Gradle is used to run the unit tests and get remote packages(dependencies) as well as building the application.

## 2.12 Concurrency issues

The application only runs on one thread, therefore there are no concurrency issues.

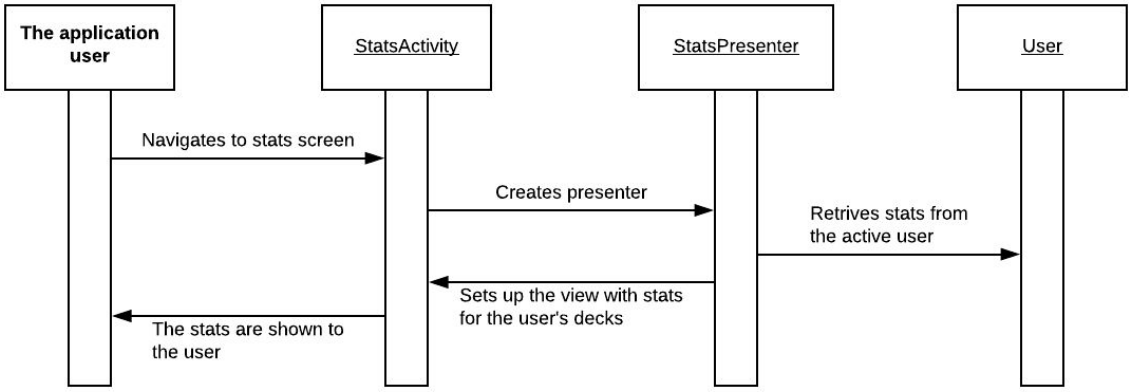
## 2.13 High-level Flow of some user stories

The sequence diagrams presented below are diagrams which describe how the classes interact with each other during some user stories. The column all the way to the left is the actual user of the application its is not a class. The diagram is then read by following the arrows while reading the diagram from the top to the bottom.

### 2.13.1 Checking stats

Checking your statistics as a user:

1. Click on "Statistics"
2. Look up the deck you would like to see your statistic on, if you do not see it you do not have any statistic on that deck or you have to scroll down a bit to find it.



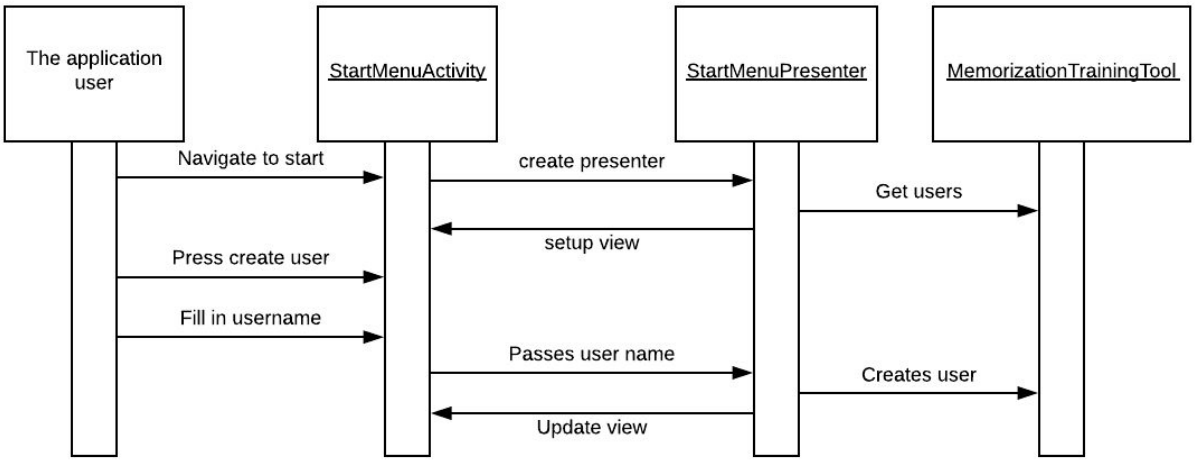

*Picture 6: The UML sequence diagram for checking one's statistics as a user*

The user of application (See the picture above named "The Application User") enters the screen where the active user's (the user that is logged in) stats are shown. The activity for Stats is created and creates its presenter. The presenter gets the Stats for that User from the active User object and then tells the activity what it should print out on the screen, in this case the Stats for all the Decks the active user have ever played.

### 2.13.2 Login/create a user

This user story contains two actions, logging into a user and creating a user.
Creating a user:

1. Enter the start screen
2. Press plus button
3. Write user name
4. User gets created

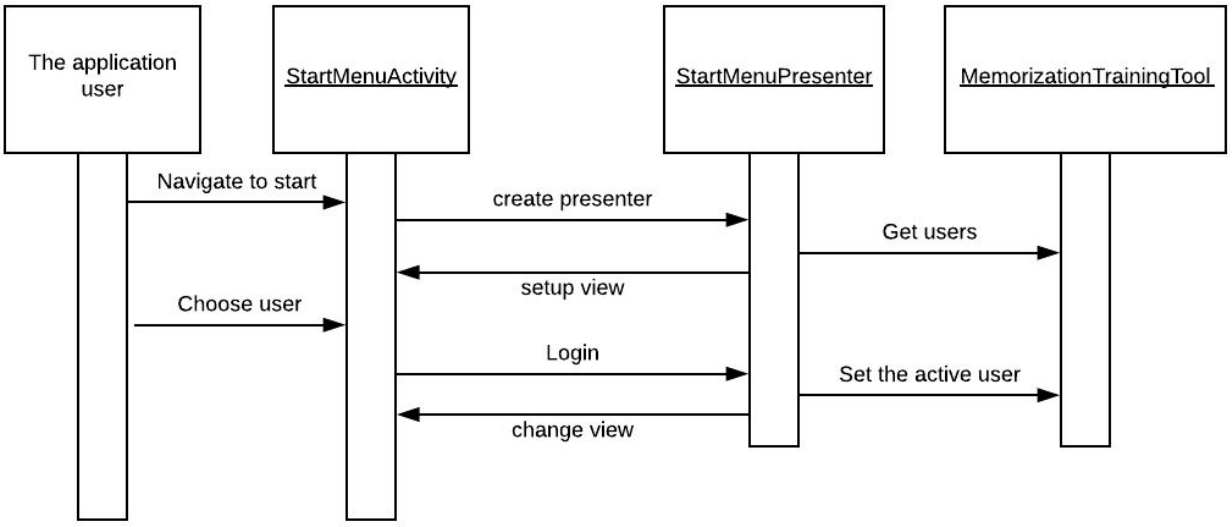



*Picture 7: The UML sequence diagram for creating a user*

When entering the StartMenuActivity the activity creates a StartMenuPresenter, the presenter retrieves all the user names from the model and sets up the view. The user presses the plus sign to add a new user, this will create a popup where the user can write a user name. The user fills in the user name and presses done, this will be sent of to the StartMenuPresenter which will talk to the model to create the user in the model. Once the user is created the StartMenuPresenter updates the view to display the new user.

Login as a user:

1. Enter the start screen
2. Click a user
3. User gets logged in



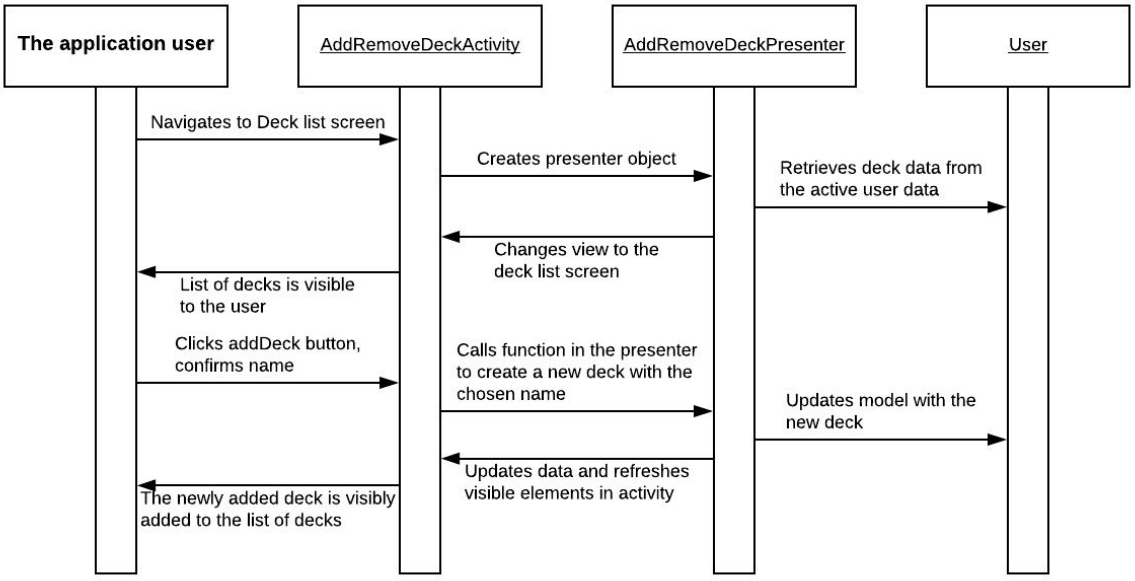

*Picture 8: The UML sequence diagram for login into a user*

When entering the StartMenuActivity the activity creates a StartMenuPresenter, the presenter retrieves all the user names from the model and sets up the view. The user then presses a user to login into the user. When the user is pressed the activity sends this to the presenter which talks to the model to actually log the user in. When the user is logged into the model, the presenter tells the StartMenuActivity to change the view since the user is now logged in.

### 2.13.3 Create a new deck

Create a new deck as a user:

1. The user enters the decklist screen from a previous screen.
2. The user clicks the plus button to bring up a pop-up, prompting the user for a name for a new deck. After filling in the name, the user confirms the addition of the new deck.
3. The deck gets created and the list of decks is visibly updated to the user.




*Picture 9: The UML sequence diagram for creating a new deck as a user*

When entering the AddRemoveDeckActivity the activity creates a presenter object, this object retrieves the user's Deck from the model. The presenter then updates the view to display the Decks. The user then presses the plus sign to add a new Deck, this will create a popup where the user can fill in the name of the Deck. Once the user confirms the name this will be sent of to the presenter which talks to the user object to create a new Deck in the model. Once the Deck is created the presenter will update the view again to show the new Deck.

# 3 Persistent data management

## 3.1 Icons

The icons are made using vector graphics in the application. These are saved using XML-files which are saved locally on the phone. The reason for using vector graphics is to make sure they look sharp on different screens with different sizes. Any custom-made icons that were needed were made in the application *Inkscape*.

## 3.2 User data

The data about the users is stored locally on the phone and persists between app sessions. The information about the user is stored in a JSON file. The User objects are converted into a JSON format using the GSON library which translates an Java object to and from JSON. The converting is

done via a service which hides the implementation details of storing and retrieving Users from the JSON file.

## 4 Access control and security

The app can handle multiple in-app user profiles which saves and separates data for each user. This data contains, for example, the user's stats and their collection of decks, cards, and notes. The users cannot set a password or add any form of authentication to their profile. The user system is not intended as a privacy or security measure, as the app is only supposed to be used locally. It exists solely to make it easier for the users to share the same device and get less clutter by not having to see each other's decks.

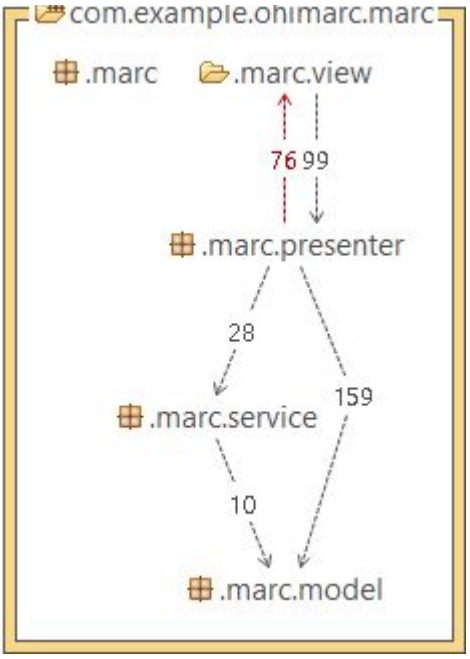
## 5 References


1. Jetbrains.com. (2018). *Code Inspection - Help | IntelliJ IDEA*. [online] Available at: https://www.jetbrains.com/help/idea/2018.1/code-inspection.html [Accessed 19 Oct. 2018].
2. Maxwell, E. (2017). *MVC vs. MVP vs. MVVM on Android*. [online] Academy.realm.io. Available at: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/ [Accessed 21 Sep. 2018].

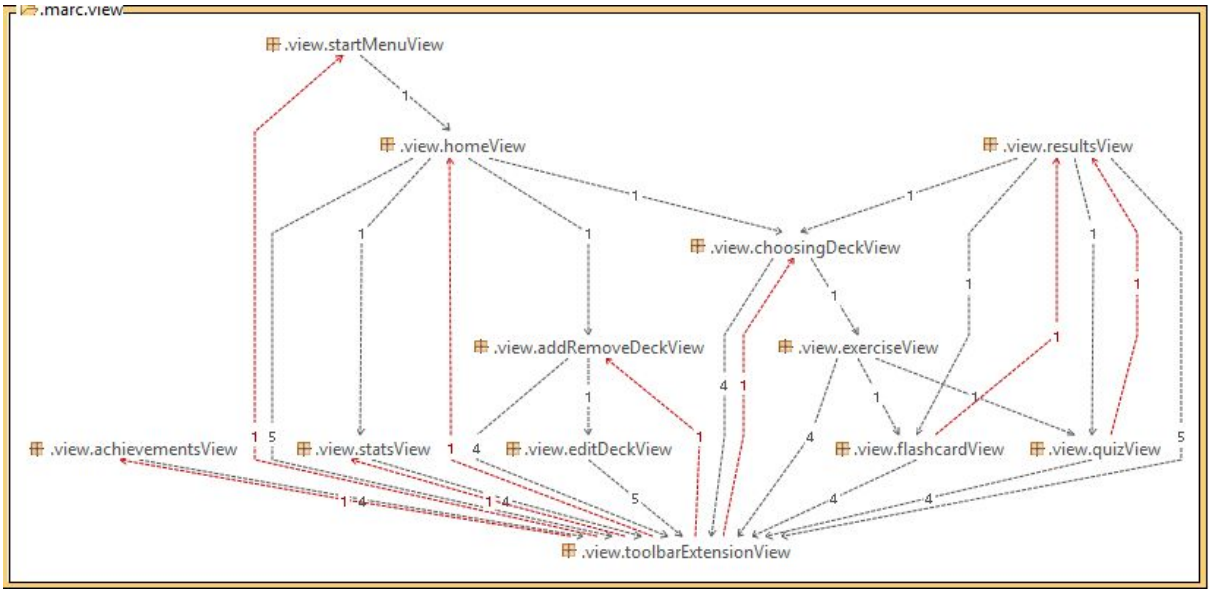

## 6 Appendix

appendix 1:

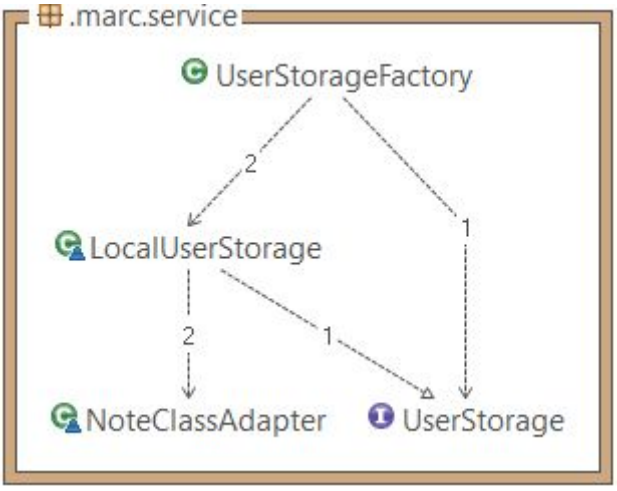

appendix 2:



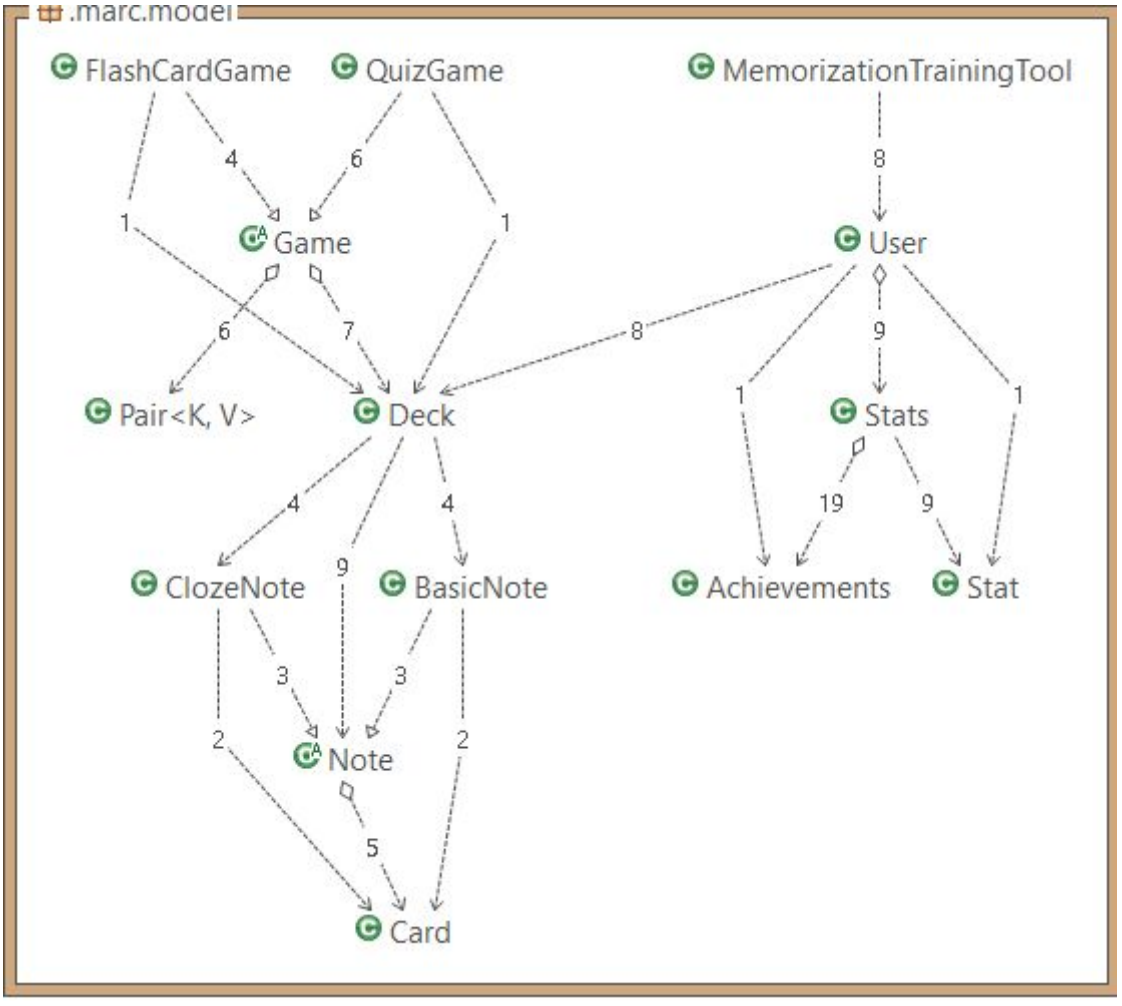

appendix 3:




appendix 4:


.marc.service
UserStorageFactory
2
1
LocalUserStorage
2
1
NoteClassAdapter
UserStorage

appendix 5:


.marc.model
FlashCardGame
QuizGame
MemorizationTrainingTool
4
6
8
1
1
Game
User
6
7
8
9
1
1
Pair<K, V>
Deck
Stats
4
4
19
9
9
ClozeNote
BasicNote
Achievements
Stat
3
3
2
2
Note
5
Card