Final report for Marc

Thomas Li, Gustav Albertsson,

Alexander Sandberg, Victor Johansson,

Mathias Forsman

Group 12

2018-10-26

version 1.1

1. Introduction

For students, there will often be situations where they need to learn definitions and vocabulary, and while important, this is a process that can often become tedious and suffer from a lack of cognitive stimulation. Marc aims to make studying more rewarding and convenient by providing tools to easily organize the information that they need to study, as well as several different ways of practicing, by way of different kinds of more "game-like" methods, such as quizzes or other challenges. The application also keeps track of the user's performance while studying in order to make it easier to view progression and improvement, encouraging the user to develop more regular review sessions. By showing stats and user progression, it is also easier for the user to feel accomplishment when they achieve goals, whether set up by themselves or as existing in-application challenges.

- The application addresses the difficulty and general lack of stimulation that comes from studying to remember things such as vocabulary, that primarily needs to be memorized.
- The application will provide tools for organizing groups of user-defined information and tools to promote repeated reviewing of said information.
- The main user group will probably be students, who often needs to memorize definitions and vocabulary.
- Since the application will be developed for mobile devices, it is suitable for use on the go. For example, while commuting to and from school, one could use the perhaps otherwise wasted time to do some quick studying. The application should be designed to allow for quickly starting up and closing down review sessions.

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.
- "decks" = Refers to the collection of notes, which are used to play the different game modes.
- "game mode" = Different kinds of exercises in the application for practicing the information contained in the cards.
- "notes" = Refers to the data collections used to generate cards for usage within the app.

- “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. Requirements

2.1. User Stories

User story

Story Identifier: 0

Story Name: Memorization Training Tool

Description

(Epic)

As a student, I want an application where I can practice my different subjects using flashcards and other types of fun exercises such as quizzes, so that I can study more efficiently.

Confirmation

Functional

- The application should allow for doing exercises with both flashcards and any other supported exercise types (such as quizzes).
- The application should allow for adding/removing/editing decks of cards.
- The application should keep stats of the user's progress and exercise results.

Non-functional

Availability:

- The application should be working on a phone using a version of Android of 5.0 or above (Lollipop).
- Cards and stats should persist between application sessions, on the same device.

User story

Story Identifier: 1

Story Name: Usable flashcard

Estimated time: 6 days

Description

As a student, I want a flashcard to be visible in the application, with a predetermined text written on the front, and the answer on the back, so that I can test myself on the information on the card.

Confirmation

Functional

- The card should have some text written on it.
- The text written on the card should fit to the size of the card by being shrunk.
- The card should be clickable, which reveals the correct answer to the question.
- The card should be marked with a Q when the question is displayed.
- The card should be marked with an A when the answer is displayed.

Non-functional

- The card should be well fitted to the current screen.

User story

Story Identifier: 4

Story Name: Show user performance

Estimated time: 6 days

Description

As a student, I want to be able to tell the application if my answer to the card was correct or not on a predefined deck, and for the application to keep track of my performance, so that I am able to see the result.

Confirmation

Functional

- The deck should contain a number of predefined flashcards, with questions and answers on the front and backs.
- The flashcards should each have a “correct” and “incorrect” button below.
- Clicking the “correct” or “incorrect” button should show the next card in the deck.
- The user inputted answer (correct or incorrect) should be saved till the end of the game
- At the end of the game, the results should be displayed.

Non-functional

None.

User story

Story Identifier: 5

Story Name: Building main menu

Estimated time: 3 days

Description

As a student, I want my application to start in a main menu, so that I can navigate to playing my deck.

Confirmation

Functional

- The main menu should have a button to navigate to the decks.
- There should be a button to navigate to Exercise
- There should be a button to navigate to Statistics

Non-functional

- There should be a header which reads “Marc”.
- The buttons should have fitting icons.

User story

Story Identifier: 7

Story Name: Add/remove/edit flashcards

Estimated time: 10 days

Description

As a student, I want to be able to add/remove/edit flashcards in the existing deck, so that I can modify my deck of cards.

Confirmation

Functional

- The text on each card should be editable (by editing the corresponding Note).
- Each card and their corresponding note should be removable.
- There should be a (+) button, which when clicked, shows a prompt to add a note.
- Notes *cannot* be empty on either side.
- Long pressing a card should prompt the user for deletion of the card that has been long pressed.
- Tapping a card should enable editing of the corresponding note:
    - While editing a note, there should be buttons for saving changes.

Non-functional

- There should be a screen where all cards in the deck should be displayed with both the front and back visible.

User story

Story Identifier: 12

Story Name: Add/remove/edit cloze cards

Estimated time: 10 days

Description

As a student, I want to be able to add/remove/edit cloze cards in the existing deck, so that I can modify my deck of cards.

Confirmation

Functional

- The text on each card should be editable.
- Each card should be removable.
- There should be a (+) button, which when clicked, shows the screen for adding Notes. This screen should let the user select Cloze as the Note type.
- The syntax for adding a cloze should be something unusual, which is not easy to write without intent.
- Tapping a card should enable editing of the corresponding Note.
    - While editing a Note, there should be a button for saving changes.
- Long pressing a card should prompt the user for deletion of the corresponding note (and therefore also any other cards associated with the deleted Note).

Non-functional

- There should be a screen where all cards in the deck should be displayed with the current text written on them.
- Removing, editing and adding a card should force the user to do such an action to the Note associated to the card.

User story

Story Identifier: 8

Story Name: Add/remove deck

Estimated time: 5 days

Description

As a student, I want to create/remove decks of flashcards, so that I am able to keep collections of different types/groups of flashcards.

Confirmation

Functional

- Swiping a deck to either direction should prompt the user to confirm deletion of the selected deck.
- A (+) symbol should be visible when you're in the "Decks" screen.
  - After clicking the (+) symbol, you should be presented with a pop-up which allows you to enter the name of the deck.
- Two buttons should exist for the deck creation pop-up, a confirm button and a discard button.
  - Clicking the confirm button should present you with an empty deck, with the title visible on top.
  - Clicking the discard button should present you with the "Decks" screen.

Non-functional

None.

User story

Story Identifier: 13

Story Name: Side menu

Estimated time: 5 days

Description

As a student, I want to be able to navigate the application by using a side menu, so that I can navigate the application more quickly.

Confirmation

Functional

- The side menu should be reached when the toolbars' menu button is clicked, or when the user swipes from the edge.
- The side menu should hold all main menu buttons.
- The side menu should hold a "Logout" button.
- The side menu should be closed by clicking outside the menu, or by swiping the menu back to the left.

Non-functional

- The icons for the menu entries should match the corresponding entries in the home menu (if applicable)

User story

Story Identifier: 11

Story Name: Users system

Estimated time: 5 days

Description

As a student, I want the application to have a system for using multiple (local) users and still have our decks separate for each other, so that I can more easily share a device with someone without having to see the other persons decks.

Confirmation

Functional

- If no user is logged in the app should start at a screen where the user can choose/create a user.
- When the profile has been selected, the main menu should appear.
- The picked user should persist between sessions; the app should automatically log the previous user in at application start, unless the user explicitly logs out from the side menu.
- The user should be able to logout using the side menu.
- The user profiles should be persistent between sessions.
- You should be able to delete a user from the start menu.

Non-functional

- Decks and games shall no longer be associated with “MemorizationTrainingTool”, but rather by different Users. Instead, Users should be associated with ”MemorizationTrainingTool”.

User story

Story Identifier: 9

Story Name: Persistent storage of decks

Estimated Time: 2 days

Description

As a student, I want my decks/cards to persist between sessions, so that I can easily practice my decks without needing to rewrite them everytime.

Confirmation

Functional

- The Decks are saved between restarts of the app
- The Notes within a deck should be saved between restarts of the app

Non-functional

None.

User story

Story Identifier: 10

Story Name: Saving statistics

Estimated time: 3 days

Description

As a student, I want my stats to be saved, and saved between sessions, so that I know how im performing on my different decks.

Confirmation

Functional

- Each user should have their own stats page, where they can view various statistics.
- The following stats should be tracked:
    - Total times played
    - Times played, per deck, per game mode
    - High score per deck, per game mode
    - Average score per deck, per game mode
- The stats should be displayable in a stats view
- The stats after a game is played should be saved to the stats object.
- When a deck is removed the stats for that deck should be removed.

Non-functional

None.

User story

Story Identifier: 14

Story Name: Quiz Mode

Estimated Time: 6 days

Description

As a student, I want a multiple choice-type game mode so that I can review my cards in different ways.

Confirmation

Functional

- There should be multiple choice questions, with 4 alternative answers
- The questions should have random answers taken from other answers in the deck
- When an answer is selected the correct answer is shown by being highlighted.
- When an answer is selected the user needs to manually proceed to the next question.
- Like the flashcard game mode, you will see your result in the end.
- The game mode should only be playable on Decks with at least 4 Cards.

Non-functional

- The layout should be similar to the flashcard mode layout.

User story

Story Identifier: 15

Story Name: Achievements

Estimated Time: 6 days

Description

As a student, I want the app to have achievements to give me milestones in my progress, so that I get motivated.

Confirmation

Functional

- When a game is played the achievements should be updated
- There should be different achievements for:
  - Get all answers correct on a deck
  - Played 10 games in total (milestones)
  - Played 20 games in total (milestones)
  - Played 50 games in total (milestones)
  - Played 100 games in total (milestones)
  - Created your first deck
  - Played your first deck
  - Unlock all achievements

Non-functional

- The achievements should be displayed as icons without text.
- When you have received an achievement you should not be able to lose it.
- When an achievement is clicked, the requirement should be visible.

User story

Story Identifier: 2

Story Name: Animated flipping of flashcards

Description

As a student, I want a good looking animation for when I flip a flashcard, so that the application better resembles real life use of flashcards.

Confirmation

Functional

- When the text of the flashcard changes by tapping the card, the change should be animated by rotating the card and the text.

Non-functional

None.

2.2. User interface

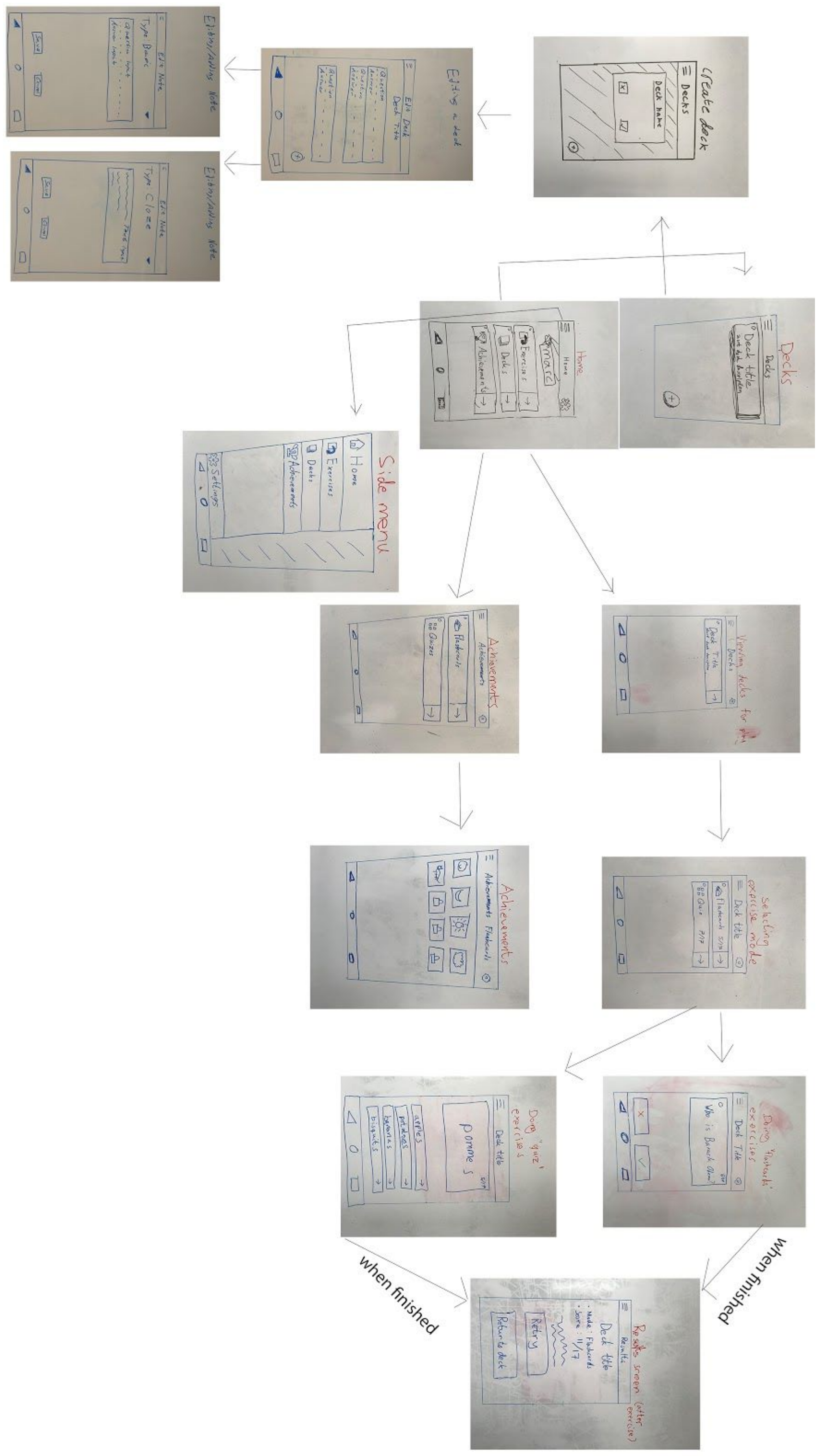


Figure 1: A sketch of the user interface

3. Domain model

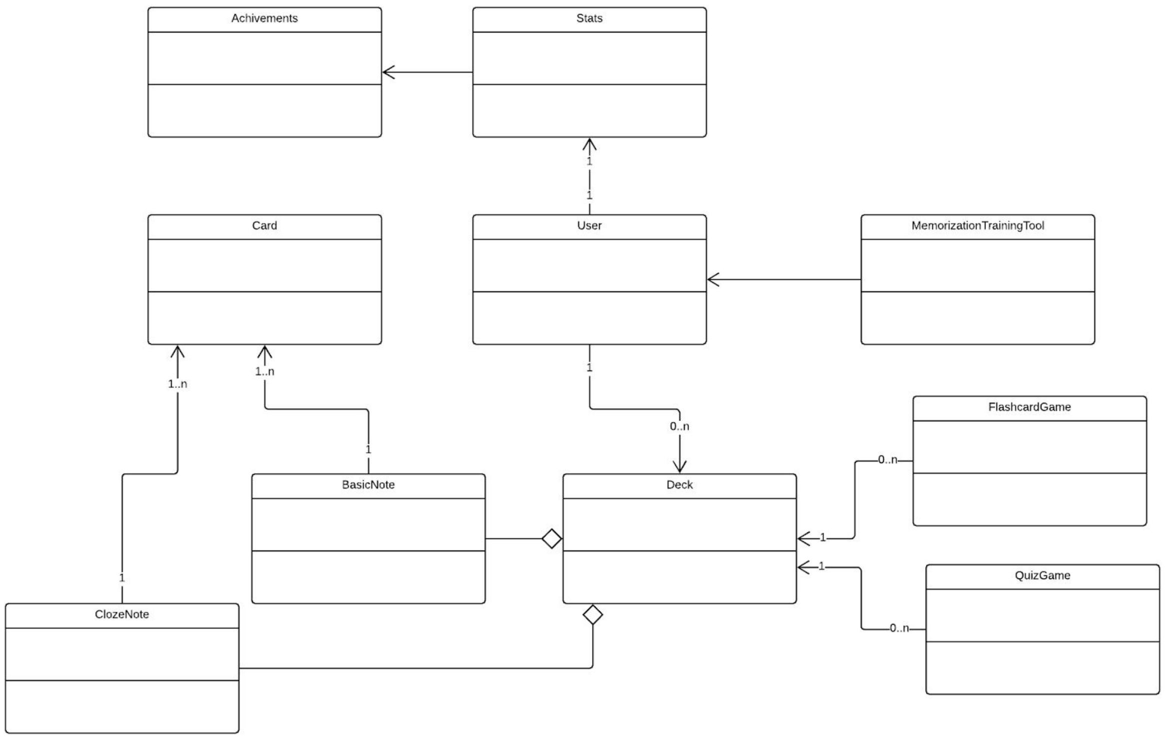


Figure 2: The domain model

3.1. Class responsibilities

The Deck class has the responsibility of holding cards that can be used in a exercise.

The Card class holds talthoalthoalthohe information for a card, which includes text for the the front and the back side of the flashcard.

The BasicNote class holds all the information for a flashcard, IE just one card with a front and a back.
The ClozeNote class holds all the information needed to generate cards with cloze deletions.

The User class holds the information about a user, their name, which decks they have, how they have performed on these decks.
The Stats class holds the statistics for a users, how they performed on their different decks in different game modes.
The Achievement class holds the achievements that the user are able to receive and are responsible for keeping track of which one are completed and which one are not.

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 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 keeping track of all the users and have a sense of which user is active at the moment.

4. 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.

4.1. Subsystem decomposition

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

4.2. 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.

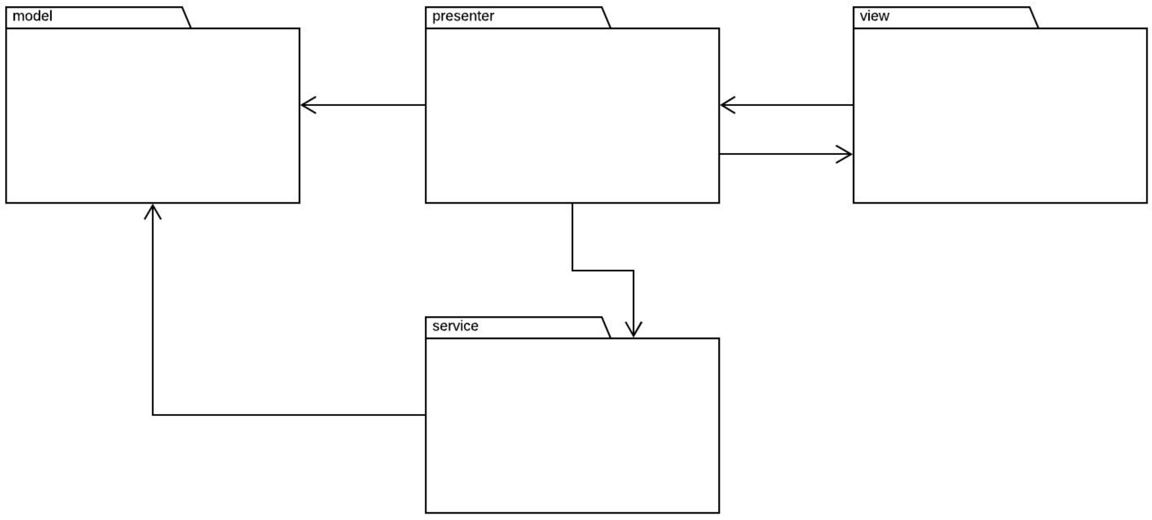
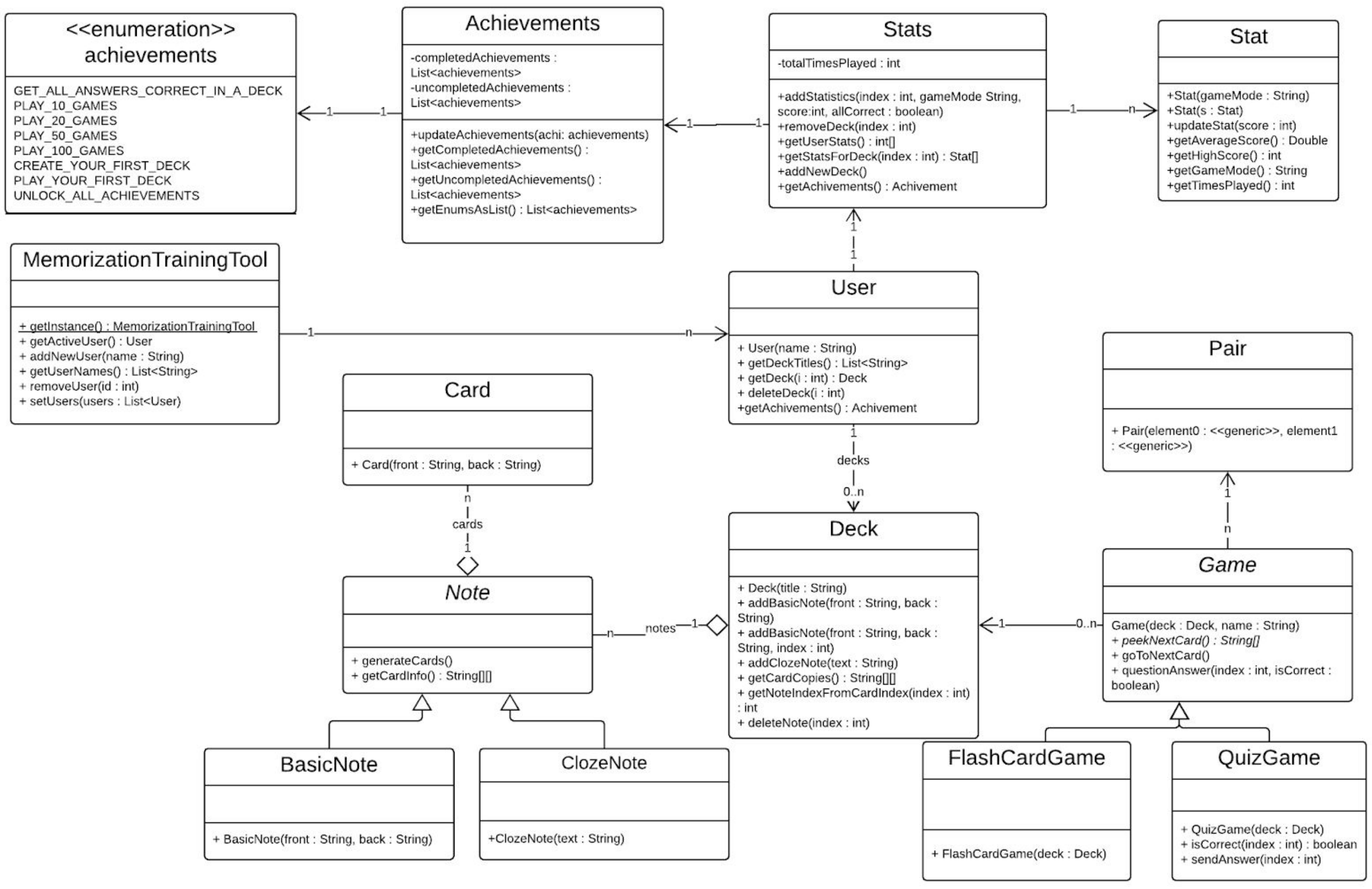


Figure 3: A high-level component diagram

Figure 3 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 figure 3 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.

4.3. Design model/Model package



<<enumeration>> achievements
GET_ALL_ANSWERS_CORRECT_IN_A_DECK
PLAY_10_GAMES
PLAY_20_GAMES
PLAY_50_GAMES
PLAY_100_GAMES
CREATE_YOUR_FIRST_DECK
PLAY_YOUR_FIRST_DECK
UNLOCK_ALL_ACHIEVEMENTS
Achievements
-completedAchievements : List<achievements>
-uncompletedAchievements : List<achievements>
+updateAchievements(achi: achievements)
+getCompletedAchievements() : List<achievements>
+getUncompletedAchievements() : List<achievements>
+getEnumsAsList() : List<achievements>
Stats
-totalTimesPlayed : int
+addStatistics(index : int, gameMode String, score:int, allCorrect : boolean)
+removeDeck(index : int)
+getUserStats() : int[]
+getStatsForDeck(index : int) : Stat[]
+addNewDeck()
+getAchivements() : Achivement
Stat
+Stat(gameMode : String)
+Stat(s : Stat)
+updateStat(score : int)
+getAverageScore() : Double
+getHighScore() : int
+getGameMode() : String
+getTimesPlayed() : int
MemorizationTrainingTool
+ getInstance() : MemorizationTrainingTool
+ getActiveUser() : User
+ addNewUser(name : String)
+ getUserNames() : List<String>
+ removeUser(id : int)
+ setUsers(users : List<User>)
User
+ User(name : String)
+ getDeckTitles() : List<String>
+ getDeck(i : int) : Deck
+ deleteDeck(i : int)
+getAchivements() : Achivement
Pair
+ Pair(element0 : <<generic>>, element1 : <<generic>>)
Card
+ Card(front : String, back : String)
cards
decks
notes
Note
+ generateCards()
+ getCardInfo() : String[][]
Deck
+ Deck(title : String)
+ addBasicNote(front : String, back : String)
+ addBasicNote(front : String, back : String, index : int)
+ addClozeNote(text : String)
+ getCardCopies() : String[][]
+ getNoteIndexFromCardIndex(index : int) : int
+ deleteNote(index : int)
Game
Game(deck : Deck, name : String)
+ peekNextCard() : String[]
+ goToNextCard()
+ questionAnswer(index : int, isCorrect : boolean)
BasicNote
+ BasicNote(front : String, back : String)
ClozeNote
+ClozeNote(text : String)
FlashCardGame
+ FlashCardGame(deck : Deck)
QuizGame
+ QuizGame(deck : Deck)
+ isCorrect(index : int) : boolean
+ sendAnswer(index : int)
1
n
0..n

Figure 4: The design model

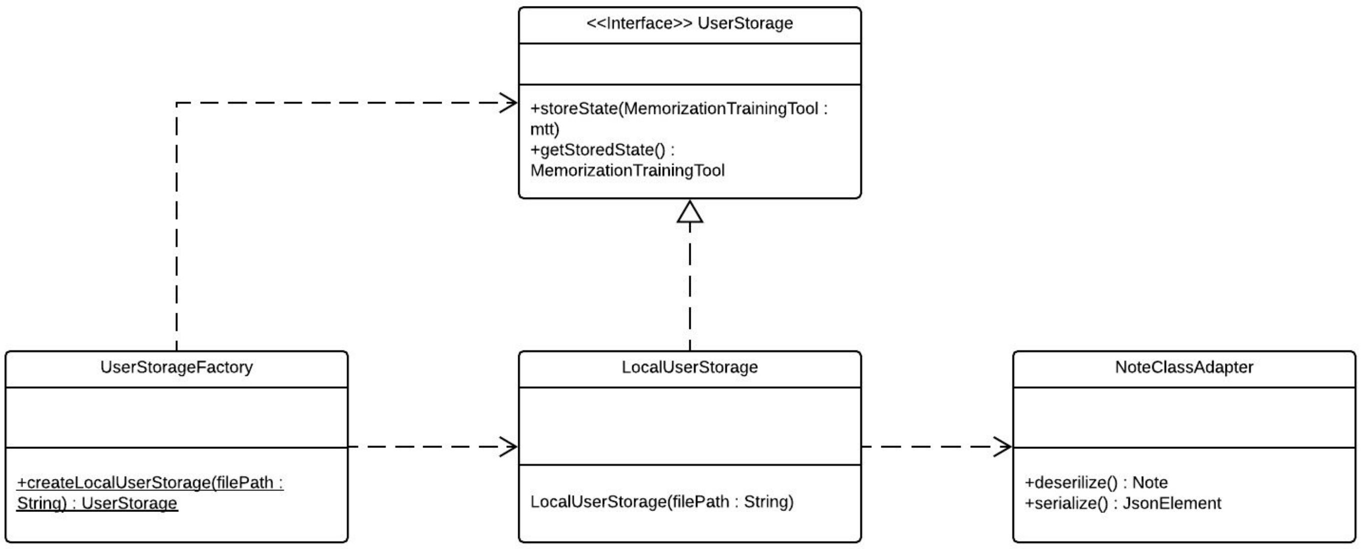
The design model (see figure 4) is derived from the domain model. Where each of the "Boxes" in the domain model has been turned into individual classes, there are also additional classes that has been added to better handle things in an object oriented way. The model package holds the data for the application 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.

4.4. Service Package



<<Interface>> UserStorage
+storeState(MemorizationTrainingTool : mtt)
+getStoredState() : MemorizationTrainingTool
UserStorageFactory
+createLocalUserStorage(filePath : String) : UserStorage
LocalUserStorage
LocalUserStorage(filePath : String)
NoteClassAdapter
+deserilize() : Note
+serialize() : JsonElement

Figure 5: The service package

This package (see figure 5) 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.

4.5. Presenter Package

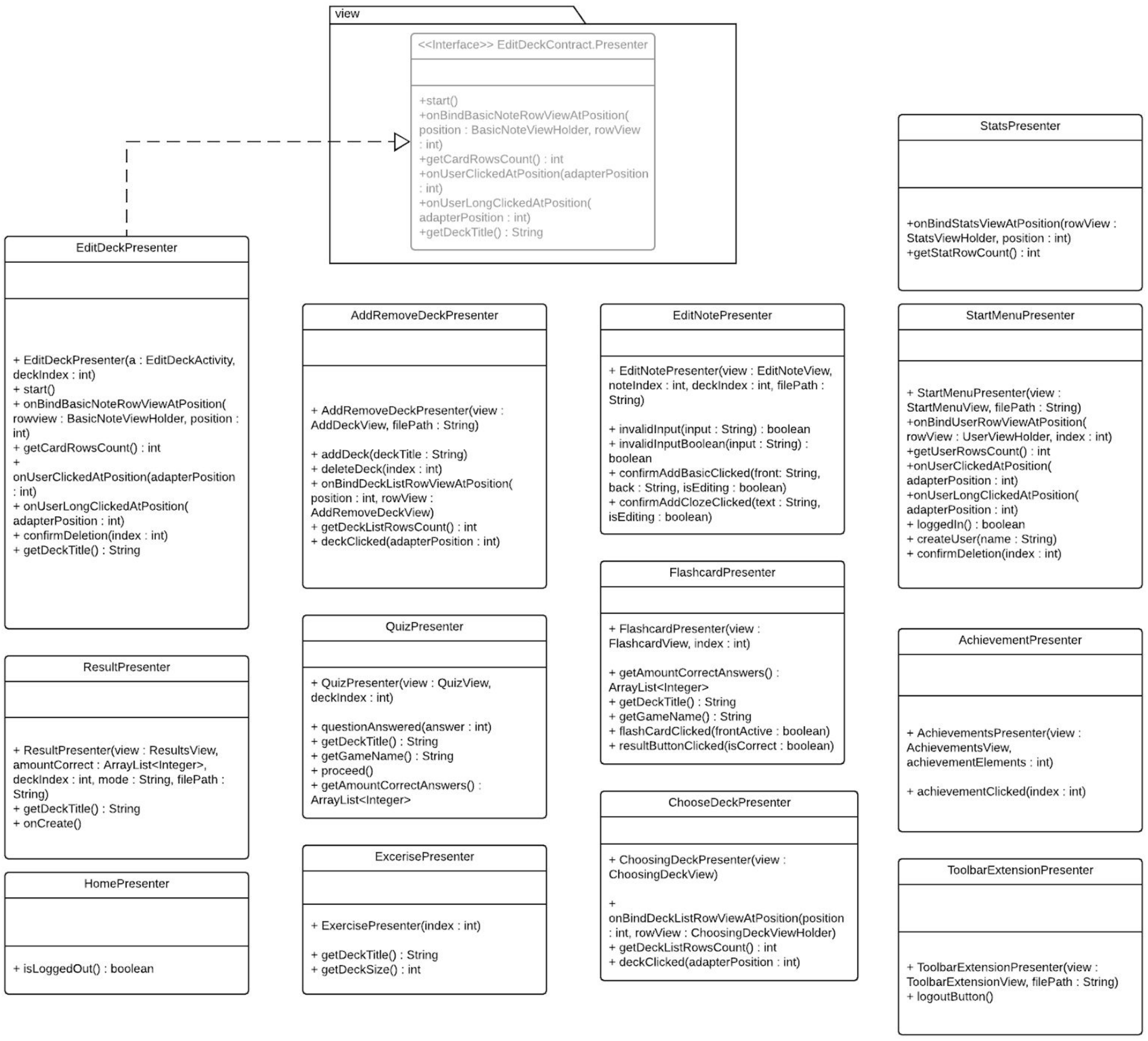
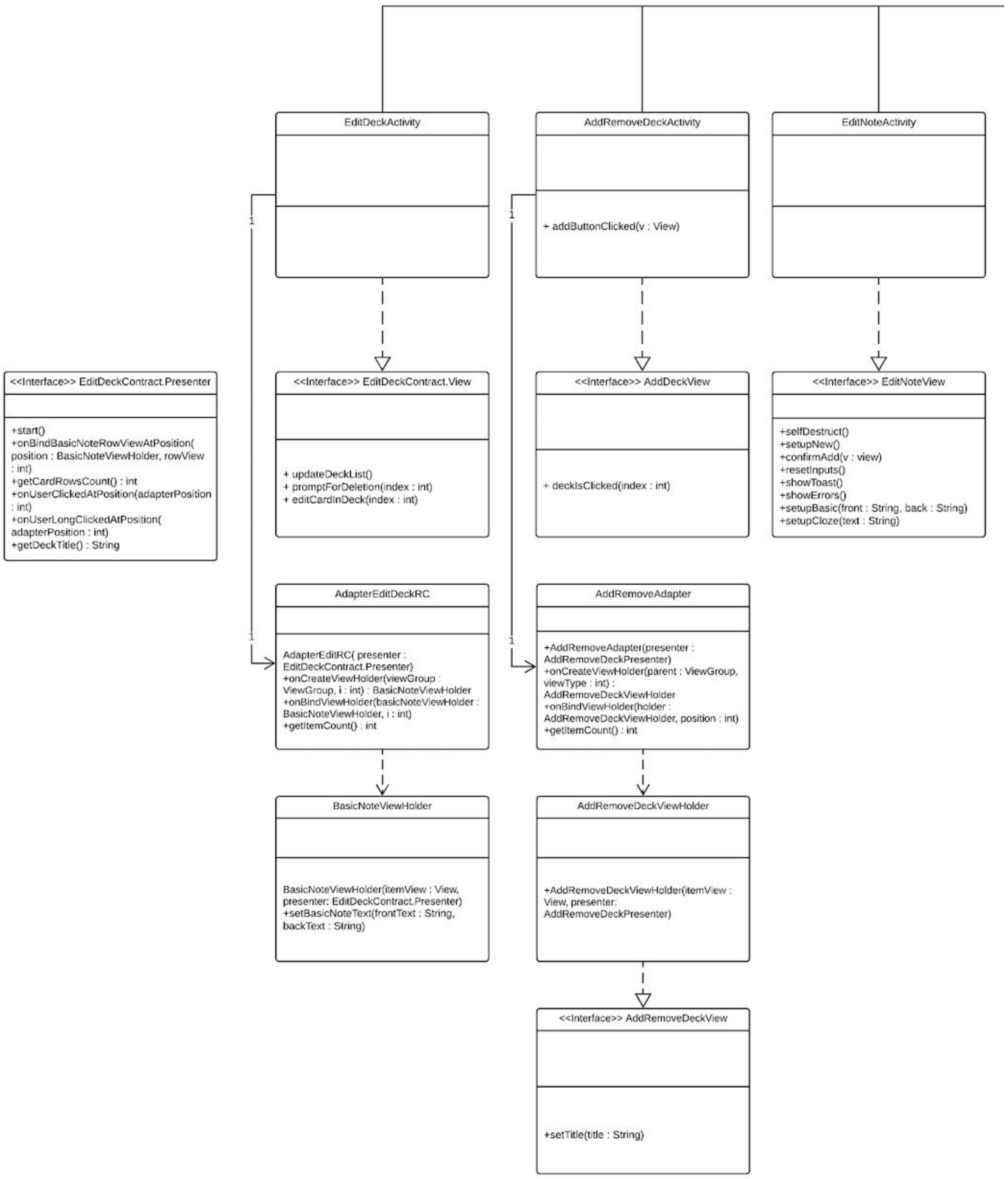
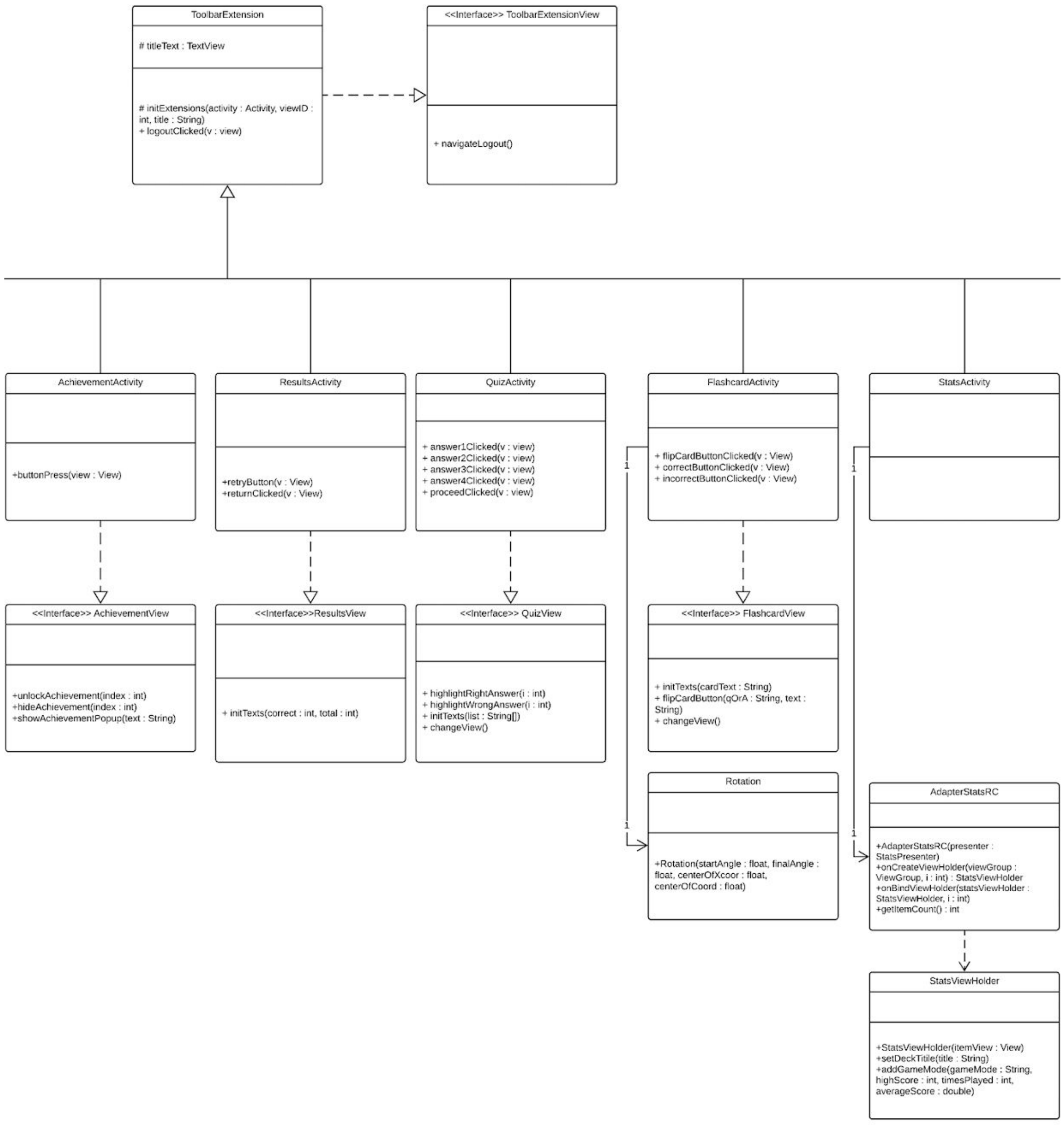

view
<<Interface>> EditDeckContract.Presenter
+start()
+onBindBasicNoteRowViewAtPosition(position : BasicNoteViewHolder, rowView : int)
+getCardRowsCount() : int
+onUserClickedAtPosition(adapterPosition : int)
+onUserLongClickedAtPosition(adapterPosition : int)
+getDeckTitle() : String
EditDeckPresenter
+ EditDeckPresenter(a : EditDeckActivity, deckIndex : int)
+ start()
+ onBindBasicNoteRowViewAtPosition(rowview : BasicNoteViewHolder, position : int)
+ getCardRowsCount() : int
+ onUserClickedAtPosition(adapterPosition : int)
+ onUserLongClickedAtPosition(adapterPosition : int)
+ confirmDeletion(index : int)
+ getDeckTitle() : String
AddRemoveDeckPresenter
+ AddRemoveDeckPresenter(view : AddDeckView, filePath : String)
+ addDeck(deckTitle : String)
+ deleteDeck(index : int)
+ onBindDeckListRowViewAtPosition(position : int, rowView : AddRemoveDeckView)
+ getDeckListRowsCount() : int
+ deckClicked(adapterPosition : int)
EditNotePresenter
+ EditNotePresenter(view : EditNoteView, noteIndex : int, deckIndex : int, filePath : String)
+ invalidInput(input : String) : boolean
+ invalidInputBoolean(input : String) : boolean
+ confirmAddBasicClicked(front: String, back : String, isEditing : boolean)
+ confirmAddClozeClicked(text : String, isEditing : boolean)
StatsPresenter
+onBindStatsViewAtPosition(rowView : StatsViewHolder, position : int)
+getStatRowCount() : int
StartMenuPresenter
+ StartMenuPresenter(view : StartMenuView, filePath : String)
+onBindUserRowViewAtPosition(rowView : UserViewHolder, index : int)
+getUserRowsCount() : int
+onUserClickedAtPosition(adapterPosition : int)
+onUserLongClickedAtPosition(adapterPosition : int)
+ loggedIn() : boolean
+ createUser(name : String)
+ confirmDeletion(index : int)
ResultPresenter
+ ResultPresenter(view : ResultsView, amountCorrect : ArrayList<Integer>, deckIndex : int, mode : String, filePath : String)
+ getDeckTitle() : String
+ onCreate()
QuizPresenter
+ QuizPresenter(view : QuizView, deckIndex : int)
+ questionAnswered(answer : int)
+ getDeckTitle() : String
+ getGameName() : String
+ proceed()
+ getAmountCorrectAnswers() : ArrayList<Integer>
FlashcardPresenter
+ FlashcardPresenter(view : FlashcardView, index : int)
+ getAmountCorrectAnswers() : ArrayList<Integer>
+ getDeckTitle() : String
+ getGameName() : String
+ flashCardClicked(frontActive : boolean)
+ resultButtonClicked(isCorrect : boolean)
AchievementPresenter
+ AchievementsPresenter(view : AchievementsView, achievementElements : int)
+ achievementClicked(index : int)
HomePresenter
+ isLoggedOut() : boolean
ExcerisePresenter
+ ExercisePresenter(index : int)
+ getDeckTitle() : String
+ getDeckSize() : int
ChooseDeckPresenter
+ ChoosingDeckPresenter(view : ChoosingDeckView)
+ onBindDeckListRowViewAtPosition(position : int, rowView : ChoosingDeckViewHolder)
+ getDeckListRowsCount() : int
+ deckClicked(adapterPosition : int)
ToolbarExtensionPresenter
+ ToolbarExtensionPresenter(view : ToolbarExtensionView, filePath : String)
+ logoutButton()

Figure 6: The presenter package

The presenter package (see figure 6) 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.

4.6. View Package


EditDeckActivity
AddRemoveDeckActivity
+ addButtonClicked(v : View)
EditNoteActivity
1
1
<<Interface>> EditDeckContract.Presenter
+start()
+onBindBasicNoteRowViewAtPosition(position : BasicNoteViewHolder, rowView : int)
+getCardRowsCount() : int
+onUserClickedAtPosition(adapterPosition : int)
+onUserLongClickedAtPosition(adapterPosition : int)
+getDeckTitle() : String
<<Interface>> EditDeckContract.View
+ updateDeckList()
+ promptForDeletion(index : int)
+ editCardInDeck(index : int)
<<Interface>> AddDeckView
+ deckIsClicked(index : int)
<<Interface>> EditNoteView
+selfDestruct()
+setupNew()
+confirmAdd(v : view)
+resetInputs()
+showToast()
+showErrors()
+setupBasic(front : String, back : String)
+setupCloze(text : String)
1
1
AdapterEditDeckRC
AdapterEditRC(presenter : EditDeckContract.Presenter)
+onCreateViewHolder(viewGroup : ViewGroup, i : int) : BasicNoteViewHolder
+onBindViewHolder(basicNoteViewHolder : BasicNoteViewHolder, i : int)
+getItemCount() : int
AddRemoveAdapter
+AddRemoveAdapter(presenter : AddRemoveDeckPresenter)
+onCreateViewHolder(parent : ViewGroup, viewType : int) : AddRemoveDeckViewHolder
+onBindViewHolder(holder : AddRemoveDeckViewHolder, position : int)
+getItemCount() : int
BasicNoteViewHolder
BasicNoteViewHolder(itemView : View, presenter: EditDeckContract.Presenter)
+setBasicNoteText(frontText : String, backText : String)
AddRemoveDeckViewHolder
+AddRemoveDeckViewHolder(itemView : View, presenter: AddRemoveDeckPresenter)
<<Interface>> AddRemoveDeckView
+setTitle(title : String)


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

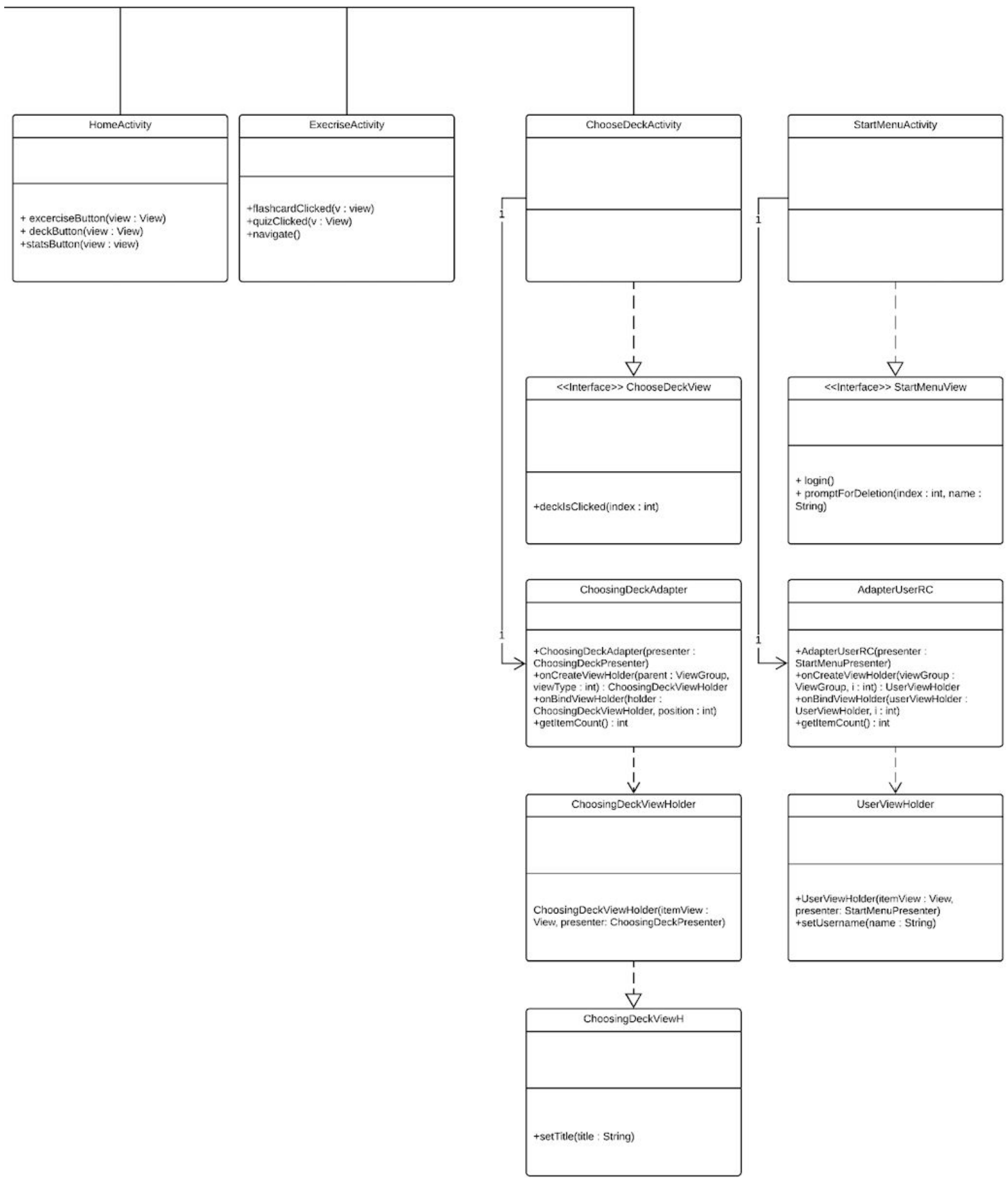

HomeActivity
+ excerciseButton(view : View)
+ deckButton(view : View)
+statsButton(view : view)
ExecriseActivity
+flashcardClicked(v : view)
+quizClicked(v : View)
+navigate()
ChooseDeckActivity
StartMenuActivity
1
1
<<Interface>> ChooseDeckView
+deckIsClicked(index : int)
<<Interface>> StartMenuView
+ login()
+ promptForDeletion(index : int, name : String)
1
1
ChoosingDeckAdapter
+ChoosingDeckAdapter(presenter : ChoosingDeckPresenter)
+onCreateViewHolder(parent : ViewGroup, viewType : int) : ChoosingDeckViewHolder
+onBindViewHolder(holder : ChoosingDeckViewHolder, position : int)
+getItemCount() : int
AdapterUserRC
+AdapterUserRC(presenter : StartMenuPresenter)
+onCreateViewHolder(viewGroup : ViewGroup, i : int) : UserViewHolder
+onBindViewHolder(userViewHolder : UserViewHolder, i : int)
+getItemCount() : int
ChoosingDeckViewHolder
ChoosingDeckViewHolder(itemView : View, presenter: ChoosingDeckPresenter)
UserViewHolder
+UserViewHolder(itemView : View, presenter: StartMenuPresenter)
+setUsername(name : String)
ChoosingDeckViewH
+setTitle(title : String)

Figure 7, 8, 9: The view package UML diagram (the full diagram can be found in the repository in Documentation/uml)

The view package (see figure 7,8,9) 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.

4.7. Code Quality

4.7.1. 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.

4.7.2. 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.

4.7.3. 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.

4.8. 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. Although 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.

4.9. 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 figure 4 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.

4.10. 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.

4.11. Concurrency issues

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

4.12. 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.

4.12.1. Checking stats

Checking your statistics as a user:

1. Click on "Statistics"
2. Look up the deck you would like to see your statistics on.

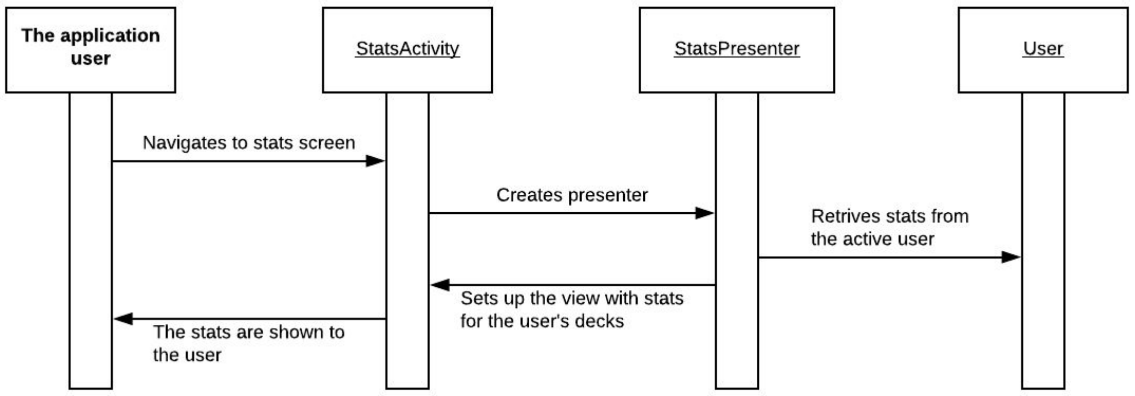

The application user
StatsActivity
StatsPresenter
User
Navigates to stats screen
Creates presenter
Retrives stats from the active user
Sets up the view with stats for the user's decks
The stats are shown to the user

Figure 10: The UML sequence diagram for checking one's statistics as a user

The user of the application (see the figure 10, 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.

4.12.2. 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

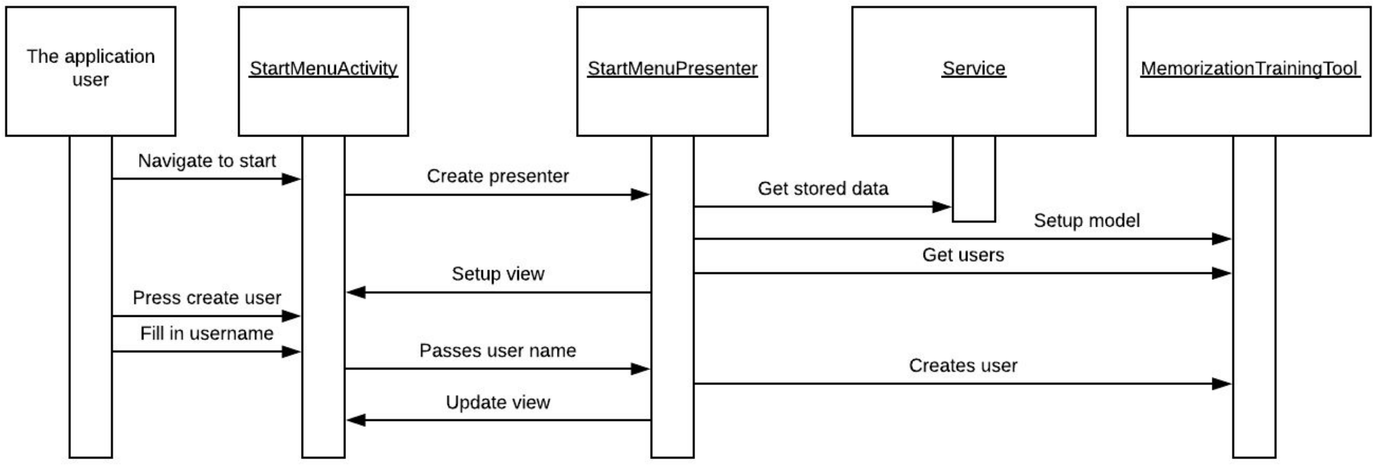


Figure 11: The UML sequence diagram for creating a user

The users enters (see figure 11) the StartMenuActivity which creates a StartMenuPresenter, the presenter retrieves all the user information from the persistent storage service and updates the model with this information. The presenter then retrieves the 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

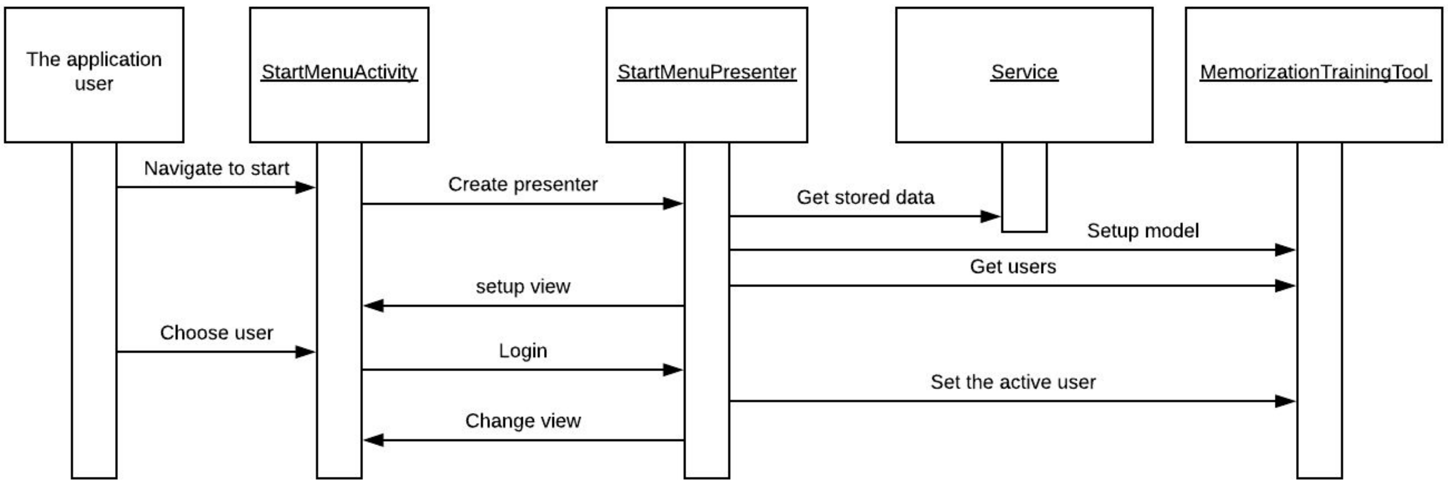


Figure 12: The UML sequence diagram for login into a user

The users enters (see figure 12) the StartMenuActivity which creates a StartMenuPresenter, the presenter retrieves all the user information from the persistent storage service and updates the model with this information. The presenter then retrieves the 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.

4.12.3. 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.

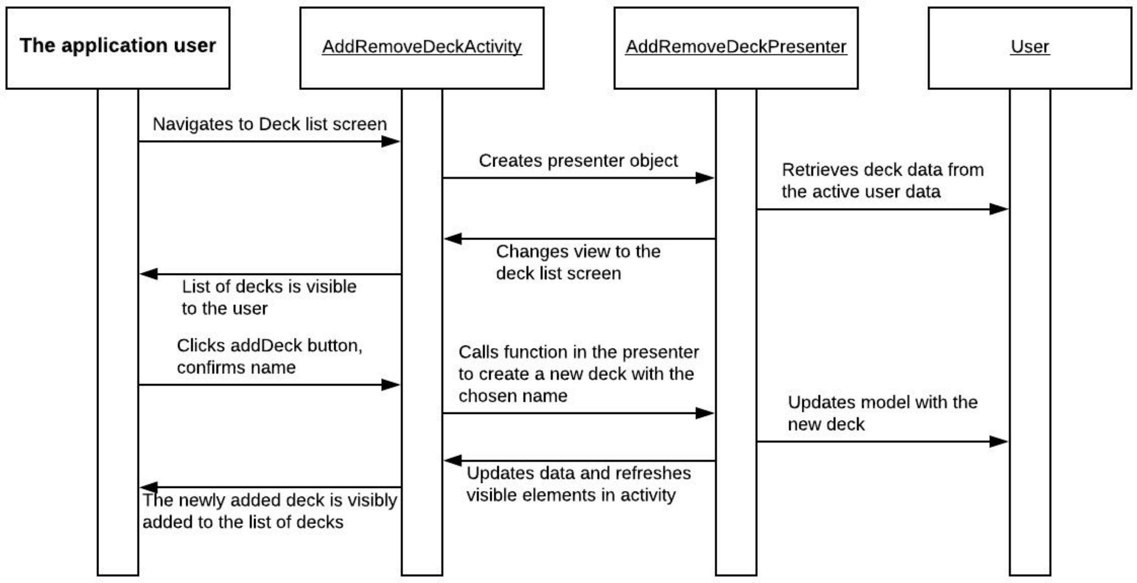

The application user
AddRemoveDeckActivity
AddRemoveDeckPresenter
User
Navigates to Deck list screen
Creates presenter object
Retrieves deck data from the active user data
Changes view to the deck list screen
List of decks is visible to the user
Clicks addDeck button, confirms name
Calls function in the presenter to create a new deck with the chosen name
Updates model with the new deck
Updates data and refreshes visible elements in activity
The newly added deck is visibly added to the list of decks

Figure 13: The UML sequence diagram for creating a new deck as a user

When the user enters (figure 13) 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.

5. Persistent data management

5.1. 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*.

5.2. 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.

6. 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.

7. Peer review

Does the project use a consistent coding style?

Yes, the names of methods and variables are consistent. For example if some class has something to do with mails, the methods and variables have “something”mail most of the times, which makes it clear that they belong to that class.

Is the code reusable?

Some parts of the model allows for good re-usage of existing code. Adding new account types could reuse a lot of code. By simply adding a new enum in the interface Account, the code in the class AccountBuilder would be heavily reusable.

Is it easy to maintain?

There are some good examples of code that is easy to maintain. For example if Gmail changes the way that their emails are sent there only needs to be changes done to a few places in the code, which handles sending and retrieving of emails.

Can we easily add/remove functionality?

Additional functionality could be added easily, due to the many interfaces used throughout the project. Removing functionality is a more difficult question. The difficulty depends heavily on what you want to remove. One well designed area is how the controller package is implemented with a hierarchy of controllers, with “bigger” ones on top. For example, the settings menu of the application has its own controller hierarchy. This makes adding or removing features easier, as they divided into clear sections with their own responsibilities.

Are design patterns used?

There are some design patterns used in the program. The model is using the BuilderPattern, which is used to prevent the need for using constructors with many parameters; this makes the creation of objects easier to understand. There is also an implementation of the MVC architectural pattern where the model does not have any dependency on things outside of the model.

Is the code documented?

Many functions and classes are not documented which makes reviewing and following the code harder. For example, DatabaseService has several methods, while only a few of them are documented.

Are proper names used?

We believe that the names of most classes, functions and variables have been properly assigned as they describe the usage quite well. The method createAccountButtonAction(), in the class AddAccountController, has good naming for both the method itself, and for its local variables.

Is the design modular? Are there any unnecessary dependencies?

It is rather obvious that the group have not ran any type of code inspector. The project contains about 26 unused imports which creates unnecessary dependencies.

Does the code use proper abstractions?

No abstract classes has been used in the project. If these are needed, we are not entirely sure. Interfaces are widely used throughout the whole project. Although, many methods in these interfaces are simply mentioned in the implementing classes, but they do not contain any functionality. These methods are also never being used. This violates the “Interface Segregation Principle”.

Is the code well tested?

The code is tested with 51% line coverage, some packages are tested more than others which is fine. From what is described as the model package in the SDD, although not clear in the code itself, these parts of the code are tested with close to 100% line coverage. For the Controller component, there exists some framework used to create tests. The classes in the Controller component have been tested to some extent, though there seems to exist some inconsistency with the tests or some race conditions in the code, causing them to fail occasionally.

Are there any security problems, are there any performance issues?

The application uses accounts, which uses usernames and passwords to authenticate users. Passwords are stored in a database in plain text, meaning that anyone with access to the database file(s) can see and retrieve the values as normal strings. This is a security issue as sensitive data should not be so easily available -- better would be to first salt and hash the passwords. The current implementation of the application, however, requires the passwords to be in plain text form. As the functionality for actually sending mails doesn't work yet, we couldn't fully investigate the performance of the application. From testing the application as-is, no performance issues could be found.

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

It is pretty hard to understand what part of the code does what, because of a lack of a good folder structure. It is hard to identify the model of the project. However, there is a controller folder, so they are easy to find. The in-code documentation is lacking overall which makes the code hard to understand, unless one reads the documentation in the SDD. The project follows the MVC structure suggested by Oracle for using JavaFX with FXML files. There is not a clear model package though, which makes it difficult to follow the MVC structure. The model is isolated from the other parts though, which is good.

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

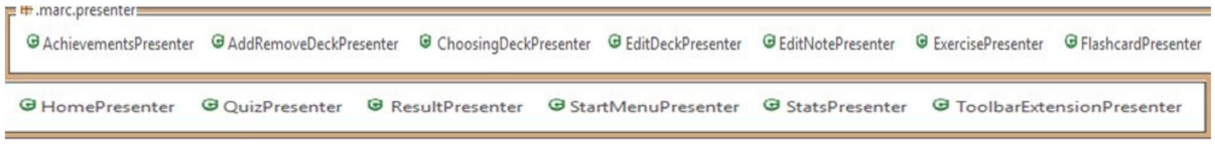
The analysis tool in IntelliJ found 564 warnings, 23 errors, and 217 typos, many of which could be easily corrected. Correcting this would improve the code. There are also classes that implement interfaces and their methods, but the methods doesn't contain any functionality. Removing unused methods would be a design improvement.

8. References

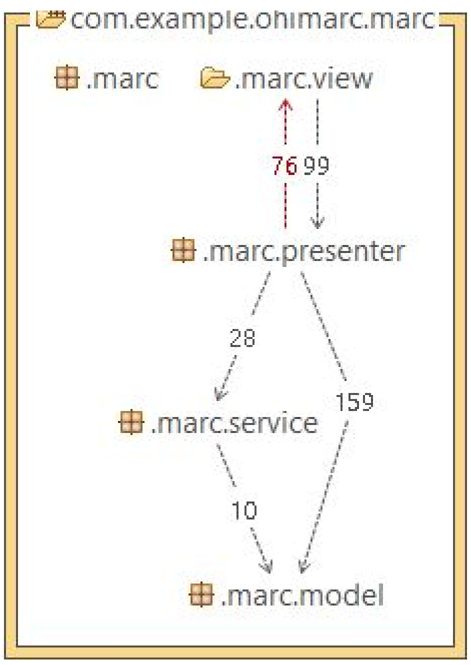
1. Jetbrains.com. (2018). *Code Inspection - Help | IntelliJ IDEA*. [online] Available at: https://www.jetbrains.com/help/idea/2018.1/code-inspection.html [Accessed 19 Oct. 2018].
2. Maxwell, E. (2017). *MVC vs. MVP vs. MVVM on Android*. [online] Academy.realm.io. Available at: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/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

9. Appendix

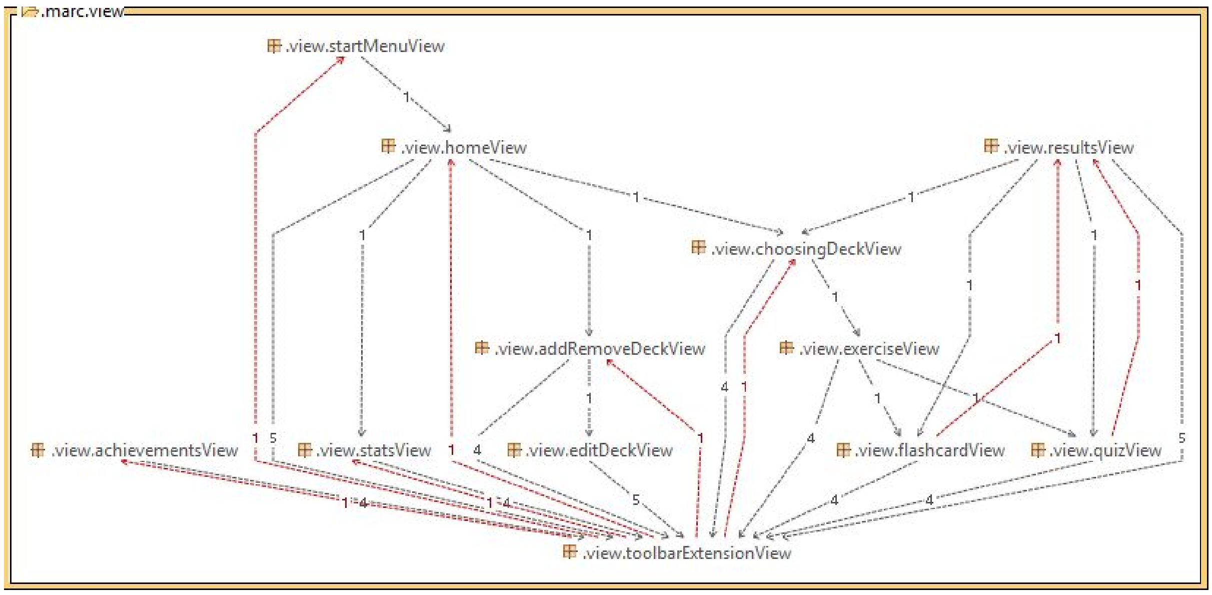
appendix 1:



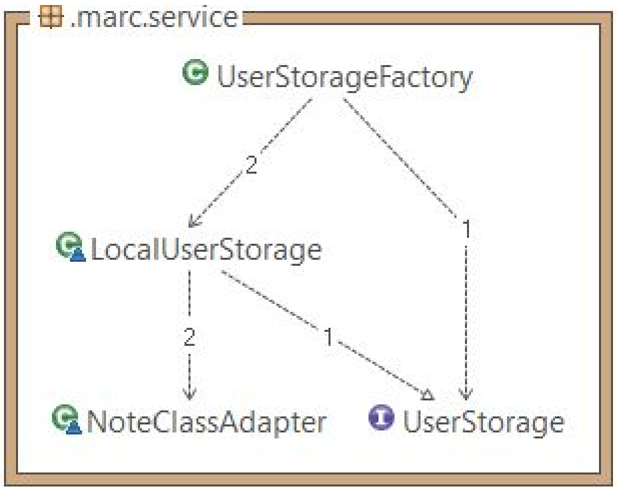
appendix 2:



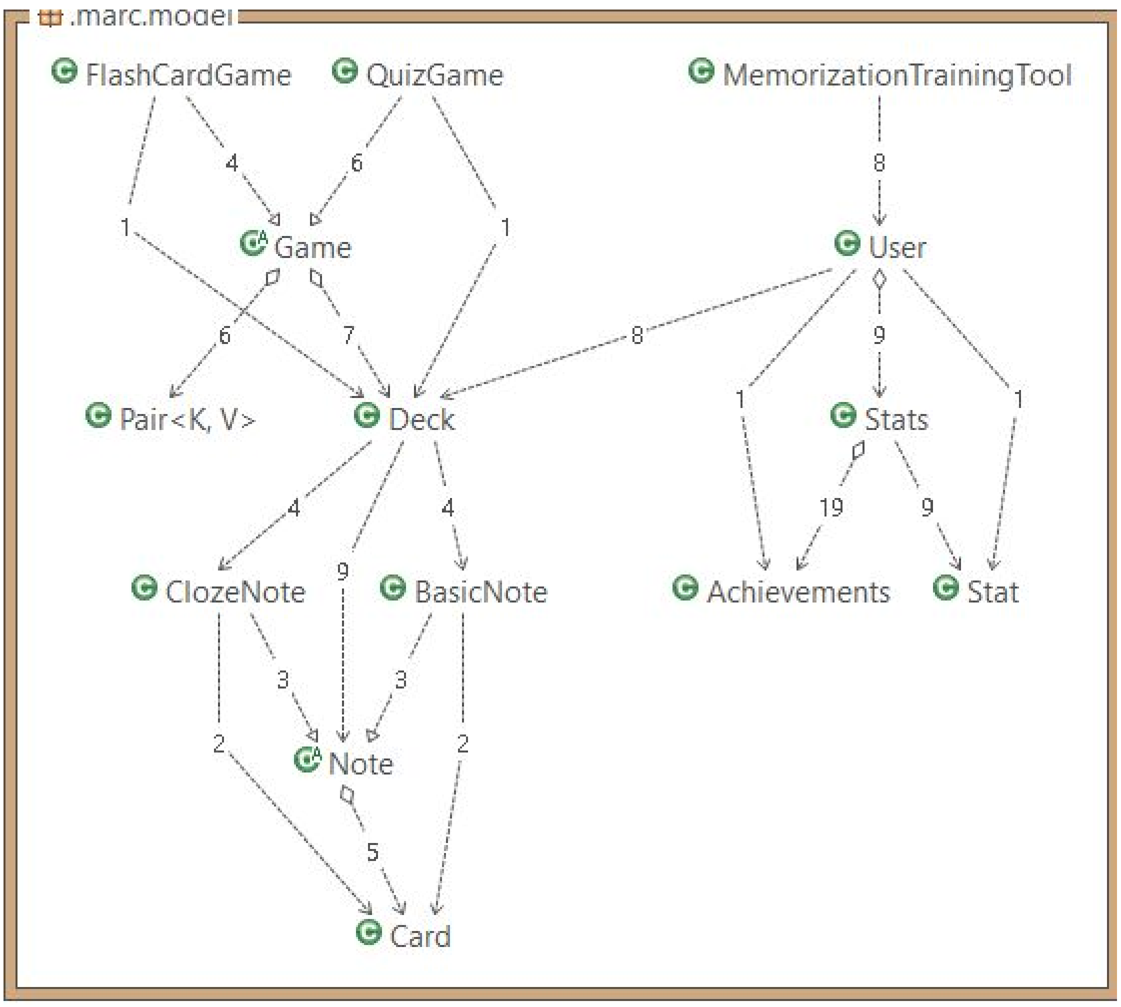
appendix 3:



appendix 4:



appendix 5:


.marc.model
FlashCardGame
QuizGame
MemorizationTrainingTool
Game
User
Pair<K, V>
Deck
Stats
ClozeNote
BasicNote
Achievements
Stat
Note
Card