System Design Document for Habba

Grupp 15

**Johannes Gustavsson, Nils-Martin Robeling, Li Rönning,
Alexander Selmanovic, Jian Shin, Camilla Söderlund**

Objektorienterat Programmeringsprojekt TDA367
Chalmers tekniska högskola
Sverige
October 2018

1 Introduction

With a schedule that can vary greatly from day to day either as a student, a businessman or as a forgetful person it can be hard too keep or create new habits.

This android application will keep track of habits the user wants to create and keep. By creating a habit to do either daily, weekly or monthly and having it to show up in the app on those days, you can easily keep track and remember to do them. It uses notifications that are set by the user, to help them remember when to do their tasks even if you don't have the app running. The app also helps motivate the user with the help of streaks and achievements.

1.1 Definitions, acronyms, and abbreviations

- **User**: A user is a person that has an account for the application and therefore can log in and use the application.
- **Habit**: A habit is an action that the user wants to repeat regularly and keep track of. For example a habit could be washing your face every morning or watering the plants once every week.
- **Group**: A user can be a part of a group of users. In this group they have shared habits. For example if the users want to work out two times a week together they can use groups and keep track of each other and stay motivated.
- **Streak**: When a user has done a habit five days in a row, the user will receive a streak next to the habit on the front page. The streak represents the number of times in a row the user has completed a habit. The streak is visible on the front page until the day the user forgets to do the habit.
- **Achievement**: Users can unlock achievements when they use the application, there are two different types of achievements:
  - **Streak achievement**: when a user has performed a habit a number of times in a row they unlock streak achievements. For example doing a habit ten days in a row.
  - **Number of habits achievement**: When the user has created a number of habits they unlock this kind of achievement. For example having a total of five habits.

2 System architecture

In this section the system architecture and necessary knowledge of the application will be presented if the project is picked up by a new programming team or member.

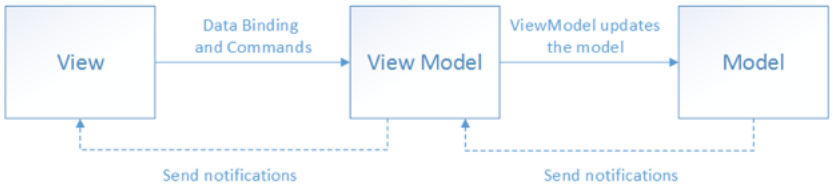
2.1 Android Basics

The application is developed and tested in Android studio with SDK28 and built in Gradle. It supports API28 and added support for lower APIs are to come in a later stage. This choice was made to as quick as possible get a working application to present with the most important functionality implemented.

2.2 Project structure

The system is constructed according to the Model-View-ViewModel (MVVM) pattern, an overview of the pattern can be seen in figure 1 below .

2.3 About MVVM



Figur 1: MVVM

This pattern is similar to the MVC, but the controller is replaced with a View Model. The modules can be single classes or packages with several classes.

2.3.1 The Model

The model encapsulates the applications data and is invisible to the user. It should represent the domain model of the project and be completely independent of the view and view model.

2.3.2 The View

The view is responsible of what the user sees on screen, meaning structure, layout and appearance. Each part of the View is defined in an XML document which does not contain any business logic. The view communicates with only the view model through commands or events like buttons and selecting an item.

2.3.3 The View Model

The view model holds properties and commands to which the view can data-bind (the process of binding the UI together with the business logic) to. The ViewModel should be holding all the functionality the view displays but the view should choose how to display it. Another responsibility introduced to view model is to keep track of any interactions with the model the view might request, the view model may then send data back or expose the model to the view (the model should support data-binding with the view in this case).

2.3.4 Benefits using MVVM

- Unit tests can be written for the model and view model without having to depend on the view and still behaving as if the view was there. [1]
- if there is an existing implementation of the model that encapsulates present logic it could prove either difficult or risky to change it. In this event, the view model can exist as an adapter for the model to avoid making any major changes.
- The view can be changed or completely re-implemented without touching the code since it is all XML.
- It's easy to work with parallel as a team since GUI-design can happen at the same time as development of the view model and model.

2.4 MVVM in the application

In this section you will get an short overview of how the application have implemented the MVVM pattern.

2.4.1 Model

The model package is based on the applications domain model. It contains classes for users, habits, groups, achievements and the application. It is also independent of other packages and of android specific implementations.

2.4.2 View

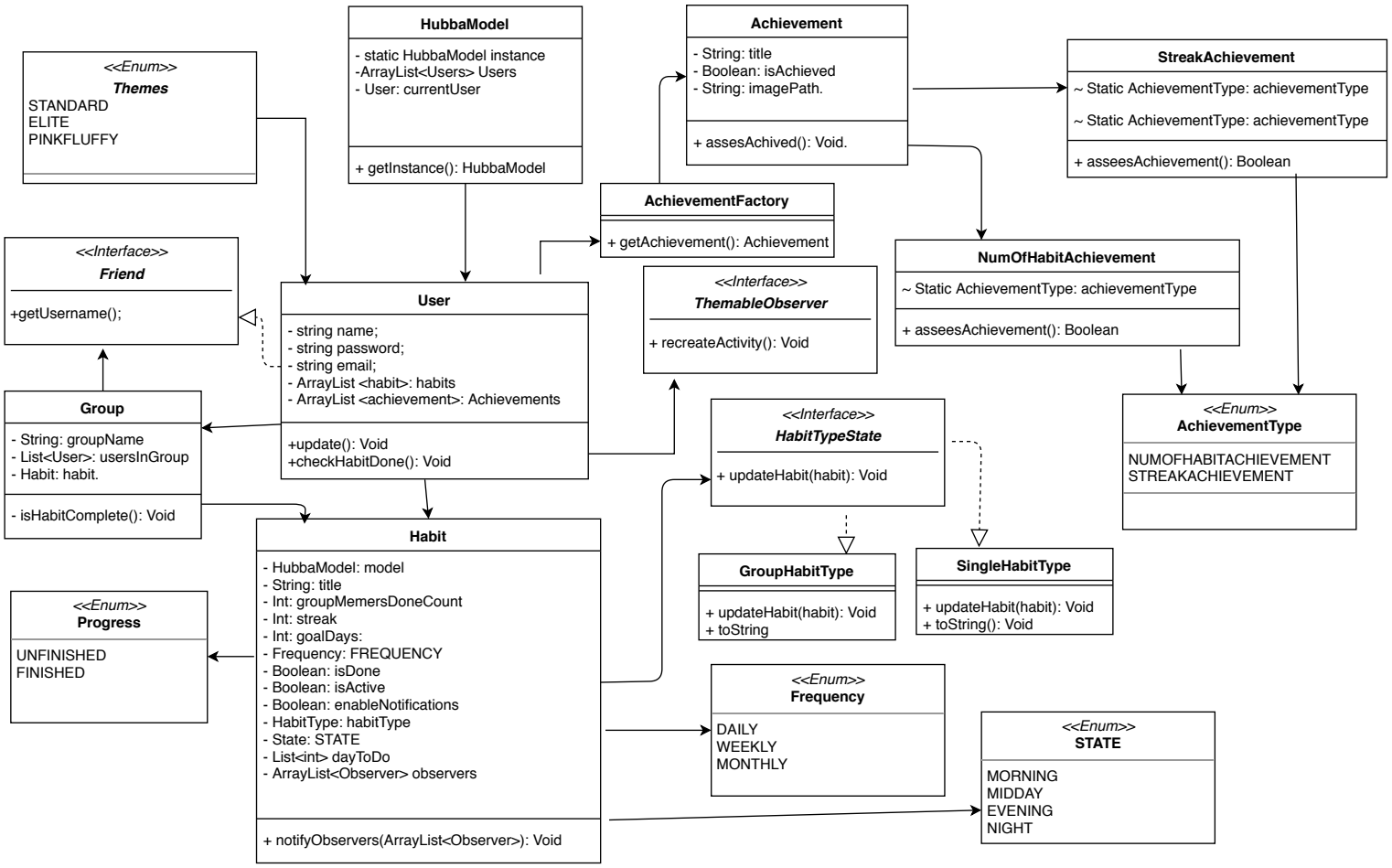
The view package contains all the xml files. The xml files contains names for the parts the user can interact with and builds upp the graphical user interface. This part of the application tells the ViewModel what has happened and then the ViewModel handles that information.

2.4.3 ViewModel

The ViewModel package contains all files that keep state for the View and also updates the model.

It also receives notifications from the model to update the view when something has changed. When it receives these notifications it sends a notification to the view so that the view updates itself accordingly.

2.5 Design model



Figur 2: Design model for Hubba

2.6 Subsystem decomposition

State pattern lets the object change between different states, changing the functionality of the object with the state. [2] In the application the pattern lets habit change between (currently) two states, which could be easily extended to several more.

Iterator pattern provides a way to access objects in for example a list, without exposing the underlying pattern [3]. In this case the built in Iterator is used in MainActivityVM to iterate through lists with strings of the habit titles. This is to not cause any exceptions when the lists are changing while being run through.

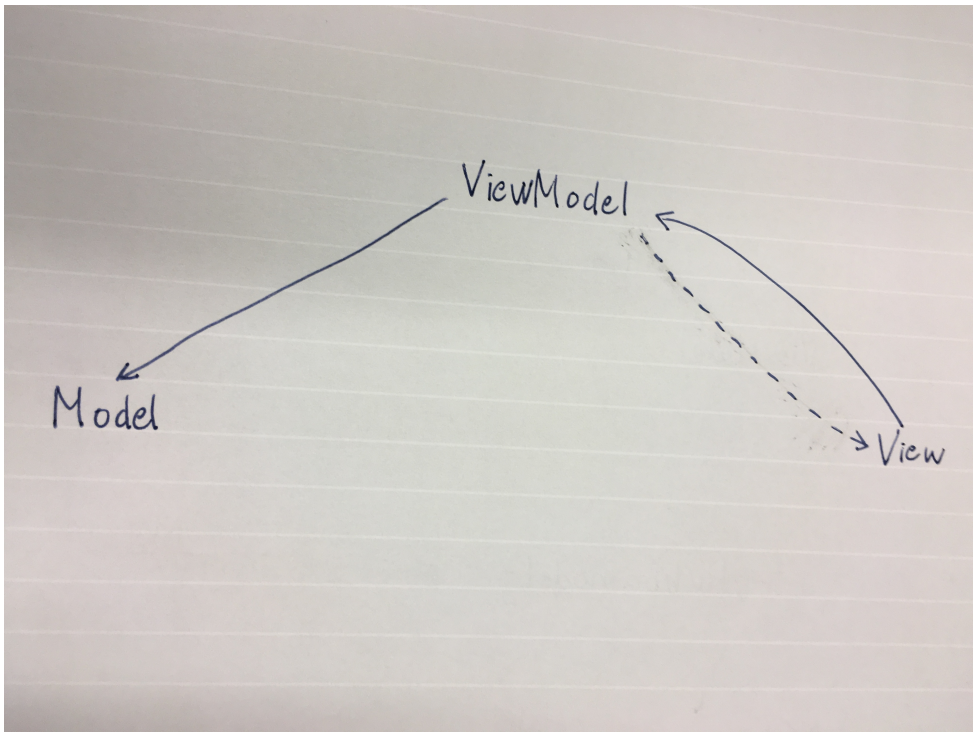
Adapter patterns allows clients that otherwise could not work together to do so by converting one of them into something that the other expects. Here it is used to load the habit list items

into the main page [4].

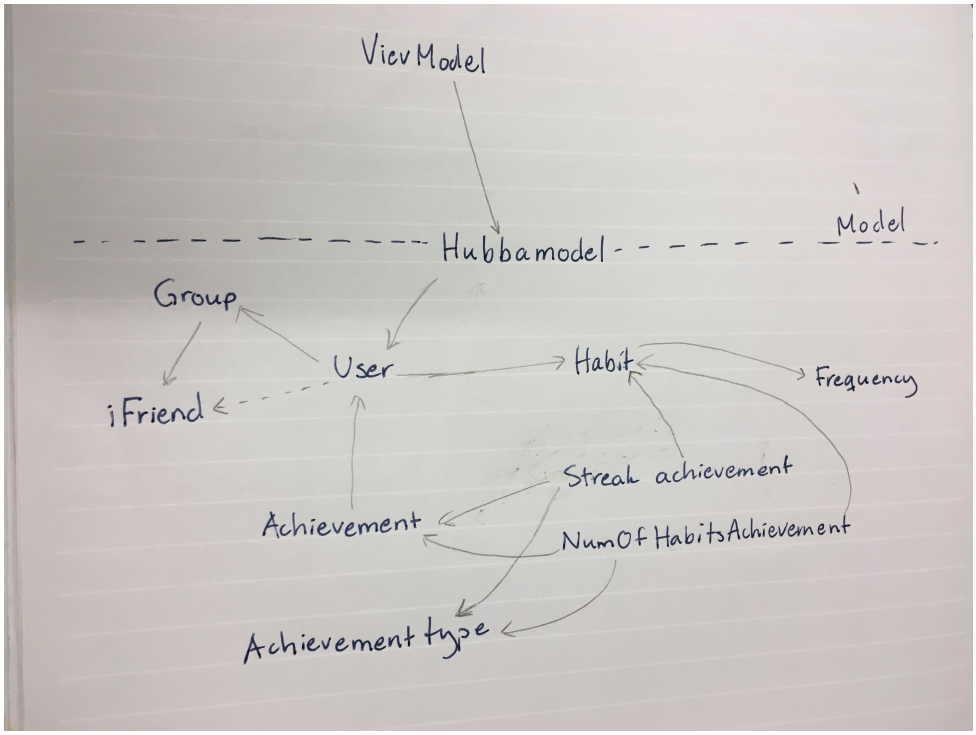
The observer pattern is used for one-to-many dependencies between a subject and observers [5]. By having all observers implementing an interface the subject can treat all observers as one type and call the methods existing in the interface on all of them when it have been updated. This is used when the app switches themes. Since all activities need to restart for the theme to be implemented we make all activities that change theme ThemableObservers. If the theme is changed the method `recreate()` from the interface will be called in all ThemableObservers.

HubbaModel is a singleton class. The singleton pattern is used to ensure that only one instance of a class is created [6]. This pattern is used since HubbaModel is a facade for the rest of the classes in the model. This makes HubbaModel the main way to talk with the model. If it was possible for the HubbaModel to have more than one instance, there would be more than one representation of the application which would be both confusing and very bad.

2.6.1 Stan



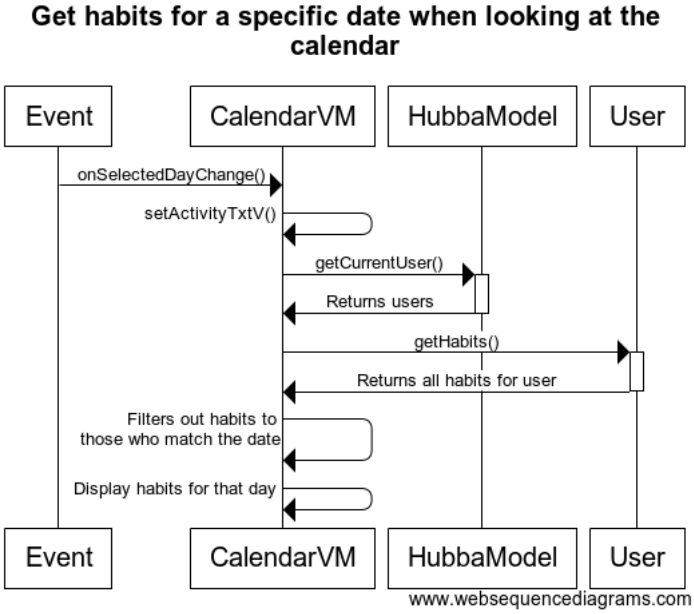
Figur 3: A diagram over the package structure



Figur 4: A diagram showing the structure of the model

2.6.2 UML sequence diagram

To make sure that no unnecessary dependency was created, a sequence diagram was made. In figure 5 it is illustrated how the commands go through the classes and code when habits are shown when the user chooses a specific date in the calendar view.



Figur 5: Diagram over the flow of display a habit in the calendar

2.6.3 Test

`AchievementTest` - This testClass contains the tests `testStreakAchievement`, `testStreakAchievementFalse`, `testNumOfHabitsAchievementFalse` and `testNumOfHabitsAchievement`.

`GroupTest` - This testClass contain the test `testCreateGroup`

`HabitTest` - This testClass contains the tests `testCreateHabit`, `testSetDone`, `testSetHabitTypeStateGroup` and `testSetHabitTypeStateSingle`.

`HubbaModelTest` - This testClass contains the tests `testGetInstance`, `testGetUser`, `testGetCurrentUser`, `testGetUsers` and `testThemeChange`.

`UserTest` - This testClass contains the tests `testCreateUser`, `testAddHabit`, `testRemoveHabit`, `testAddFriend` and `testRemoveFriend`.

3 Persistent data management

The application stores data with the help of `SharedPreferences` and the editor in `SharedPreferences` that stores the data in the form of strings into a file with he help of `JSONObject` and `JSONArray`. The data is loaded with the help of `SharedPreferences` and in most cases google's Gson library, `JSONObject` and `JSONArray`.

3.1 Storing data

An instance of `SharedPreferences` is created from which an editor is also opened. The `SharedPreferences.Editor` is an interface which allows modification of the `SharedPreferences` object. After that the data that needs to be saved is transformed to `String` to be able to be saved. This is done with the help of `JSONObject` and `JSONArray` that convert data into `String`. The save functionality is divided into multiple parts that all convert the data from the object to `String`. That `String` is then used by `SharedPreferences` and stored in a file with a key on the local device with the command `apply`.

3.2 Loading data

`SharedPreferences` is once again created and the data is loaded with the help of the key created earlier to store the data in a `String`. For the most types of data a `Gson` object is created as well as a `JSONObject` to handle the stored data and assign it to the required places. Some types of data cannot be cast so a new `Type` and `TypeToken` is created to handle the data. In some other cases the data cannot be loaded with `Gson` or `JSONObject` and must therefore be loaded from the`String` and then assigned to the right place with the help of `StringBuilder`.

3.3 Reasoning behind the functionality of the save and load functions

They way the application's data is stored and loaded is not extendable, but was still implemented because it was the only way that storing and loading data seemed to work, and yet some data was not able to be loaded from the stored state, for example is it not possible at the moment to load the habit that a group has. This is a known problem and will be updated and implemented in the future.
`SharedPreferences` is once again created and a new instance of a Gson-object is created. The json-string is then assigned to a String variable which cojoined with a Type-variable sets the userlist to the saved value. An if statement then makes sure the userlist is not null.

4 Access control and security

In this application there aren't different roles such as admin etc. Everyone that uses the application are users and they have no impact on each other except as friends.

The user can be treated as a friend from another users perspective. The other user can only see his friends name, common habits and nothing more. The friends is used so the user can make group with other users in which they can do habits together.

Referenser

[1] Microsoft docs,"The Model-View-View model Pattern". [Online]. Available: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm, accessed: 2018-10-28.

[2] Source making, "State Design Pattern". [Online]. Available: https://sourcemaking.com/design_patterns/state, accessed: 2018-10-28.

[3] Source making, "Iterator Design Pattern". [Online]. Available: https://sourcemaking.com/design_patterns/iterator, Accessed: 2018-10-26.

[4] Source making, "Adapter Design Pattern". [Online]. Available: https://sourcemaking.com/design_patterns/adapter, Accessed: 2018-10-26.

[5] Source making, "Observer Design Pattern". [Online]. Available: https://sourcemaking.com/design_patterns/observer, Accessed: 2018-10-26.

[6] Source making, "Singleton Design Pattern". [Online]. Available: https://sourcemaking.com/design_patterns/singleton, Accessed: 2018-10-21.