Report 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, businessman or if you simply are a forgetful person it can be hard too keep or create new habits. This app will keep track of habits the user wants to create and keep. It also uses notifications that are set by the user to help them remember their habits. The app also helps motivate the User with the help of streaks and achievements.

1.1 Definitions, acronyms, and abbreviations

- **Habit**: A habit is an action that the user wants repeat and keep track of. For example a habit could be washing their face every morning or water the plants once every week.
- **User**: A user is a person that has an account for the application and therefore can log in and use the application.
- **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.
- **Achievement**: When users have a number of habits or have performed their habits a number of times they unlock achievements.
- **Streak**: When a user has done a habit five days in row, the user will receive a streak. 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.

2 Requirements

2.1 User Stories

1. **Story name:** Creating habits

   Description: As a user I want to be able to to create a new habit so that I can keep improving my self.

   Confirmation:

   Functional: - be able to create a new habit
   - have the habit added to the right list
   - the habit is connected to the user and saved

   Non-functional:

2. **Story name:** Check off habits

   Description: As a user I want to be able to check of the habits I have done so I can see what I have left to do.

   Confirmation:

   Functional: - check off habits
   - make them go to done list

   Non-functional:

3. **Story name:** Reset habits

Description: As a user I want the list of daily tasks to be reset each day so I can check them off again the next day

Confirmation:

Functional: - The tasks that was marked as done the previous day goes back to undone
- You'll keep your streak when the daily reset goes through

Non-functional:

4. **Story name:** Set themes

   Description: As a user I wanna be able to set a color theme to make it more personal

   Confirmation:

   Functional: - set different themes

   Non-functional:

5. **Story name:** Undo checks

   Description: As a clumsy person i wanna be able to undo my done 'check off' if I press the wrong habit so i don't forget to do it

   Confirmation:

   Functional: - check off habits
   - undo check

   Non-functional:

6. **Story name:** Get streaks

   Description: As a user I wanna be able to get streaks when I've done a specific task multiple days in a row to make it more fun to do my habits

   Confirmation:

   Functional: - keep track of number of consecutive days habit has been done
   - receive some kind of reward when achieved a set number of days

   Non-functional:

7. **Story name:** Reminders

   Description: As a user I want to be able to set a reminder so that I get reminded about a task and don't forget about it

   Confirmation:

   Functional: - be able to set reminders
   - get reminders

   Non-functional:

8. **Story name:** Notifications

   Description: As a user I want to be able to adjust the way the app communicates (Notifications) with me to fit my personality.

   Confirmation:

   Functional: - enable or disable notifications

- receive notifications if enabled

Non-functional:

9. **Story name:** Share habits

   Description: As a user I want to be able to add friends to the app so that we can share our habits

   Confirmation:

   Functional: - be able to find friends
   - add friends
   - share habits with friends

   Non-functional:

10. **Story name:** Compete with friends

    Description: As a user I want to be able to add friends so that we can compete in who completes their habits

    Confirmation:

    Functional: - add friends
    - compare achievements with friends

    Non-functional:

11. **Story name:** Rewards

    Description: As a user I want confirmation that i am progressing so that I feel encouraged to continue

    Confirmation:

    Functional: - save user progress
    - receive achievements when goals are reached

    Non-functional:

12. **Story name:** Statistics

    Description: As a user I want to keep track of my progress so that I feel motivated

    Confirmation:

    Functional: - save information about done habits over time
    - put saved information together and show to user

    Non-functional:

13. **Story name:** Access account

    Description: As a user I want to be able to create an account so if I get a new phone I can have access to my account on the new one

    Confirmation:

    Functional: - create a user account
    - externally save accounts

    Non-functional:

14. **Story name:**

    Description: As a user I want my user account and habits to be saved when I exit the application so when I start the application again I can continue working on my good habits

    Confirmation:

    Functional: - create account for user
    - save data when exiting app

    Non-functional:

15. **Story name:** Have an overview

    Description: As a user I want to be able to overlook upcoming and past habits so I can easily take them into account when planning

    Confirmation:

    Functional: - implement a calendar to have an overview
    - keep information of habit frequencies to show in calendar

    Non-functional: -

2.2 User Interface

The user interface has received less attention due to the focus of the project being to create an object oriented program and designing the code itself. In this section the GUI design choices will be presented and explained.

The first part of the application that the user meets is the log in screen. It has a very basic layout with two fields with input prompts where the user can enter their username and password. After entering valid credentials and pressing the login button the user is taken to the applications main page. If the credentials are invalid, the user will remain on the log in page. If the user doesn't have an account there is a button called "NEW USER" that will take the user to a page where they can create a one.

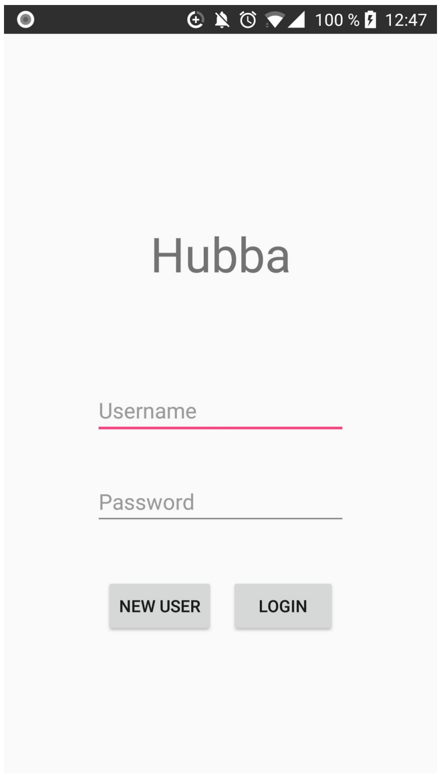

Figure 1: Screen shot of log in screen

When you want to create a new User there are some fields that are necessary to fill in so that the app receives enough information to be able to operate. The username and password are needed to be able to login and an email is for future functionality, such as recovering a username or resetting a password. At the bottom of the interface a big button that says ”create new user” can be seen (See fig 2). When pressed a new account and user is created.

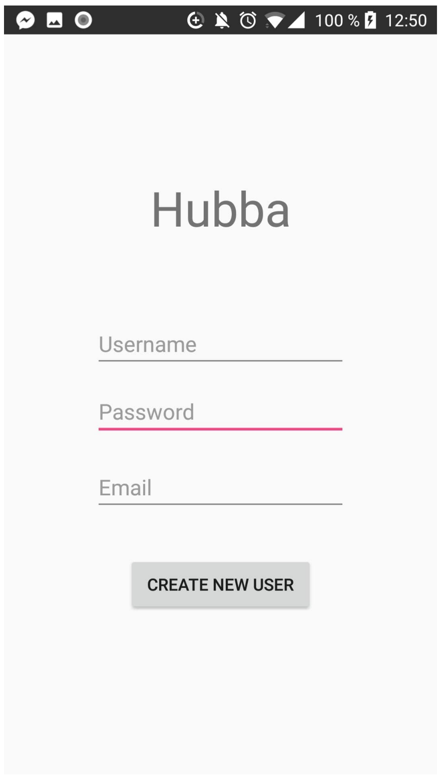

Figure 2: Screen shot of Create new user

On the main page the habits are shown under the categories Morning, Midday, Evening, Night or Done depending on what state and category the habit has been given, as you can see in fig 3. This is to easily get an overview of what you habits should be done during the day with clear separations between the different categories.

On the main page there are three navigational buttons. In the top left corner there is a menu button which takes you too the menu. In the top right corner there is a button to get to a calendar view of when the habits are to be done and down in the right corner there is a button that lets you create habits on a separate screen. Due to this being an habit application for phones and that most people are right handed the button to add habits was put down to the right for easy access.

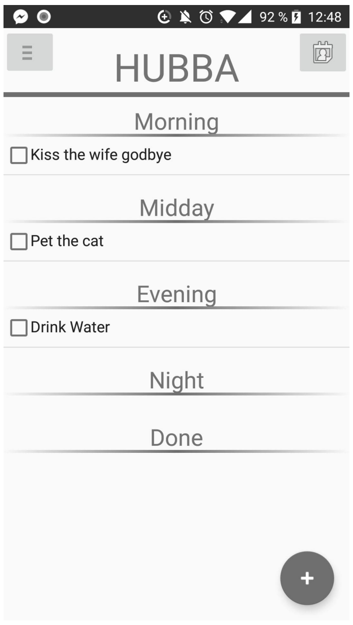

Figure 3: The front/main page with a few habits added

When the button for adding a new habit is pressed a new activity is shown, create habit, that has with only have a few buttons. These buttons impact the GUI and depending on the users choice different options will be shown as you can see in figure 4.

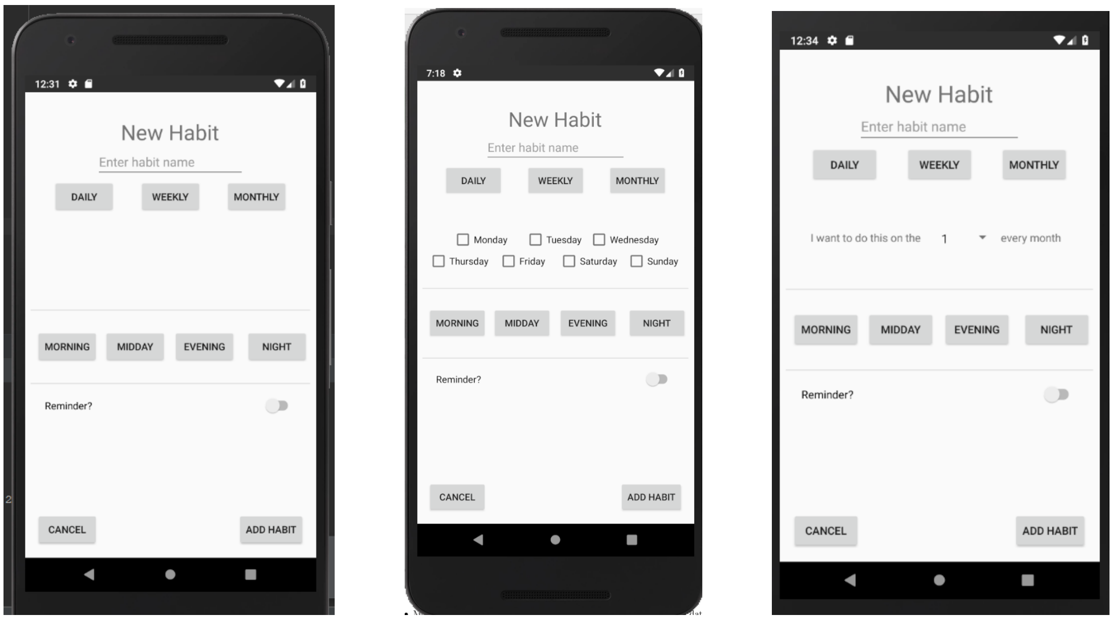

Figure 4: Different layout depending on if daily, weekly or monthly have been chosen in that order

- Daily: If the habit is to be done daily there is no special options to be chosen.
- Weakly: When weekly is chosen the user is able to chose on what days of the week it is supposed to be done.
- Monthly: When monthly was chosen the user are able to chose what date to do it on.

3 Domain Model

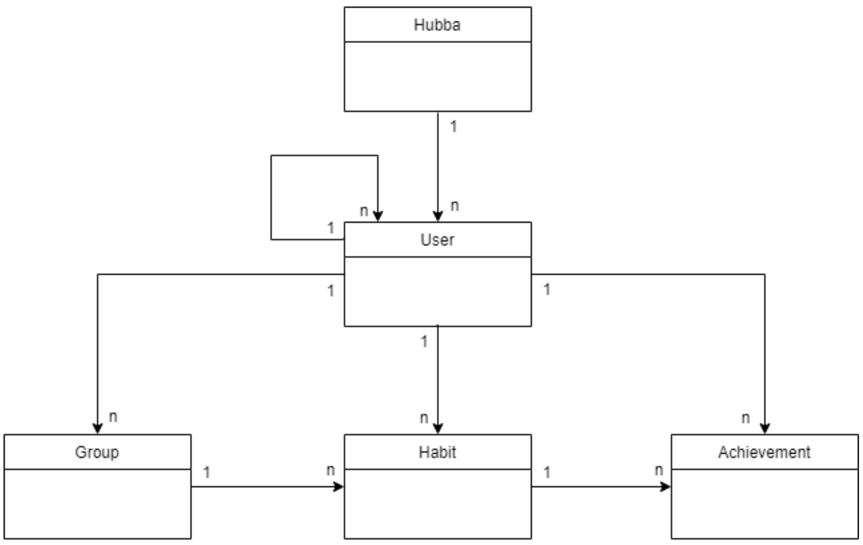

Figure 5: Hubbas domain model

3.1 Class Responsibilities

The different classes that are shown in the domain model have different responsibilities. They all belong to the model module, to which there are more classes, but these are the most relevant ones.

3.2 Hubba

Hubba works as a facade for the model module, which mainly consists of the classes shown in the domain model. All communication between the model and the other modules, view and viewmodel, go through Hubba. Another one of the main responsibilities for this class is to keep a list of all the users, and to know which user is currently using the app.

3.3 User

The user class represents the user and keeps information about the users name, email, password, habits, groups and achievements. It also keeps track of whether the user wants notifications from the app or not, and sound settings. So when the saved user logs in the settings will be the same as they were last time.

3.4 Group

If the user wants, it is possible to create group within the app, and share a habit with the other users in the group. This class is responsible for keeping the name of the group, a list of the users in the group, and a habit that the group shares. It also checks if the habit is completely done, meaning that all users have completed the habit.

3.5 Habit

This class is responsible for habits, which is what the application is based on. It holds all the detailed information about a habit:

- Title: what the habit is called.
- Streak: how many consecutive times the habit has been performed.
- State: a habit has several categories of states. Time of day, how frequently, if it's finished or not, if it's a habit for a group.

The class also makes sure that the different states can be updated.

3.6 Achievement

To help keep the user motivated to continue building good habits, achievements are added. This class has the information of an achievement, such as name, an image, and if it has been achieved or not.

4 System architecture

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

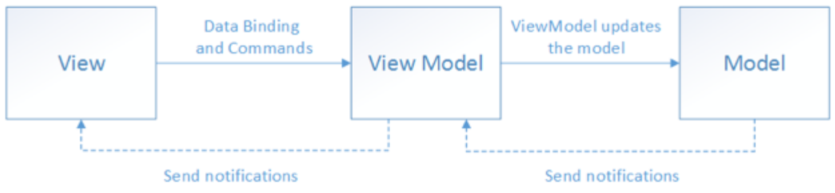

Figure 6: MVVM

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.

- The model represents the data, state, and business logic. It is completely independent of View or View Model
- The view binds to actions and variables in View Model. It can also be a package consisting of the XML-files [5].
- The view model binds the other two modules together. It provides a bridge for the view to pass events to the model, and wraps the model and the data to fit the view. The view model is not tied to the view.

4.1 Design models

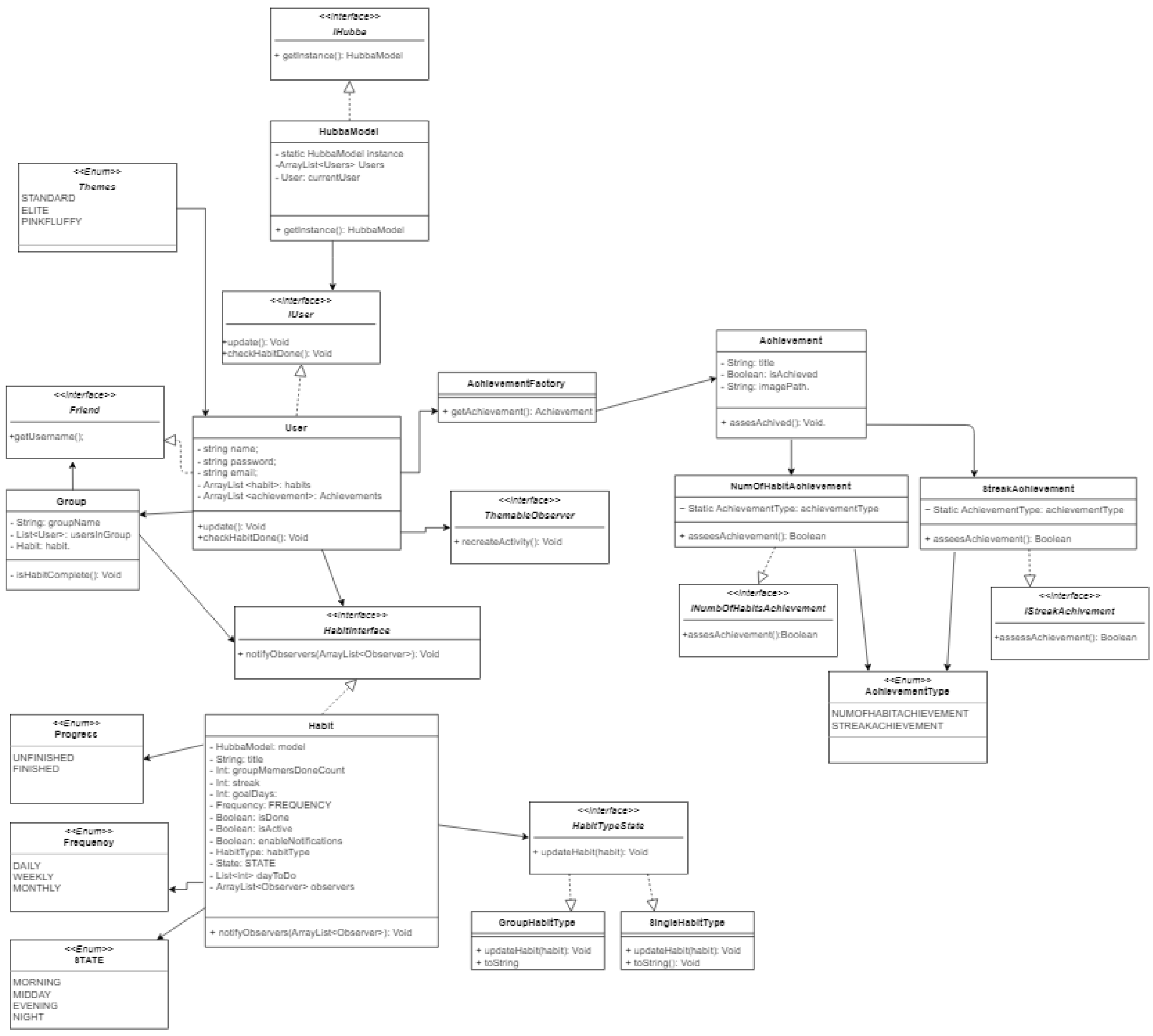

Figure 7: Design model for Hubba

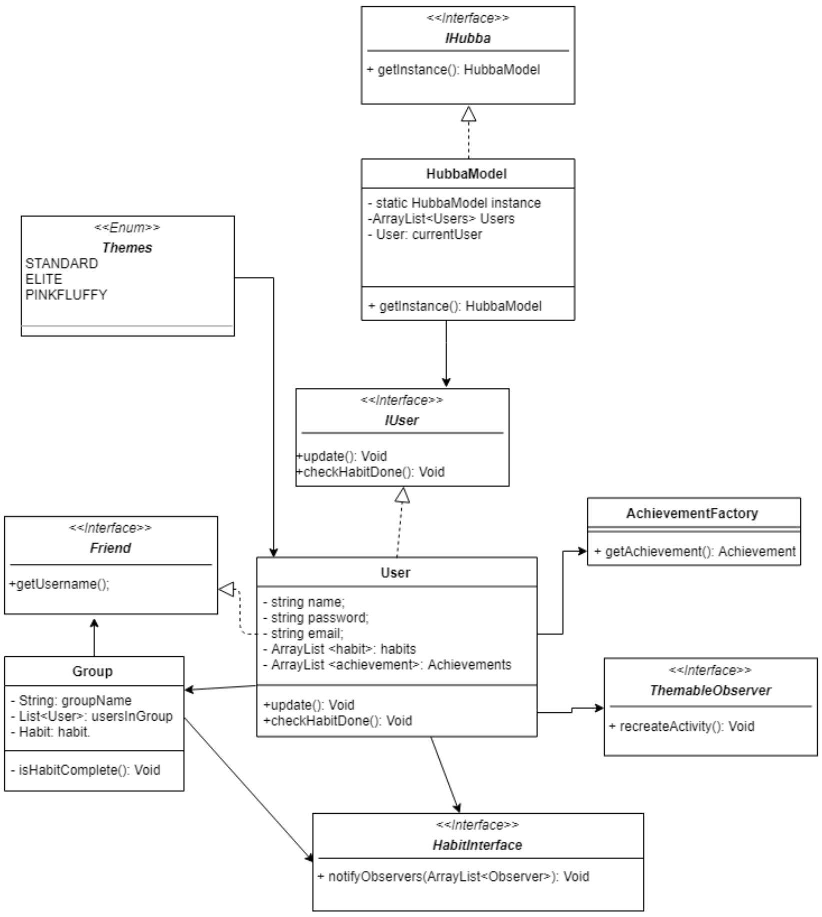

Figure 8: Closeup 1 of the design model

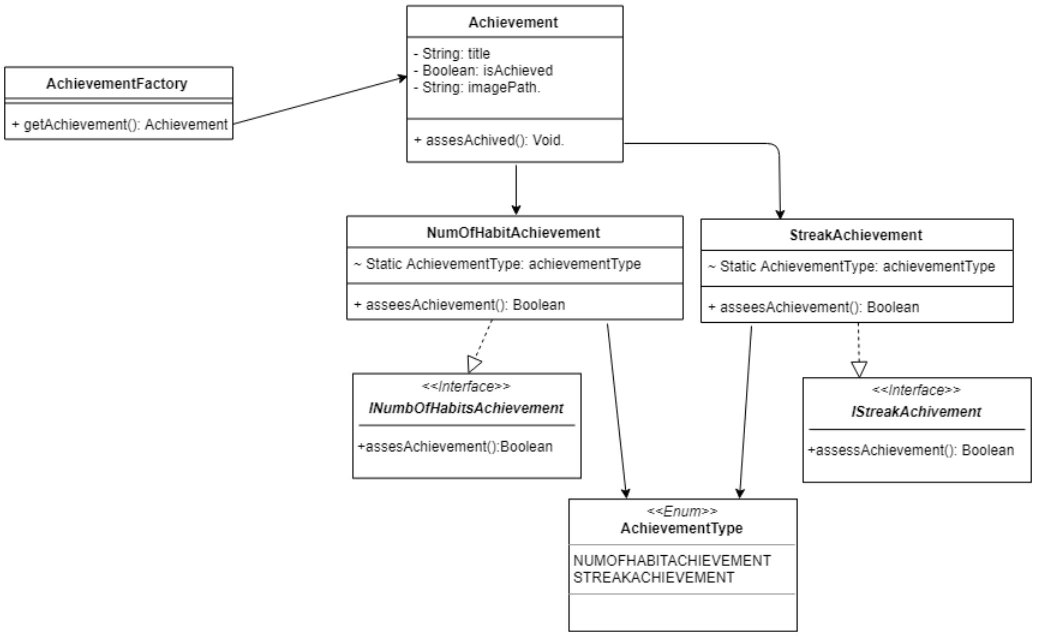

Figure 9: Closeup 2 of the design model

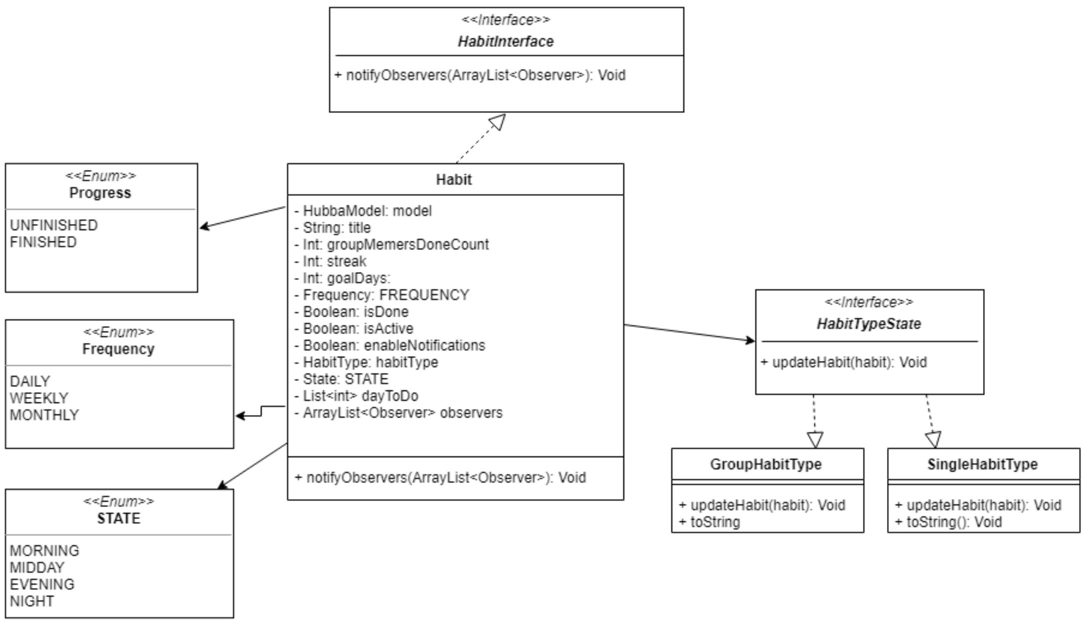

Figure 10: Closeup 3 of the design model

4.2 Subsystem decomposition

Iterator pattern provides a way to access objects in for example a list, without exposing the underlying pattern [1]. 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 [2].

Observer pattern [3].

HubbaModel is a singleton class. The singleton class is used to ensure that only one instance of a class is created [4]. 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.

4.3 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.

4.3.1 Stan

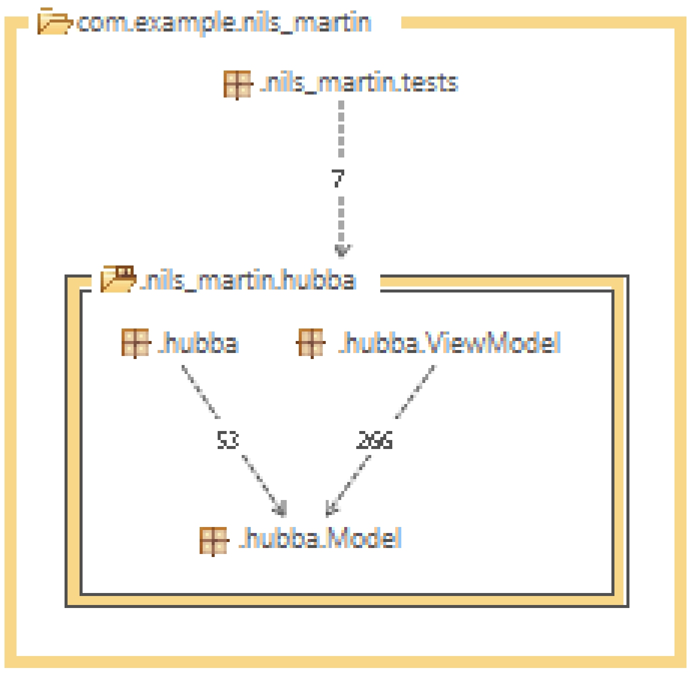

Figure 11: A Stan diagram over the package structure

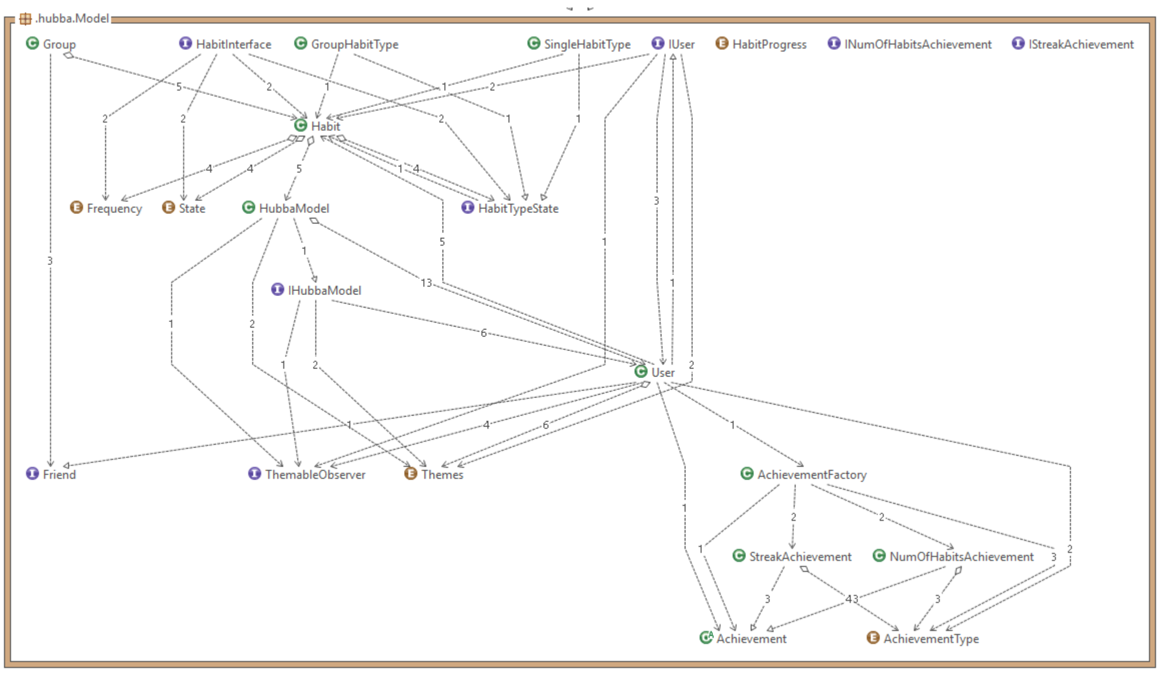

Figure 12: A Stan diagram showing the structure of the model

4.3.2 UML sequence diagram

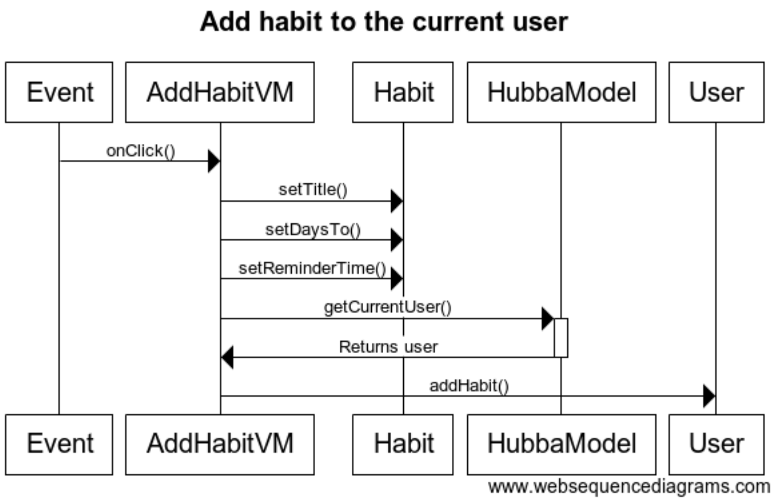

Figure 13: Diagram over the flow of create a habit

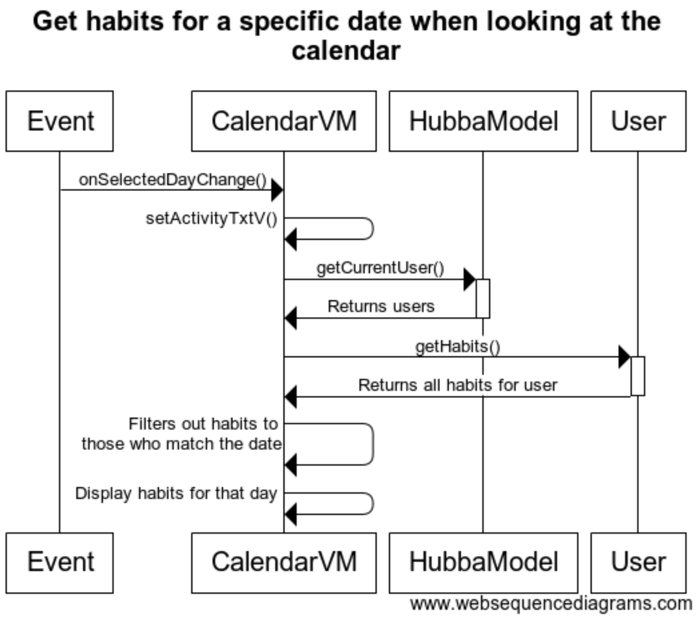
Get habits for a specific date when looking at the calendar
Event
CalendarVM
HubbaModel
User
onSelectedDayChange()
setActivityTxtV()
getCurrentUser()
Returns users
getHabits()
Returns all habits for user
Filters out habits to those who match the date
Display habits for that day
www.websequencediagrams.com

Figure 14: Diagram over the flow of display a habit in the calendar

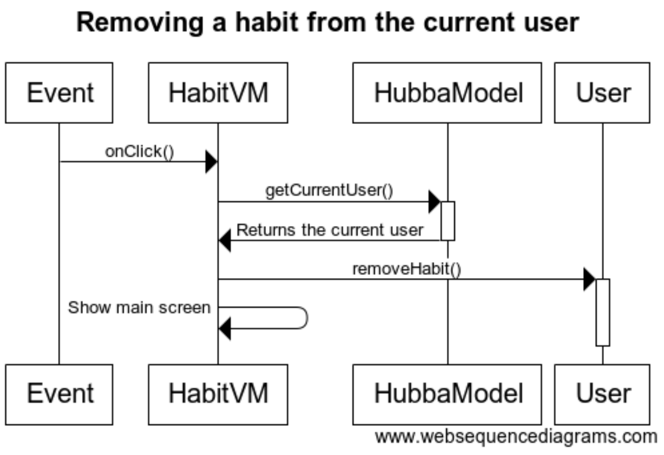

Figure 15: Diagram over the flow of remove a habit

4.3.3 Test

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

`ExampleUnitTest` - This testClass contains the tests `testCreateHabit, testCreateUser, testThemeChange` and `testUpStreak`.

`HabitUnitTest` - This testClass contains the tests `testCreateHabit, testSetDOne, testSetHabitTypeStateGroup` and `testSetHabitTypeStateSingle`.

`HabitViewTest` - This testClass contain the test `testString`.

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

4.4 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 uodate 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.

4.5 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 handels that information.

5 Persistent data management

The application stores and loads data with the help of google's Gson library, Json and android's SharedPreferences.

5.1 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. Second, a gson object is created which is used to put the userlist into a String in Json-format. The editor then matches the string with a key and sets in in the editor which stores it back to `SharedPreferences` as soon as `apply()` och `commit()` is called.

5.2 Loading data

`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.

6 Access control and security

In this application there are no different role such as admin etc. Everyone that uses the application are users and have no impact on each other expect 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.

7 Peer Review

Do the design and implementation follow design principles?
Does the project use a consistent coding style?
Yes, small classes, small methods for the most part, with exceptions for some bigger classes with methods that could be broken down into smaller parts. For example a lot of if-statements in a row in RegisterPresenter

Is the code reusable?
Some methods would be more reusable if they were decomposed into smaller methods. There are methods returning interfaces which lowers dependencies.

Is it easy to maintain?
Would be easier if some methods used more functional decomposition. But for the most part yes.

Can we easily add/remove functionality?
Uses interfaces and factories for abstraction and extendability. The code can be adjusted to use a real database instead of the internally stored files that are used now. Both this, and the fact that the user password is not encrypted is reasoned for in the SDD. (not really necessary for this project)

Are design patterns used?
Singleton pattern is used in the project as described in the project SDD. MVP is used, for more information see "Is the code easy to understand? Does it have an MVC structure?" Iterator and Adapter pattern was found when the project was read on first glance. An example of this can be seen in the class EventCollection where an iterator is used to iterate over the different collection of events.

Is the code documented?
The amount of comments varies between different parts of the code. Some classes and methods are well documented while others are still missing their documentation. This will probably be added in the future.

Some of the comments can be seen as unnecessary when it comes to commenting getters and setters since these in general have a obvious purpose. This makes the code less readable on first glance.

Are proper names used?
Proper name are mostly used in the project, which means that when reading the name of methods/classes/interfaces it is possible to understand what that part of the project is supposed to do without looking at the code.

The definition of "event", "task" and "deadline" is confusing. "Task" and "deadline" is defined as the same as "event" when in fact "task" should be something to be done and "deadline" is when to do it. This is written in both the SDD and as a documentation in the class Event.

In some of the classes, for example MonthCalendarPresenter, the local variables in some methods have very inexplicit names so that it is more difficult to understand what is happening.
Is the design modular?
The design is partially modular, it is packaged in the larger groups: (model, presenters, services and views). It might be even more modular if the classes would be grouped by the class's area of use, for instance could everything connected to 'Event' in the model-package be put in a smaller package to be even more modular.

Does the code use proper abstractions?
The code uses proper abstractions, which can be seen in the code with the implementations of a lot of different interfaces.

Is the code well tested?
The code has some tests connected to it, but since we cannot build the project it is impossible to see if they pass or not. A lot of classes and bigger methods are missing tests.

Are there any security problems, are there any performance issues?
There is some security for example ZxcvBn. They don't explain what Zxcvbn is or does. When first viewed it is very confusing and required Google to understand.

Is the code easy to understand? Does it have an MVC structure?
They have used MVP pattern which seems to be followed. It could be further explained in the SDD so that people with little to no knowledge about the pattern could understand what it is about.

Can the design or code be improved? Are there better solutions?
Yes, overall it's very good, but some things could be implemented for clarification and easier overview. For example more functional decomposition in larger methods in combination with some documentation of the external libraries used in the project.
Well commented code in Model package.

References

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

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

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

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

[5]