University of Southern Denmark

Software Engineering of Mobile Systems, (e20) - Project

MSc in Engineering (Software Engineering) - 1st semester

Deadline for submission: December 17th 2020

Green Parking Spotter (GPS)

Student:	**Student:**	**Student:**
Jan Kramer	Sigurd Eugen Espersen	Troels Zink Kristensen
jakra20@student.sdu.dk	siesp17@student.sdu.dk	tkris17@student.sdu.dk
13/07-1995	21/11-1996	30/04-1997
Exam no.: 489492	Exam no.: 455883	Exam no.: 455427

Lector:
Mikkel Baun Kjærgaard
mbkj@mmmi.sdu.dk

Abstract

In this prototype of an application, the aim is to incentivize the use of green cars. This is an object for making the transport sector more sustainable than it is today. For this reason, the parking lots at SDU should be prioritised for users with green cars.

The application is written in Java within Android Studio. In order to control the system behaviour, quality attributes and tactics are applied.

A major aspect of the course Software Engineering of Mobile System, for which this prototype is developed, is the use of sensors. Therefore, the location sensor of the mobile phone is utilised in this project. The MVVM (Model-View-ViewModel) architecture is used for the implementation of the application. Within this architecture one activity is used together with several fragments in order to guarantee separation of concerns in the application. A flowchart is created to show the flow of these fragments.

According to the MVVM-architecture, the different parts of the implementation are separated. Therefore, the functionality of the application are structured into Model, View and ViewModel. Furthermore, the Acquaintance package is used to expand the MVVM-architecture to provide better usability of the architecture.

Finally, a discussion and evaluation are conducted, where quality attributes, tactics and the choice of sensors are critically questioned. Furthermore, the report is briefly concluded and an outlook for future works is provided.

Preface

This project is composed by Jan Kramer, Sigurd Eugen Espersen and Troels Zink Kristensen.

The project is associated with the course: Software Engineering of Mobile Systems in MSc in Engineering (Software Engineering) on the 1st semester in autumn 2020 at the University of Southern Denmark in Odense. The course covers 10 ECTS-points.

The project started September 22th and was concluded on December 17th with the submission of this report.

Reading guide:

Literature references are shown by $^{[x]}$ and refers to the reference list in section 9. Code specific expressions are formatted using the `menlo font`. An overview of the electronic appendix can be found in section 10 (Appendix).
The PDF is interactive, which means that it is possible to click on references to chapters, figures and code listings to jump directly to them.

Table of Contents

1 Introduction **1**

1.1 Motivation and Context . . . 1

1.2 Problem and Objective . . . 1

1.3 Report Outline . . . 1

2 Problem Description **2**

2.1 Background of the Problem . . . 2

2.2 Proposed Solution . . . 2

3 Analysis **3**

3.1 Android Development . . . 3

3.2 Quality Attributes . . . 3

3.3 Tactics . . . 5

4 Design **7**

4.1 Sensors and Services . . . 7

4.2 MVVM-Architecture . . . 7

4.3 Activities and Fragments . . . 9

4.4 Flowchart and Use Cases . . . 10

5 Implementation **13**

5.1 MainActivity . . . 13

5.2 Create Account and Login . . . 13

5.3 Map . . . 18

5.4 Parking . . . 22

5.5 Acquaintance . . . 24

6 Discussion and Evaluation **26**

7 Conclusion **28**

8 Future Works **29**

9 References **A**

10 Appendix **B**

10.1 Overview of the Electronic Appendix . . . B

10.2 Images of Fragments . . . B

1 Introduction

This project is part of the course: Software Engineering of Mobile Systems, in which it is required to develop a prototype of an application to mobile phones. The solution for the application includes techniques used in the course, such as design, architecture and implementation of mobile sensing systems.

1.1 Motivation and Context

Several possible projects cases were made available to choose from, and those were provided by the companies: Cybot, House of Code and University of Southern Denmark (SDU). The project case chosen for this project is the *Green Mobility* provided by SDU. This exact case was thought to be relevant and interesting, especially as a result of the other cases being outdated ones, as they were presented last year as well. *Green Mobility* is a new project case for this year.

1.2 Problem and Objective

The project case: *Green Mobility* is about reducing the number of conventional cars (running on fossil fuels) and thus increasing green cars (electric and hybrid cars) on the parking lots at SDU. The transport sector has been changing rapidly up until now, and will continue to do so in the future. This sector has a large impact on the environment; therefore, it is desired to exchange conventional cars with green cars. SDU is planning to replace their own cars with green ones, and creating charging stations for those cars. The university want to encourage their employees and students to use green cars by prioritising parking lots for those cars over conventional cars.

The application will contain a login system, where users can create an account with their car type connected to their e-mail. When an employee or a student wants to park at SDU, they will use the application to indicate that the car is parked at that location. On a map it is then possible to see the occupied and available parking spaces on the parking lot.

1.3 Report Outline

The disposition of the report is given here in chronological order:

- Problem Description
  Detailed description of the problem
- Analysis
  Considerations are obtained
- Design
  Choice of architecture and sensors
- Implementation
  Results of the design and code
- Discussion and Evaluation
  Reasons behind choices
- Conclusion, Future Works
  Results of project + extra features

2 Problem Description

This section is an extension to the introduction. The problem described is explained in detail here in conjunction with a proposed solution for this project.

2.1 Background of the Problem

The *Green Mobility* project case is an important initiative composed by SDU. It is desired to contribute to the worldwide task of reducing the emission of greenhouse gasses, especially carbon dioxide (CO_2). The transport sector has a substantial impact on the environment. The transport sector constituted about 57 percent of the total emission of CO_2 from Danish economical activities in 2019.[1] Furthermore, in 2018 about 67 percent of the total emission of CO_2 produced from Danish companies was due to transport.[2] From these numbers, it can be concluded that the transport sector is by far the largest contribution to the emission of CO_2. However, a high proportion of the CO_2 from the transport sector is due to large ships, aircraft and similar vehicles. Nevertheless, passenger cars are a factor as well.

The direction for a sustainable world and a green transport sector has to start somewhere, which could be started at SDU by the creation of this application, which intention is to increase the number of green cars at the parking lots at SDU.

2.2 Proposed Solution

The proposed solution consist of an application for Android users. A login system is created, where a person can create an account if the person is employed or enrolled as a student at SDU. The user will create an account with their corresponding working / student e-mail from SDU, along with the car type and car brand of their personal car. The car type does not have to be an electric or hybrid car (green cars); it can also be a conventional car. The users, who have a green car will receive prioritization over users with a conventional car in the form of available parking lots and benefits / perks at the university. When the user has logged into the application, then he / she will be represented with a button, which should be pressed in order to park their car in the cloud at the location they have parked physically. The user has to press the button once more, when leaving the parking lot. A map will be provided to locate available and occupied parking lots; there is a larger pool of parking lots for users with green cars. It will also be possible to see, where the user's own car is parked on the map. There should be little to no waiting time, when the user presses the button to park their car, and it should be clear, when the car is parked.
The proposed solution will be analyzed, designed and implemented in the next sections.

3 Analysis

The proposed solution given in the last section has to be analyzed before designing and implementing the application. The analysis is done in terms of coding environment, quality attributes and tactics.

3.1 Android Development

As the aim of this project is to design and implement an application for the mobile phone, Android Studio the free and official IDE for Android application is used. For this prototype, only the implementation on the Android operating system is of relevance. Android Studio has two programming languages:

1. Kotlin
2. Java

Java is the programming language, which was chosen for this project. That happened, as the software developers of this project are more familiar with Java. Moreover, the time investment to acquire the necessary Kotlin skills would have been not in relation to the actual outcome.

3.2 Quality Attributes

The quality attributes (QAs) of a system are equivalent to non-functional requirements. Eight common QAs are listed below.[3] Although, there are many more attributes available.

- Availability
- Modifiability
- Performance
- Security
- Testability
- Usability
- Energy efficiency
- Resource adaptability

When describing a quality attribute to a specific problem in an application, six subjects have to be considered, which are listed below.

- Source of stimulus
  Entity that generated the stimulus
- Stimulus
  Condition to be considered when arriving at the system
- Environment
  Where/when does the stimulus occur
- Artifact
  This one is stimulated
- Response
  Activity executed when the stimulus arrives
- Response measure
  Measurable, requirement can be tested

The important QAs for this application are *performance*, *availability* and *usability*. Performance is an attribute which consider the time from a user event to a system event. Availability is considered as the application has to be operational when the user needs to use it; for parking at and leaving a parking space. It is desired for this time interval to be low. Lastly, usability is required, as the application has to be as reliable and user-friendly as possible. Three quality attribute scenarios are constructed based on availability, performance and usability, respectively. See tables 1, 2 and 3.

Table 1: *A quality attribute scenario for performance.*

PERFORMANCE	
Portion of Scenario	Possible Values
Source	Users
Stimulus	Initiate parking requests
Artifact	System
Environment	Many users parking in the morning
Response	Incoming parking requests are put into a queue
Response Measure	No waiting time for users, as the system will execute the queue

Table 2: *A quality attribute scenario for availability.*

AVAILABILITY	
Portion of Scenario	Possible Values
Source	Users
Stimulus	Accessing the map overview
Artifact	System
Environment	Normal operation
Response	Map overview is accessed and the used parking space is marked on the parking lot
Response Measure	Within two seconds

Table 3: *A quality attribute scenario for usability.*

USABILITY	
Portion of Scenario	Possible Values
Source	Location provider
Stimulus	Parking request is registered at an incorrect parking space
Artifact	System
Environment	Parking spaces are close to each other at the parking lot
Response	GPS coordinates has to saved at 0.000001 precision to distinguish between the parking spaces
Response Measure	Reduces incorrect measurements down to zero percent

The quality attribute scenario in table 1 is important, as the users should not wait for the car to be parked within the application. The requests are put into a queue, which does not necessarily park instantly. However, the user does not have to wait, but instead, can walk away from the car and go to class or start working at SDU. The scenario in table 2 ensures that one of the key features can be accessed and available within a concise time frame. In this way, the user will not quit the application as a result of a long loading time. The scenario in 3 is most important one of three three scenarios. The location provider should have a high precision, in which all of the parking spaces on the parking lot will be distinguishable.

3.3 Tactics

Architectural tactics are designs to handle occurring events or stimuli and to provide a adequate reaction. This stimuli are described by the QAs. These stimuli and their reactions differ from tactic to tactic. Subsequently, several tactics will be listed and three will be explained in detail. Even though there are more tactics than the ones listed below, this should give an impression of the variety and function of tactics.

- Availability
- Modifiability
- Performance
- Security
- Testability
- Usability
- Energy efficiency
- Resource adaptability
- Privacy

The reason why the list of tactics is similar to the one of the QAs is that the tactics are the designed response to the stimuli (QAs), as aforementioned.

Energy efficiency, resource adaptability and privacy will be explained further.

Energy efficiency:
Energy efficiency is the tactic to use the energy consumption of the system as efficient as possible. It handles the energy consumption of an entity or the entire system and response in a designed manner to keep the energy consumption low. An example for that might be the request of location data. For a long path it is more efficient to request the location data after a certain time interval and calculate the interim movement with trajectories.

Resource adaptability:
Resource adaptability is the tactic to adapt to a changing environment by adapting the resources. If a change in the environment occurs, the resource adaptability tactic describes how the resources should adapt so that the system keeps working as expected. This is important for the guaranteed continuous usage of the system. An example here could be to increase the amount of available resources. This helps to avoid using dysfunctional resources.[4]

Privacy:
Privacy is the tactic to ensure the privacy of the user. It describes how to secure the data. Some of the aspects used by the privacy tactic are described in the following. *Hide* prevents the data to be exposed to access, to be visible or to be understandable by unauthorised access. *Separate* prevents the correlation of personal data. *Minimise* limits the access to the personal data. *Abstract* keeps the personal data general and does not go into detail. These are some of the aspects that could be used by the privacy tactic.[5]

In this project the tactic of resource adaptability is applied. Furthermore, the sensor in this project is the location sensor. Therefore, the design of the resource adaptability in this case is to switch between the resources to find the resource delivering the best location data. There are two ways in this application to obtain location data. One way is the locating via the internet connection and the other way is the GPS-based location. Thus, the stimuli is the change of quality of the location data of the resources and the response is then to switch to the resource with the better location data quality.

4 Design

In the following section the design choices made are described, covering the sensors and services used for the application. Following up with an overview of the MVVM-architecture, and some key android development features (activities and fragments). Finally, some flowcharts and use cases will be formulated.

4.1 Sensors and Services

Using the sensors of the mobile phone is a major aspect of the course Software Engineering of Mobile Systems. In this project the location sensor is to be used. To access the location of the phone the class `FusedLocationProviderClient` can be used. It inherits from the `Google Api` class.
When using the `FusedLocationProviderClient`, one way to obtain the location is the `getLastLocation()`, which "returns the best recent location currently available". [6] Therefore, the return type is `Location`. In rare cases the return type could also be `null`, if the location is not available (for example if the location on the phone is turned off). Another way of obtaining the location is for example the `getCurrentLocation(int priority, CancellationToken token)`, which takes an integer as priority and a `CancellationsToken`. It returns a fresh location, if it could fetch the location within a time span, otherwise it returns `null`.[7]
In order to use the `FusedLocationProviderClient` it can be necessary to fulfil some preconditions. Firstly, the dependency `play-services-location` has to be included. This is done in the gradle file. Secondly, the permission for accessing the location could be needed, for example the internet and the GPS of the phone.

Moreover, the class `Google Map` can be used to display markers as an example and locations on a map interface. `Google Maps` has different methods, such as `addMarker()`. Furthermore, `Google Maps` can invoke the `animateCamera()` method, which manipulates the camera and the zooming in and out.
However to use the `Google Map` class an API key has to be created, in order to display a map. This is free for a low frequent use of the map, while for a high frequent use the API key becomes chargeable. This is no problem for a prototype; however, if this application should be made available for a broader audience, this has to be taken into consideration.

4.2 MVVM-Architecture

When developing any kind of software a strong foundation is required. The building blocks for any successful enterprise in regards to software are found in its architecture. Once the formal requirements are found, the architecture needs to be set to ensure a mutual agreement between developers. In addition, the type of software system needs to be taken into account to make the best possible decision of architecture. Since the Green Parking Spotter app is an android app, the MVVM (Model-View-ViewModel) architecture is a solid choice as it handles all major concerns with android development. MVVM separates the system into distinctive segments that each have their own respective role and functionality, thereby fulfilling the design principle, separation of concerns (SoC).[8] However, all communication is one-directional going from above and down, never the other way around. Therefore, the View can directly communicate with the ViewModel that can directly communicate with the Model. The ViewModel can communicate with the View using the data binding technique, which will be described in further detail in the ViewModel section. The Model can communicate with the ViewModel using the callback pattern, which is described in further detail in the Model section.

4.2.1 Model

The Model segment contains and handles all business logic and communication with the server. The Model serves as an entry/exit point for the system and is where implementations of common interfaces are created. The Model can communicate with the ViewModel using the strategy pattern callback. This pattern makes it possible to receive data at a later point than execution, by passing a callback object along with the other data. Once the Model has e.g. received a response from a server, the callback object can be used to communicate back to the ViewModel. Data received from the server will, therefore, be packaged into an object and/or sent directly to the ViewModel using the callback pattern. Data coming from above in the ViewModel is received and packaged into requests that are being sent to the server.

4.2.2 View

The View segment consists and handles the layout and overall appearance of the screen including all user interactions. The View registers clicks, events, and other interactions through various listeners. When a gesture is made the View can alter the appearance of objects on the screen or call the ViewModel for assistance to handle some data. Furthermore, the View segment is the only segment that can change actual views in android i.e. the ViewModel or Model cannot alter the appearance of the screen.

4.2.3 ViewModel

The ViewModel is an abstraction of the View, providing public methods for the View to use. This segment serves as a middleman between the View and Model, receiving data from both and transferring it along depending on direction. If data were to be sent to the

Model, simply having a reference to the Model and calling methods to pass on the data will suffice. However, to send data to the View, a data binding must be used. MVVM uses data binding between the View and ViewModel; this works by posting data into a LiveData which is a data holder class that can be observed. From the View, a reference to the ViewModel is created at the start of the life cycle; from here LiveData objects can be observed for changes. Once the ViewModel posts data into a LiveData, the View will be notified of the change and can use the data as it pleases.[9] [10]

4.2.4 Acquaintance

As an extension of the MVVM-architecture, an acquaintance package can be used to provide a reference to common interfaces and data that all segments use. E.g. handling of a specific object used by all segments without the acquaintance package, would require each segment to have their own definition of said object. In addition, the acquaintance package makes it easier to distribute objects between segments and can ease development by having a shared data class. A shared data class could, for instance, contain login and connection information and common functionality, such as replacing activities and fragments.[11]

4.3 Activities and Fragments

Within the design of the application, activities and fragments are used in Android Studio, which are considered below.

4.3.1 Activities

An activity should be considered as an equivalent component to the `main()`-function from ordinary programming in C, C++ and Java. The main activity is the one that is initialized when opening the application. Unlike ordinary programming, where there is only one `main()`-function, multiple activities are a possibility in Android; this is usually desired when a new action is required inside the application. An example is an e-mail application, where a new activity can be invoked when the user wants to write a new e-mail.[12] The main activity can launch distinct activites from other applications as well, such as opening Google Chrome from a link in the Facebook application. For this particular application in this project, it is not desired to make use of multiple activities. However, it is desired to used one activity (the main activity) and several fragments.

4.3.2 Fragments

Fragments are parts of the user interface (UI). They are used to get modular and reusable components into the UI. Fundamentally, a fragment has to be hosted from an activity or another fragment. Furthermore, they have their own lifecycle and own layout within an activity or another fragment. For a layout containing different screens, fragments are used to display the aforementioned screens. Moreover, the fragment is tied to the activity it is hosted in and cannot be used outside this activity.

4.3.3 Lifecycles

The lifecycles of fragments depends on the lifecycle of the hosting activity or fragment. This means that if the lifecycle of the hosting activity or fragment ends, the fragment is also destroyed. On the other hand, the lifecycle of the fragment starts when the fragment is created by an activity or another fragment Additionally, if the lifecycle of the activity is destroyed or had not yet begun, no fragment can be hosted from it. Therefore, the lifecycle of a fragment can be started or stopped without interfering with the lifecycle of the hosting activity or fragment.
The lifecycle of an fragment runs through different states:[13]

1. Initialised
2. Created
3. Started
4. Resumed
5. Destroyed

... including the callback methods `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, `onDestroy()`.
Fragments are added to the `FragmentManager`. This manages the transaction from one state to another and returns the current state. Furthermore, it also attaches and detaches the fragments to the hosting activity.

4.4 Flowchart and Use Cases

Figure 2 is a flowchart of the application, which depicts the flow of the entire core functionality. The flow is divided into three phases, numbered by 1, 2 and 3, and three colors.

Red Phase [1] - This phase consists of setting up the application before the user is able to use it for its main purpose. Firstly, the Login-fragment is initialized, and secondly, the user can either create a new account or login with a already existing account. When a new account is created, the user is transferred back to the Login-fragment. Afterwards, the user attempts to login, which can either be a success or a failure.

Green Phase [2] - The login has been successful, which will initialize the Map-fragment and instantly initialize the Parking-fragment afterwards. This will create the map, which is needed before the user is able to park. Furthermore, the user can park by pressing the Park button which will occupy a parking space in the parking lot on the map. It is possible to press the Leave button afterwards, which should be done before leaving the parking lot physically. The Map fragment is updated every time the user parks or leaves.

Yellow Phase [3] - This phase contains a supplement to the application, which grants a simplified and user-friendly graphical user interface. A hamburger-menu is implemented with six different items, which are: Parking, Map, Perks, Account, About and Logout. Each of these items will invoke a fragment of the main activity. Parking and Map open up the parking and leave functionality, and the map itself, respectively. Perks, Account and About consists of discounts for user's with green cars, information about their account, and information about the creators, respectively. The Logout item logs the user out, who will be returned to the Login-fragment.

The equivalent use cases are shown in figure 1 to simplify what the user is able to do within the application.

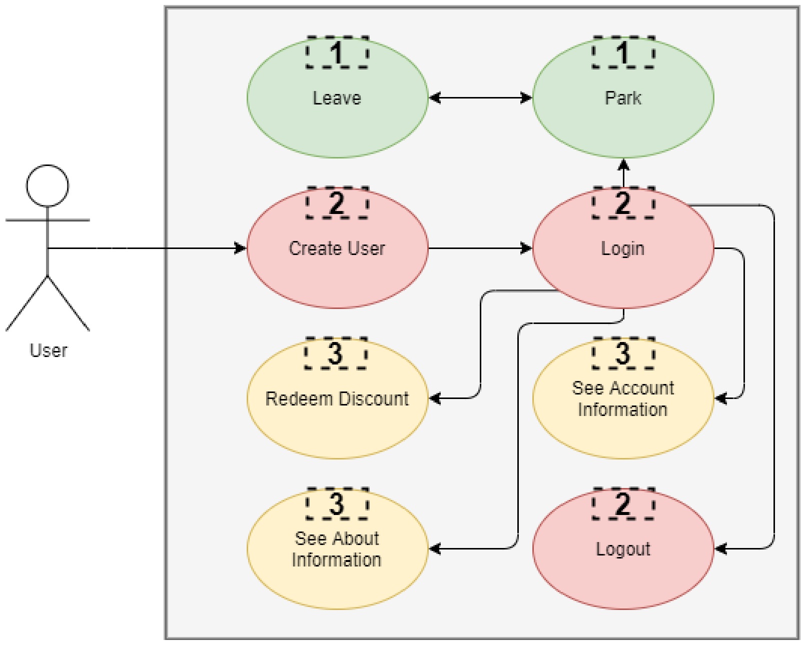

Figure 1: *Use cases from the application.*

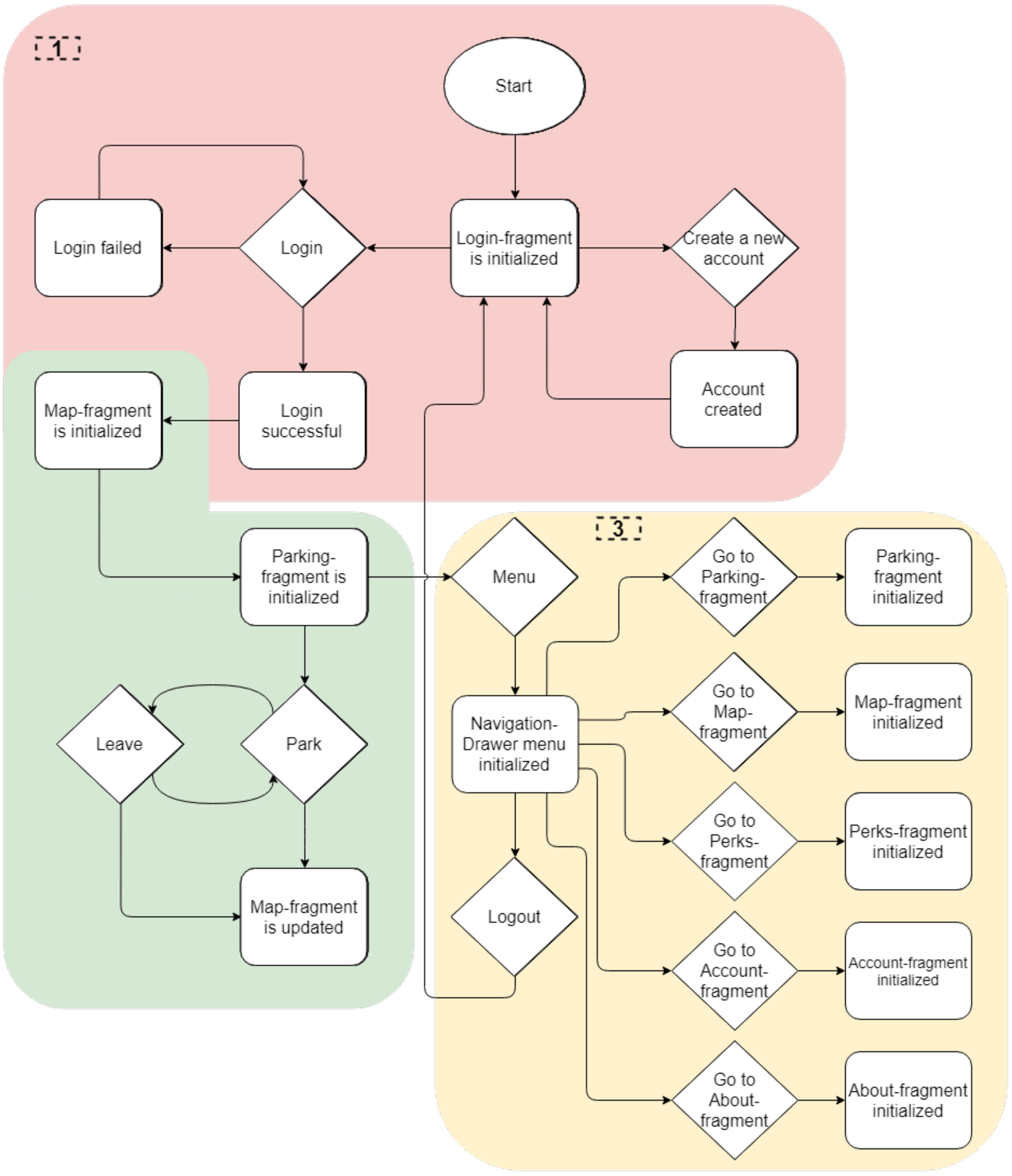

Figure 2: *Flowchart of the application.*

5 Implementation

In the following section the implementation choices made are described, starting with an overview over the MainActivity following up with important fragments within the application, and lastly describing some architectural decisions including the implementation of the database.

5.1 MainActivity

The application uses a single activity that operates as a main class, that during launch will go through its lifecycle. During the `onCreate` lifecycle, Green Parking Spotter will check for permissions to use the `ACCESS_FINE_LOCATION`, which is required to retrieve location data. Having this check in the main activity will ensure that the user has given permission to use location data, before being able to use any features that require this permission.

Furthermore, the main activity also initializes the menu that is used to navigate the entire application. The menu uses the interface `NavigationView.OnNavigationItemSelectedListener` that ensures that the `onNavigationItemSelected` is called whenever a user selects a menu item. The handling of `onNavigationItemSelected` is also done within the main activity, as this code is something that needs to be run no matter what fragment the user is on.

Finally the main activity as a last step in the `onCreate` lifecycle, uses `FragmentTransaction` to swap from the `activity_main.xml` that contains the overall layout for the application to the `login_view.xml`, so that the user is prompted with the login screen and can proceed to use the application. The menu button persist through all fragments, as it is contained in another layout within the `activity_main.xml` than the one being swapped for fragments.

5.2 Create Account and Login

Two fragments of the main activity are created, called CreateAccount and Login, respectively. Before a user can login, an account has to be created with a student- or employee e-mail. The View, ViewModel and Model are explained below. The design of both fragments are shown in the *Appendix* in figures 3 and 4.

5.2.1 View

The CreateAccount fragment contains a View. Inside this View, four different options have to be filled out by the user:

- Student- or employee e-mail
- Password
- Car type
- Car brand

```
VM = new ViewModelProvider(requireActivity()).get(CreateAccountViewModel.class);
...
VM.createAccountStatus().observe(requireActivity(), new Observer<Boolean>() {
    @Override
    public void onChanged(Boolean status) {
        if (status) {
            Toast.makeText(getContext(), "Account created",
            ↪ Toast.LENGTH_SHORT).show();
            LoginView loginView = new LoginView();
            FragmentTransaction fragmentTransaction = ((FragmentActivity)
            ↪ getContext()).getSupportFragmentManager().beginTransaction();
            fragmentTransaction.replace(R.id.container_fragments, loginView);
            fragmentTransaction.addToBackStack("LoginView");
            fragmentTransaction.commit();
            ...
        } else {
            Toast.makeText(getContext(), "Something went wrong. Try again",
            ↪ Toast.LENGTH_SHORT).show();
        }}
```

Listing 1: *Creating a new account.*

The method in listing 1 will be executed when a new value is posted from the ViewModel. The View receives a `MutableLiveData`, which can be observed upon.
If the observed `MutableLiveData` has changed, the `onChanged()`-function will be executed. If the account was successfully created, status will be equal to true, which will in turn make a fragment transaction and replace the CreateAccount fragment with the Login fragment. A message will be sent to the user indicating that the account was created. Afterwards the user is able to login.

If the account was successfully created, the Login fragment will replace the CreateAccount fragment. The method `loginStatus()` is called on the Login ViewModel, where a `Pair` of `Boolean` (successful login or not) and `String` (message to the user) is observed. Listing 2 shows this method.

```
loginVM = new ViewModelProvider(requireActivity()).get(LoginViewModel.class);
loginVM.loginStatus().observe(requireActivity(), new Observer<Pair<Boolean,
↪ String>>() {
    @Override
    public void onChanged(Pair<Boolean, String> status) {
        if (status.first) {
            ...
            ParkingView parkingView = new ParkingView();
            fragmentTransaction = ((FragmentActivity)
            ↪  getContext()).getSupportFragmentManager().beginTransaction();
            fragmentTransaction.replace(R.id.container_fragments, parkingView);
            fragmentTransaction.commit();
            ...
        } else {
            Toast.makeText(getContext(), status.second,
            ↪  Toast.LENGTH_SHORT).show();
        }}});
```

Listing 2: *Attempting to login.*

If the `Pair` has changed, the `onChanged()`-method will be executed, which checks whether the `Boolean`-value is true or not. The login attempt was successful if `status.first` returned true, which in turn will replace the Login fragment with the Parking fragment. However, a part of the code is removed in this code snippet, as the Map View fragment is actually initialized before the Parking View, as the Map has to be initialized first to create the map.

5.2.2 ViewModel

The ViewModel ensures the communication between the View and the Model. Listing 3 shows the method that the View executes on the ViewModel, when the Create Account button is pressed within the application.

```
public void createAccount(String mail, String password, String carType, String
↪  carBrand, Context c) {
        createAccountService.createAccount(new Callback() {
            @Override
            public void onResponse(Object o) {
                createAccountStatus.postValue((boolean) o);
            }
        },mail,password, carType, carBrand, c);
    }
```

Listing 3: *Communcation between the View and the Model in CreateAccount.*

This method calls the `createAccount()`-method on an instance of the `createAccountService`, which is within the Model. The `createAccount()`-method in the ViewModel simply forwards the information given in the View to the Model. Furthermore, the response from the Model regarding a successful created account is posted to the `createAccountStatus`, which the View observes.

The `login()`-method is called in the Login View, when a user attempts to login.
The information given by the user, username and password, is sent to the Model with a callback. Once the model has retrieved data from the database, a response is created and returned through the callbacks' `onResponse()`-method and posted into the `loginStatus`, which is a `MutableLiveData`.

5.2.3 Model

The information sent from the View to the ViewModel is received by the Model within the `creatAccount()`-method. Listing 4 shows this method. An `AsyncTask` is created to use this method on a separate thread.

```
public void createAccount(final Callback callback, final String mail, final
↪ String password, final String carType, final String carBrand, final Context
↪ c) {
    new AsyncTask<Void, Void, Void>() {
        @Override
        protected Void doInBackground(Void... params) {
            User user = new User();     user.student_mail = mail;
            user.password = password;   user.car_type = carType;
            user.car_brand = carBrand;  user.coins = 100;
            UserDao userDao = Common.getInstance().getDatabase(c).userDao();
            User existingUser = userDao.getUserByMail(mail);
            if (existingUser == null || !existingUser.student_mail.equals(mail))
            ↪ {
                userDao.insert(user);
                callback.onResponse(true);
            } else { callback.onResponse(false); }
        }
    }.execute();
}
```

Listing 4: *Creating a new account and inserting into the database.*

A new user is created, which is a data object. The information given by the ViewModel is set to the corresponding member variables within the user. Furthermore, `user.coins` is initialized to 100. Afterwards, it is checked whether the user already exists compared to the e-mail. If the user does not exist or the mail is not applied to another user, the user will be inserted into the database. A boolean value equal to true will be returned to the ViewModel.

As the user's new account is now created, the user is able to login. The Model has received information given by the user through the View and ViewModel. Listing 5 shows the method called in the Model.

```
public void login(final Callback callback, final String mail, final String
↪  password, final Context c) {
    new AsyncTask<Void, Void, Void>() {
        @Override
        protected Void doInBackground(Void... params) {
            ...
            UserDao userDao = Common.getInstance().getDatabase(c).userDao();
            User user = userDao.getUserByMail(mail);
            if (user == null || !user.student_mail.equals(mail)) {
                Pair<Boolean, String> result = new Pair<>(false, "User does not
                ↪  exist");
                callback.onResponse(result);
            }
            if (!user.password.equals(password)) {
                Pair<Boolean, String> result = new Pair<>(false, "Wrong password.
                ↪  Try again");
                callback.onResponse(result);
            }
            SessionData.getInstance().setCurrentUser(user);
            Pair<Boolean, String> result = new Pair<>(true, "");
            callback.onResponse(result);
        }
    }.execute();
}
```

Listing 5: *Creating a new account and inserting into the database.*

The user is acquired from the database. If the user does not exist or the e-mail is incorrect, the result of the `Pair` will be false and a message, which indicates that the user does not exist. The `result` is given to the callback function, which will be retrieved in the ViewModel, If the user does exist, but the password is wrong, the `result` will be false as well, but with a message describing that it was a wrong password. If both e-mail and password were correct, `result` is returned with true and an empty message.

5.3 Map

The Map is supposed to show the different parking spots, their availability and the current location of the user.

5.3.1 View

To start with, the fragment MapView has two important methods for the creating process, which are specific for a fragment. In the `onCreateView()`-method several ArrayList are initialised and the `supportMapFragment` is initialised with the reference of the map fragment. Furthermore, the `mapView` is initialised with the `ViewModelProvider` using the `requireActivity()` method.
After the MapView is created, the `onViewCreated()` method is called. In this method the `getMapAsync()`-method is called in the `supportMapFragment` object with an `onMapReadyCallback` object as the input.

```
    @Override
    public void onMapReady(GoogleMap googleMap) {
        googleMap.setLatLngBoundsForCameraTarget(SDU_PARKING_BOUNDS);
        ...
        checkPermission();
        googleMap.setMyLocationEnabled(true);
        ...
            mapVM.setLocationParked(SessionData.getInstance().getLocation());
        ...
```

Listing 6: *MapView: OnMapCreated(GoogleMaps googleMaps)*

The `Callback` object set by the `getMapAsync()`-method indicates the readiness of the `GoogleMap`. Subsequently, the `onMapReady()`-method is called, when the `onMapReady-Callback` provides a response (see Listing 6). Furthermore, the boundaries of the map are set to the parking area of SDU by `GoogleMap.setLatLngBoundsForCameraTarget()`. To enable the location of the device (`googleMap.setMyLocationEnabled(true)`), the permission to access the location has to be checked (`checkPermission()`). In this case the `ACCESS_FINE_LOCATION` is checked. Lastly, when the last location of the device is queried from the `SessionData`, the location is sent to the MapViewModel. This is regularly called.

```
private void setMapRequiredObservers() {
    mapVM.fetchParkingLotsCoordinates(getContext()).observe(requireActivity(),
     ↪  new Observer<List<IParkingLot>>() {
        @Override
        public void onChanged(List<IParkingLot> parkingLots) {
            ...
            if (!savedParkingLots.contains(parkingLotCoords)) {
                savedParkingLots.add(parkingLotCoords);
                ...
                MarkerOptions markerOptions = new
                ↪  MarkerOptions().position(parkingLotCoords).icon(
                BitmapDescriptorFactory.
                        defaultMarker(color));
                markers.add(map.addMarker(markerOptions));
            } else if (saveParkingLots.contains(parkingLotCoords)) {
                for (Marker marker : markers) {
                    ...
                }}
            ...
```

Listing 7: *MapView: Observer on fetchParkingLotsCoordinates()*

In the first part of the method `setMapRequiredObservers()` an observer is set on the `fetchParkingLotsCoordinates()`-method from the MapViewModel to the observable `MutableLiveData` (see Listing 7). Whenever a change in the list of parking lots in the database occurs, the method `onChanged(List<IParkinglot>)` is called and obtains a list of `IParkingLot`. Subsequently, the method iterates through all the objects in the list of `IParkingLot`. If the coordinates of the objects are not part of a locally saved list of `LatLng`, the object is added to this list. Moreover, markers are added to the map with a colour depending on if the spot is occupied (red) or free (green) and also are added to a local list of markers. If the `ParkingLot` object is part of the locally saved list of `LatLng` then the method runs through all `marker` objects of the markers list and if the availability of the `ParkingLot` object is true, the color of the `marker` is set to green. Otherwise, it is set to red. Furthermore, if the location is equal to the coordinates of the `Parkinglot` object, the color is set to blue.

```
private void setMapRequiredObservers() {
    ...
    mapVM.getLocationParked().observe(requireActivity(), new Observer<Location>()
    ↪  {
        @Override
        public void onChanged(Location location) {
            IParkingLot parkingLot = mapVM.calculateNearestMarker(location);
                ...
                for (Marker marker : markers) {
                    if (marker.getPosition().equals(coordinates)) {
                        marker.setIcon(BitmapDescriptorFactory.defaultMarker(
                        BitmapDescriptorFactory.HUE_BLUE));
                        parkingLot.setAvailability(false);
                         ↪  SessionData.getInstance().setCurrentParkingLot(
                        parkingLot);
                        mapVM.updateParkingLot(parkingLot, getContext());
                    ...
```

Listing 8: *MapView: Observer on getLocationParked()*

In the second part of the `setMapRequiredObserver`, an observer method `getLocationparked()` in the MapViewModel is set to observe the `MutableLiveData` (see Listing 8). Therefore, if the user updates his/her location by parking (see MapViewModel) the `onChanged()`-method is called and obtains a location. Furthermore, the method `calculateNearestMarker()` is called, passing on the location. Subsequently, the `onChanged()`-method runs through all `marker` objects of the list `markers`. If the position of `marker` is equal to the position of `parkinglot` (`coordinates`) the color of the marker is set to blue. Besides, the `avalability` of `parkingLot` is set false and the `SessionData` and the database (through MapViewModel) are updated.

An example of the Map can be seen in figure 5 in the *Appendix*.

5.3.2 ViewModel

In the MapViewModel the data is passed on from the MapView to the MapModel.

```
public LiveData<List<IParkingLot>> fetchParkingLotsCoordinates(Context c) {
    ...
    fetchCoordinatesFromModel(c);
    return allParkingLots
}
private void fetchCoordinatesFromModel(Context c) {
    mapCoordinateService.fetchCoordinates(new Callback() {
        @Override
        public void onResponse(Object o) {
            allParkingLots.postValue((List<IParkingLot>) o);
        }
    }, c);
}
```

Listing 9: *MapViewModel:Fetching parking lots from model*

In Listing 9 the method `fetchParkingLotsCoordinates()` is presented. It calls the `fetchCoordinatesFromModel()` method. Within this method a `Callback` is set and on the response of this, the method `onResponse` is called, obtaining an `Object`. As this `Object` is of the type `IParkingLot`, it can be casted to a list of `IParkingLot`. Finally, this list is returned.

```
public IParkingLot calculateNearestMarker(Location location) {
    ...
    for (IParkingLot parkingLot : allParkingLots.getValue()) {
        Location.distanceBetween(parkingLot.getlatitude(),
                parkingLot.getlongitude(),
                location.getLatitude(),
                location.getLongitude(),
                results);
        if (results[0] < min) {
            min = results[0];
            returnParkingLot = parkingLot;
        }
    }
    return returnParkingLot;
}
```

Listing 10: *MapViewModel:calculate the nearest Marker to the location*

Lastly, the method `calculateNearestMarker(Location location)` will be explained. Within the method, it iterates through all object of a local list of `IParkingLot` and calculates the distance between the coordinates of this object and the location given as a parameter. Conclusively, the methods calculates the minimum of the distance and returns it (see Listing 10).

5.3.3 Model

The MapModel in this project includes *Data Objects* such as *ParkingLot* and *ParkingLot-Dao* and a service *MapCoordinateService*.

```
public void fetchCoordinates(final Callback callback, final Context c) {
    new AsyncTask<Void, Void, Void>() {
        @Override
        protected Void doInBackground(Void... params) {
            ParkingLotDao parkingLotDao =
            ↪  Common.getInstance().getDatabase(c).parkingLotDao();
            List<ParkingLot> parkingLots = parkingLotDao.getParkingLots();
            callback.onResponse(parkingLots);
        }
    }.execute();
}
public void updateParkingLot(final IParkingLot parkingLot, final Context c) {
    new AsyncTask<Void, Void, Void>() {
        @Override
        protected Void doInBackground(Void... params) {
            ParkingLotDao parkingLotDao =
             ↪  Common.getInstance().getDatabase(c).parkingLotDao();
            parkingLotDao.updateParkingLot(parkingLot.getAvailability(),
             ↪  parkingLot.getlatitude(), parkingLot.getlongitude());
        }
    }.execute();
}
```

Listing 11: *MapModel: fetch the coordinates and update parking lot*

Listing 11 depicts two methods. The first method, `fetchCoordinates()`, fetches the coordinates from the database. Thereupon, `fetchCoordinates()` starts an `AsyncTask`, which can run without interferring with the main thread. In this task a `ParkingLotDao` object (`parkingLotDao`) is created and the location of `parkingLotDao` is written into a list of `ParkingLot`, which then is returned in the `callback` response.
The second method `updateParkingLot()` works similar. It starts an `AsyncTask` and creates `ParkingLotDao parkingLotDao`. It then updates the `parkinLotDao` with passing on `latitude, longitude` and `availability` of the `parkingLot`.

5.4 Parking

The core functionality of the application happens at the Parking fragment, where the buttons to park and leave are located. The design of the fragment is shown both for the parking and leave button in the *Appendix* in figure 6 and 7, respectively.

5.4.1 View

When the Parking fragment is initialized, it is first checked whether the user has already parked. The parking button will be replaced with the leave button after the user has parked. Listing 12 shows a code snippet of the `onViewCreated()`-method.

```
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle
↪  savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
    ...
    btn_park.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            updateLocation();
            IUser user = SessionData.getInstance().getCurrentUser();
            String userCarType = user.getCar_type();
            if (userCarType.equals(CARTYPE.Electric.toString()) ||
            ↪   userCarType.equals(CARTYPE.Hybrid.toString())) {
                user.gainCoins(10, getContext());
            } else {
                user.gainCoins(2, getContext());
            }
            user.setParkedStatus(true, getContext());
            Animation animation = new AlphaAnimation(0.0f, 1.0f);
            animation.setDuration(500); animation.setStartOffset(20);
            btn_park.startAnimation(animation); btn_park.setText("PARKED");
            Handler handler = new Handler();
            handler.postDelayed(new Runnable() {...}, 5000);
        }});
    ...
```

Listing 12: *`onViewCreated()` and the parking button.*

A String `userCarType` is created to get the car type of the user that is logged in. If the car type is an electric or a hybrid car, ten coins will be given each time the user parks his/her car. Two points will be given to users with other types of cars to encourage all people to use the application. Furthermore, the user's parked status is set to true.
Animation will occur when the parking button is clicked. The button will blink and change to ”PARKED” after 0.5 seconds. Afterwards, the leave button will appear after five seconds. This should be about a minute beyond this prototype to ensure that the user has left the car after parking. When the user decides to leave the parking lot, the same is done to replace the leave button with the parking button, except the `setParkedStatus()` is set to false and the occupied parking space is available again.
The location of the user's parked car will be updated to the map. Listing 13 shows the location update.

```
private void updateLocation() {
    checkPermission();
    fusedLocationProviderClient.getLastLocation().addOnCompleteListener(new
    ↪   OnCompleteListener<Location>() {
        @Override
        public void onComplete(@NonNull Task<Location> task) {
            Location location = task.getResult();
            if (location != null) {
                try {
                    IUser user = SessionData.getInstance().getCurrentUser();
                    user.setLocation(location, getContext());
                    SessionData.getInstance().setLocation(location);
                    ...
                } catch (IOException e) {
                    e.printStackTrace();
                }}}});}
```

Listing 13: *`updateLocation()` of the user's car.*

The permission to access the location service on the user's mobile phone is acquired. When the `getLastLocation()`-method has acquired the location, then location is set to the user who is logged in. The latitude and longitude of the parked car is acquired and set to a text field below the parking button, which will be displayed for the user to see.

5.4.2 Model

There is no ViewModel for the Parking fragment, as the fragment is mostly used for user interaction. However, information is acquired from the `SessionData` class, which is within the acquaintance. Data is fetched from and submitted to the `SessionData` file, which have common reachability from View, ViewModel and Model accross all classes. The `SessionData` works as a class to get and set data within the Model. The `User` class is within the Model, which is the class that Parking has to communicate with through the `SessionData`. The `IUser` interface is implemented by the `User` class.
In the `onViewCreated()`-method, the `User` is utilized to get the current user's parked status to know which button to show. The current user is also used to access the car type. In the `updateLocation()`-method, the current user is fetched and the location of the car is set to the newly updated location. When the user leaves the parking space, the parked status is set to false together with the parking space for it to available for another user.

5.5 Acquaintance

The Acquaintance package is a very useful addition to the MVVM-architecture, as it enables all layers to communicate with common classes and methods, without having a reference to each other. Within the Green Parking Spotter the Acquaintance package is used to hold interfaces, `IParkingLot` and `IUser`, for the two data objects. These two interfaces make it possible to use `ParkingLot` and `User` objects in all layers without knowing about any specific implementations from the Model, where the actual data objects are defined. In addition, Green Parking Spotter also uses a `SessionData` class that is used to store important data for each session. This for instance enables the application to always being able to retrieve a reference to the currently logged in user and his/her location, besides also being able to get a reference to the parking space that the user is parked in. All of this makes it possbile for Green Parking Spotter to use and manipulate data from all layers of the MVVM-architecture.

5.5.1 Application Database

The Acquaintance package is also fitted with a `Common` class, where common methods are located. This helps the developers writing less duplicated code, by being able to store often used methods that share logic with multiple use cases in said class. Furthermore, the `Common` class holds a method that provides a reference to the Green Parking Spotter database, which the different "Service" classes in the Model layer can benefit from when trying to connect to the application database. The database itself is also defined in the Acquaintance; however, this location could be anywhere, as it is only the `Common` class that needs a reference to the database to retrieve the connection for it.

```
@Database(entities = {User.class, ParkingLot.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
public abstract UserDao userDao();
public abstract ParkingLotDao parkingLotDao();
}
```

Listing 14: *Acquaintance:AppDatabase*

Looking at the above code in Listing 14, the first line clearly indicates that the class is defined as a `Database`, containing the two entities `User` and `ParkingLot`. Specifying the two entities will create a table for each entity with the name defined in each of the classes. Following is an example from the `User.class` where the `User` entity is defined with `tableName` and a constraint on the column `student_mail` being unique. Furthermore, this is where database logic such as `PrimaryKey` and other constraints are defined.

```
@Entity(tableName = "users", indices = {@Index(value = "student_mail", unique
↪  = true)})
public class User implements IUser {

@PrimaryKey
public Integer uid;

@ColumnInfo(name = "student_mail")
@NonNull
```

```
public String student_mail;

@ColumnInfo(name = "password")
@NonNull
public String password;

@ColumnInfo(name = "car_type")
public String car_type;
...
```

Listing 15: *Model:User*

Finally looking at the last three lines from Listing 14, the class `AppDatabase` extends `RoomDatabase` specifying that this is a ”RoomDatabase”, which is a local database built for Android that stores your data securely in a `sqlite` database locally on your device. This means that other devices will not be able to communicate with each other, but for the purpose of the prototype, that is Green Parking Spotter, a `RoomDatabase` will suffice especially as it is very easy to setup and use. The last two lines in Listing 14 specifies two abstract methods that when implemented will provide a reference to a `Dao`, short for ”Data Access Object”. This can be used to run predefined queries towards the database. An example of the interface `UserDao` is provided in Listing 16.

```
@Dao
public interface UserDao {
@Query("SELECT * FROM users WHERE uid=:Id")
List<User> getUserById(int Id);
...
```

Listing 16: *Model:UserDao*

Here the method `getUserById` takes an integer `Id` as parameter, and uses this parameter in the predefined query that is specified in the above `@Query`. The interface is also tagged with the `@Dao` tag, which specifies that this is a `Dao`. This implies that the underlying framework will implement the methods defined within the `Dao` and provide logic that executes the written query against the database.

6 Discussion and Evaluation

In this section the results obtained in this project will be discussed and evaluated.

Among many quality attributes (QAs), three were chosen as important for this project. Those are *performance*, *availability* and *usability*. A quality attribute scenario was devel-

oped for each of the attributes, which enabled an understanding of what challenges that could be present.
The performance QA has been accounted for in several ways, one being many users parking in the morning. Incoming parking requests are to be put into a queue and executed in the same order to eliminate the possibility of two users requesting the same parking space. However, this was not obtained as a local database was used. The implemented method for acquiring the user's location and parking at a parking space has not been optimised for performance.
The quality attribute scenario regarding availability considers accessing the map overview within two seconds. This has been obtained below two seconds.
In the current solution for the application, accounts are created on a local database within the user's mobile phone. Therefore, the user is not able to login from another phone, which lowers the availability of the application.
The usability of the application has been obtained by saving GPS coordinates at 0.000001 precision to distinguish between the parking spaces. This will however only distinguish between the actual coordinate sets for each parking space, and can result in miscalculations if the user steps out of the car to park. The usability is also prevailed within the design of the application as well, as the user is presented with a simple graphical user interface and a menu to simply navigate within the fragments.

When it comes to the tactics, the aim was to apply the resource adaptability. Resource adaptability enables the system to switch between resources to find the best resource. By using the `getLastLocation()`-method, the system accesses two resources for the location access. One is the GPS-based location access and the other one is the access via the internet connection. Therefore, the system automatically chooses the best resource for the location. Thus, the tactic of resource adaptability was applied.
Another tactic, which could be taken into account, is the privacy tactic. After logging in with an account, the username and the password are visible in the about section. This contradicts with the aspect of *Hide*, which tries to prevent data to be visible. Besides, the tactics energy efficiency was not applied either. While programming, this tactic was not taking into account, which could or even must be changed for further implementation.

Mobile sensing has been the foundation during the development of the prototype Green Parking Spotter. The project has utilised location data to enable parking of a car at a specific parking space, based on which parking space is closest to you. However, since the width of the parking spaces is only a few meters, calculating the precise parking space, which is actually closest to you, can be troublesome. Depending on the location of the user, when he/she presses the park button, the results can vary. For instance, if the driver

presses the park button, then the location will be slightly more to the left side of the parking lot, and if the driver steps out the vehicle before pressing park, it will be even more to the left side, making it hard to always select the correct parking space, as the shortest distance could now be to another parking space.
Currently Green Parking Spotter only has a single set of latitude and longitude coordinates for each parking space, so a possible fix would be to instead save an entire set of coordinates defining the entire parking space, which would have increased the accuracy of the parking space calculations drastically.

7 Conclusion

In this project the prototype Green Parking Spotter has been developed. The goal for creating this application was to encourage the usage of green cars over conventional cars at SDU, while providing useful features for all students and employees driving to and from SDU. Currently no application that achieve those goals exists, and SDU has reached out to students to help them come up with ideas for a possible solution.

In order to pursue the overall goal, four main features were included and applied:

- **Login**
  The user can create an account and specify his/her car type, making it possible to differentiate between green and conventional cars.
- **Parking**
  The user can press the park button, which sets the location that the map can use to occupy the parking space physically parked on and mark it within the map. Furthermore, the leave button becomes available, which will release the occupied parking space when pressed.
- **Map**
  The map displays the different parking spots (occupied and free), the location of the current user and the parked spot of the current user. The map is automatically zoomed in to the parking lot when the user parks.
- **Perks**
  Every user is able to redeem perks with coins obtained by parking at the SDU parking lot. Parking with a green car increases the coins obtained.

Finally it can be concluded that the Green Parking Spotter application achieves the requirement of providing benefits for people using green cars at SDU. In addition, extra features such as displaying availability for parking spaces have been implemented, to provide extra helpful features besides the benefits for green cars. This improves the quality of life for all students and employees driving to and from SDU.

8 Future Works

As this project was intended to be a prototype and is an university project, the time frame was limited and therefore, the resulting application lacks some features. Thus, the following section considers future implementations and features that should be developed and might improve the application.

A part of the aim of this project beside implementing benefits for users with green cars, was to prioritize or reserve a certain number of parking spaces for these users. This was not obtained, but however, it should be implemented in a future implementation. Within a parking lot a few rows of parking spaces could have been reserved for green cars. This would probably ensure an increase of green cars, but still have the possibility to use conventional cars. Furthermore, over time the number of green parking spaces could be increased relatively slow to make people adapt to the idea of green cars.

If the application had to be released as an actual application, a server with an online database would be required. This would enable users to see where other users have parked on the map, and actually make the application work in a real-life scenario. The server could be setup with a RESTful api, with endpoints to send and receive data to/from. The server would then update the database as required.

Up to this state of the project, only two rows of the parking spaces from a single parking lot can be used for parking. However, in a future implementation all parking spaces for all parking lots of SDU would have to be inserted into the database. Thus, making it possible to use the application for all parking lots at SDU.

9 References

Notes

1. Facts About Denmark's Emission of Greenhouse Gasses and Energy Consumption. (2020, 21. October). Statistics Denmark. https://www.dst.dk/da/Statistik/bagtal/2018/2018-12-06-fakta-om-danmarks-udledning-af-drivhusgasser-samt-energiforbrug

2. Danish Companies Emits More CO2. (2020, 18. May). Statistics Denmark. https://www.dst.dk/da/Statistik/bagtal/2020/2020-05-18-danske-virksomheder-udleder-mere-CO

3. Baun Kjærgaard, M. (s.d.). Quality Attributes - Lecture 4 [Lecture slides]. Software Engineering of Mobile Systems.

4. Lecture materials of the course Software Engineering of Mobile System, Lecturer: Mikkel Baun Kjærgaard, Semester 20/21 at SDU

5. Lecture materials of the course Software Engineering of Mobile System, Lecturer: Mikkel Baun Kjærgaard, Semester 20/21 at SDU

6. Android Developers, "FusedLocationProviderClient: Android Developers", Android Developers [Online], Available: https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderClient [Accessed: 1-Dec-2020]

7. Android Developers, "FusedLocationProviderClient: Android Developers", Android Developers [Online], Available: https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderClient [Accessed: 1-Dec-2020]

8. Separation of concerns. (s.d.). Wikipedia. Localized on December 9th 2020 at https://en.wikipedia.org/wiki/Separation$_of_concerns$

9. Android Developers, "ViewModel Overview : Android Developers," Android Developers. [Online]. Available: https://developer.android.com/topic/libraries/architecture/viewmodel.htmljava. [Accessed: 05-Feb-2020].

10. Android Developers, "Android Jetpack: ViewModel," YouTube. 14-May-2018 [Video file]. Available: https://www.youtube.com/watch?v=5qlIPTDE274t=30s. [Accessed: 05-Feb-2020].

11. L. Maciaszek, B. L. Liong, and S. Bills, Practical Software Engineering: A Case Study Approach. Harlow: Pearson/Addison-Wesley, 2005, chapter 9.

12. Introduction to Activities. (s.d.). Developers Android. Localized on December 13th 2020 at https://developer.android.com/guide/components/activities/intro-activities

13. Android Developer, "Fragment Lifecycle: Android Developers", Android Developers, [Online], Available: https://developer.android.com/guide/fragments/lifecycle [Accessed: 01-12-2020]

10 Appendix

10.1 Overview of the Electronic Appendix

- SEMS_Report_GroupJ.pdf
- SEMS_DemoPresentation_GroupJ.mp4
- SEMS_ApplicationPrototype_GroupJ.zip

10.2 Images of Fragments

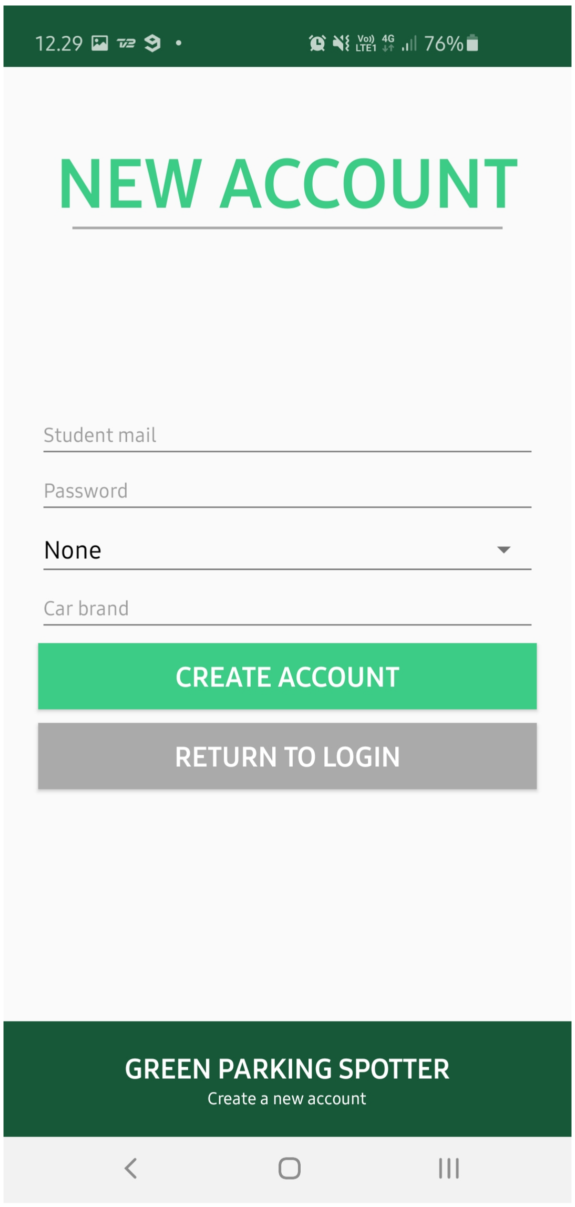

Figure 3: *The design of the CreateAccount fragment.*

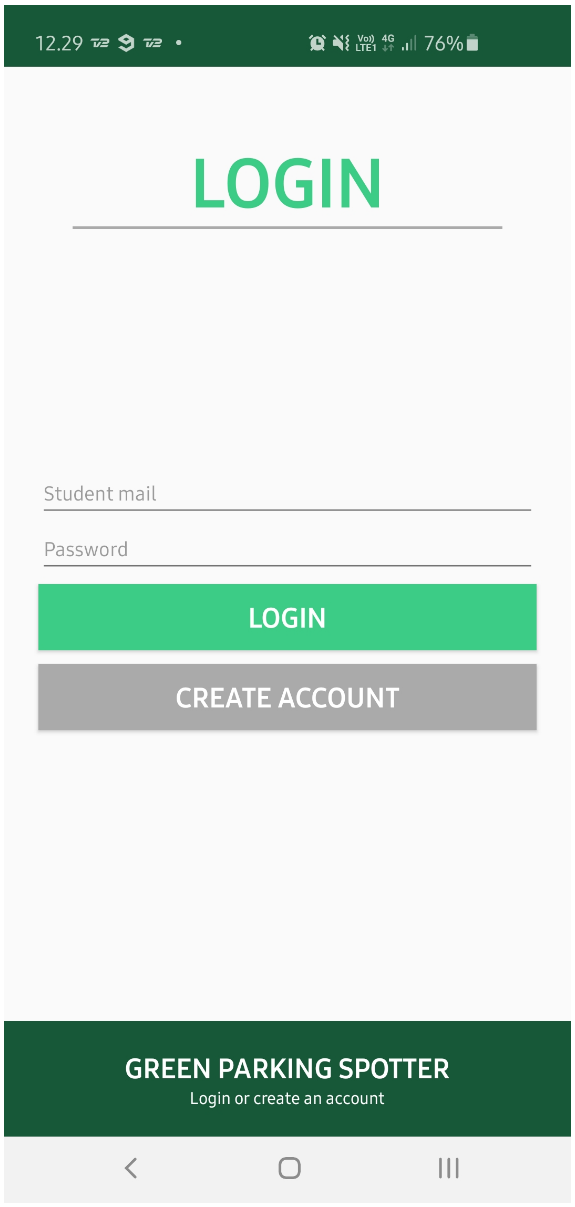

Figure 4: *The design of the Login fragment.*

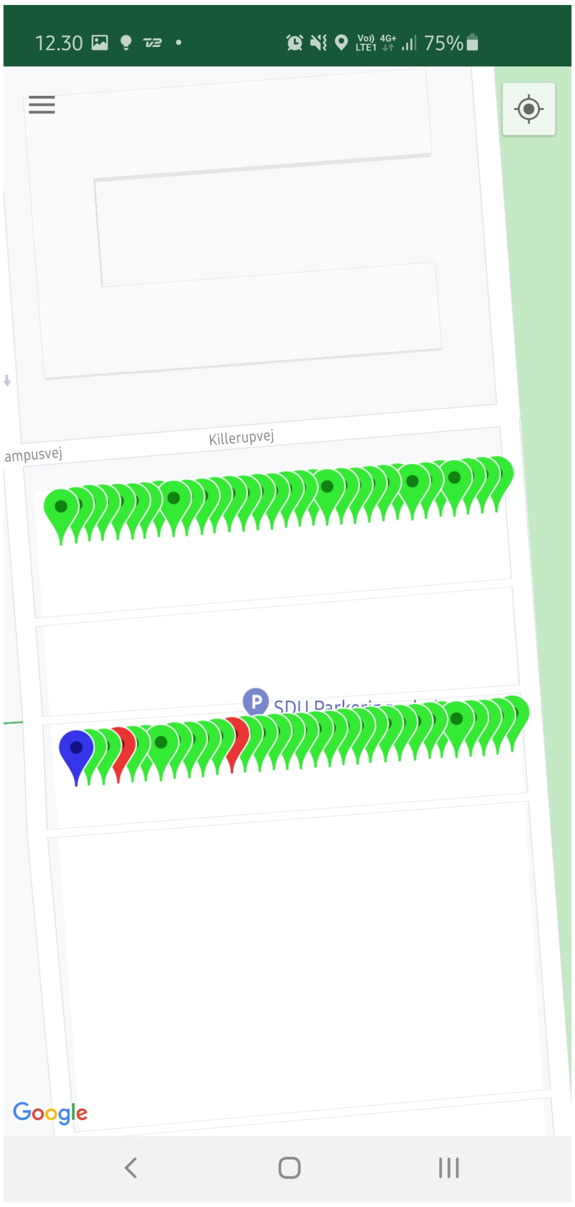

Figure 5: *The design of the Map fragment.*

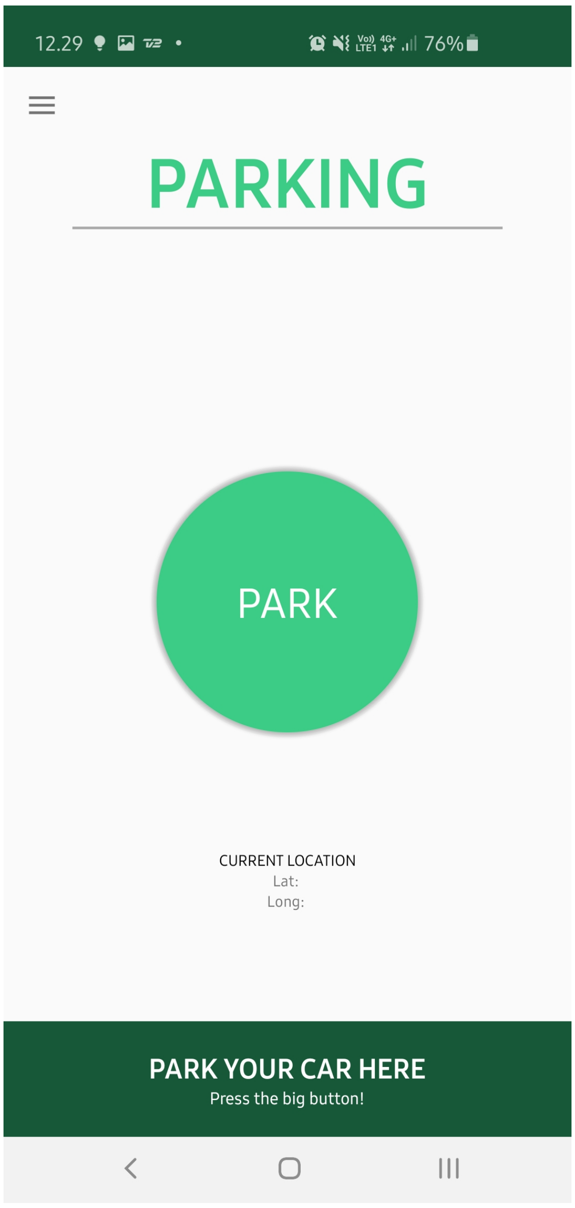

Figure 6: *The design of the Parking fragment - Park.*

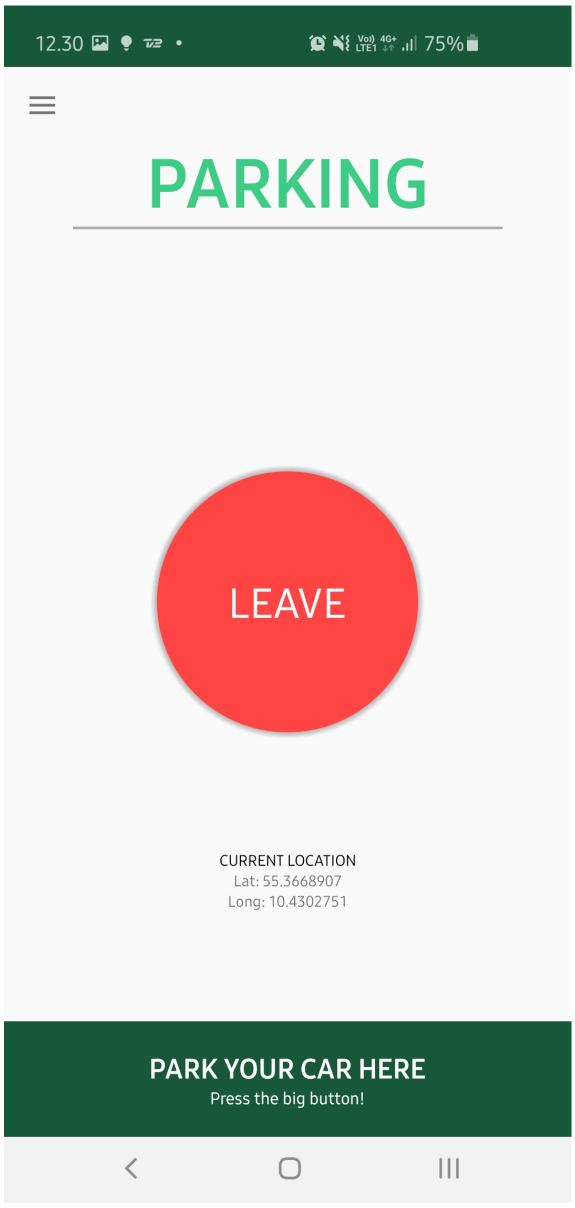

Figure 7: *The design of the Parking fragment - Leave.*