AALBORG UNIVERSITY
STUDENT REPORT
First Year of Study
P0 project, 1st semester
A. C. Meyers Vænget 15
2450 København SV.
http://tnb.aau.dk

Title: WeRoom

Theme: System Development
Project period: 26/03/19 - 27/05/19

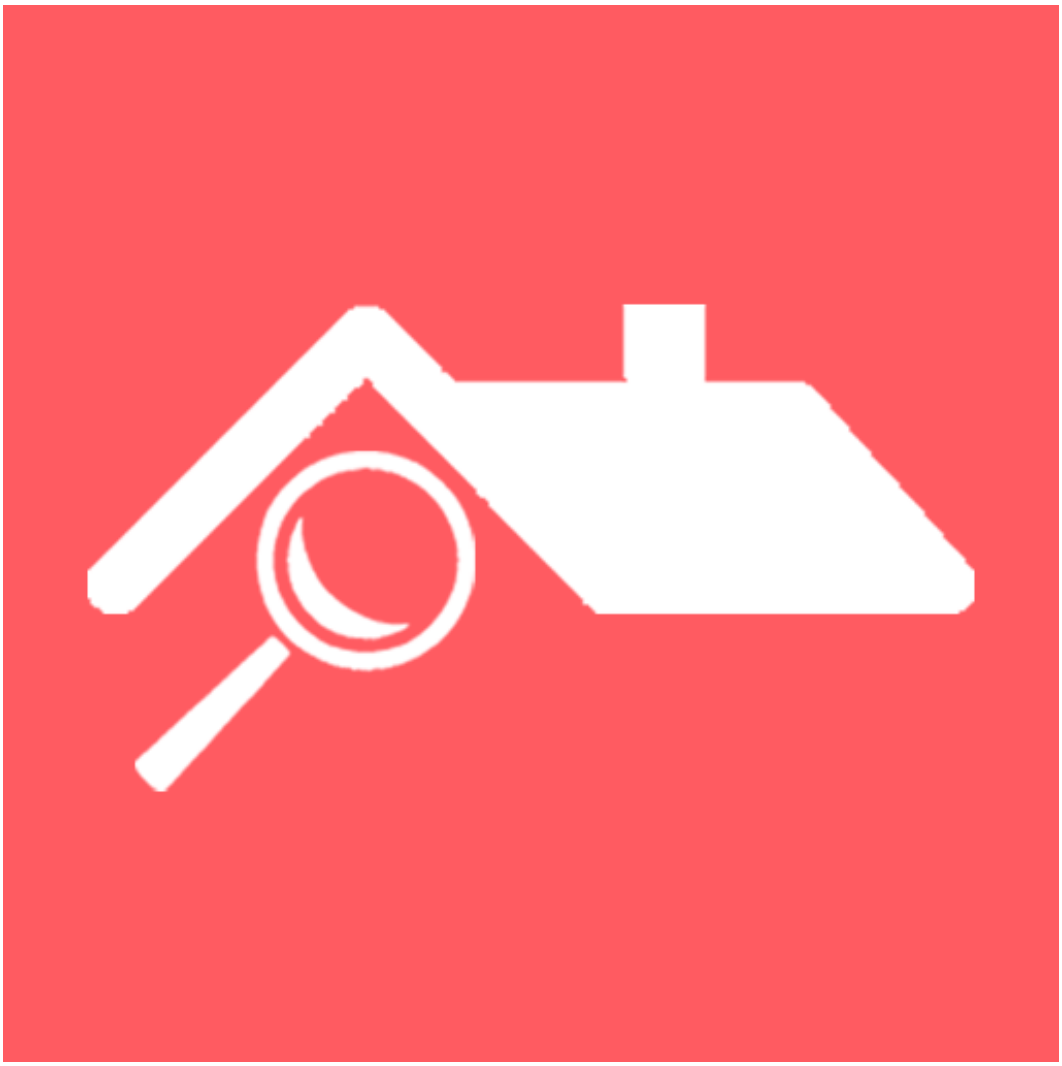
Project group: ITCOM 202
Members:
Patrick Vibild
Demira Dimitrova
Christian Holfelt
Loredana Cosma

Supervisor(s):
Lene Tolstrup Sørensen

Synopsis:

Finding a place to live can sometimes be a long process, especially in the bigger cities. Therefore, this project was made to help connect tenants and landlords in an easy and manageable way.
Being inspired by other rental sites, we came up with a more unique idea, where combining the swiping concept with room rental would play an essential role in the project.
We analyzed the rental market and performed interviews with both tenants and landlords to find the requirements needed for the application.
After the implementation was done, we performed the testing to find bugs and potential improvements to our application.

Copies:
Pages: 123 pages
Annex number and kind: 40 pages
Finished: 27/05/2019



WeRoom

By Demira Dimitrova, Patrick Vibild, Christian Holfelt and Loredana Cosma

May 26, 2019

Table of Contents

Chapter 1

Introduction

Finding a room to rent can be hard, whether it is a foreign country you want to move to or not. As tenants, people can find themselves writing numerous messages without getting a response, whereas, from the landlords' view, they receive hundreds of messages from tenants who mostly do not meet their requirements. The bigger the city, the bigger the demand for rentals.

There are also people who strive to achieve home-ownership, but home-ownership isn't for everyone. It can be hard to decide whether you should buy or rent. Renting has a lot of benefits for those who want financial freedom and flexibility.[1] It can be debated if renting is better than buying and usually, buying your own house is cheaper in the long term. The crash in home prices combined with low mortgage rates made buying and owning a home a cheaper and better investment, than renting. But now the market has changed. Prices for homes are continuously rising and thus making it cheaper to rent than to buy a home.[2] Urbanisation and increasing housing demand are a well-known phenomenon in most western cities and Copenhagen is no exception.[3] Finding a place to rent, whether you are going as an international student, for work or any other reason, can be a challenge. Renting and buying an apartment or a house can be easy if you have the money, but if we take students as an example, they usually cannot afford the expensive ones. They will have to look at different websites and sign up for dorms or cheaper apartments, but the waiting lists are usually more than 2 years long.

Malou Björnslätt who is the administrator of Student and Youth Accomodation Office Copenhagen (KKIK) says: "We experience both international and Danish students who unfortunately have to move back home because they couldn't get accommodation."[4]

These problems can unfortunately not be solved in the short run. Building new dorms/housing is taking years and since the demand is high for affordable places to live, the prices will be high and the waiting lists long. A possible cheap solution for the individual person, is to rent a room. This benefits both the tenant and the landlord. Of course, there will be both advantages and disadvantages for renting a room. Some of the disadvantages could be personality clashes and possible mortgage issues. Nevertheless, it has its advantages and it can be a great solution. First of all, it can boost your income. By two or more people splitting the rent and other expenses, both the landlord and tenant will find it cheaper to live this way. Second, the landlord has spared many of the legal obligations associated with being a traditional one. Lastly, living with someone is always better than living alone. The company can provide you a greater sense of security and if there are some domestic chores which are beyond your ability, you can ask the flatmate for help.[5]

1.1 Problem Formulation

Every project has its purpose, namely to try to solve a real-life problem. This is why, through WeRoom, we decided to come up with an idea for the accommodation problem. As stated in the introduction, finding a place to rent can be a burden nowadays. There are a multitude of websites and applications for renting and yet the problem persists. All of these brought us to one simple question:

How can we make the room renting procedure easier by developing a system for Android users?

Supporting the main question, there are also sub-questions that broaden the understanding of the problem. They go more in depth by questioning what methods should be used in the project.

- How can we use a process model to optimize the implementation of the application?
- How can we use software engineering and design tools to structure the system?
- How can we make the user interface experience better, by following the material design?

1.2 Project Limitations

The project faced different limitations, be it because of lack of knowledge in Android, the requirements our application had to respect or various problems we encountered during the designing and implementing.

Additionally, we had some specific programming requirement that we had to fulfill in our final developed product. Some of those requirement had an impact on the decision made during the design of the application, so the final application meets the expected requirement.

The final application had the next programming requirements:

- Should have at least 5 activities
- Should use Material Design
- Should support portrait and landscape mode
- Should use Shared Preferences
- Should make use of fragments
- Should use at least one AsyncTask or Handler
- Should store data in a database
- Should use Firebase for user authentication
- Should use at least one of the sensors (camera, NFC, GPS, etc.)
- Should have a functionality that does not execute if the battery is less than 10%
- Should have a functionality that does not execute if the network is not WiFi

Some requirements, like using Firebase as authentication, had influenced the project greatly for all the design around users account was made around Firebase. In an attempt to accomplish the programming requirements, we created some software requirements that are not necessarily needed, but follow the specifications above. Having a functionality that would not execute when the phone had less than 10% of battery is an example of this.

During the design of the application, we encountered a lack of knowledge of User Interaction, Experience and graphic design skills. Trying to counter this lack of knowledge, we designed the application and created the blue print of the application following the material design of Android[67] guidelines.

Lastly, we took the decision to use the databases provided by FireBase during the design of the application. During the implementation we discovered that FireBase would only provide non-sql databases. None of the databases provided by FireBase, RealTime database or FireStore, would allow us to make queries with range filters on different fields of a document.[8]

This inconvenience forced the project to either redesign the application and refactor the entire access of the data of the application, or get a list of almost all the document from our non-sql databases and then perform the range filters of the list of documents using the phone resources. We decided to use the non-sql databases provided by FireStore, since we detected the issue with the non-sql queries when half of the application was already developed. The application should either be able to filter all the required data with the queries, or have access to a back-end that would perform this information filtering. We did not have the knowledge about this during the design of the application and created a limitation during the implementation of the project.

Chapter 2

Methodology

After understanding the problem, the next step is to decide on what methods and how we should use them in order to complete the project. This chapter presents all the methods used for this purpose.

We started by organizing some ideas on how we would like our application to be and researching on how other applications solved this issue. Afterwards, we created use cases, user stories and scenarios, and several UML diagrams, in order to have an organized visual representation of our thoughts and ideas. We then understood better what should the behaviour of the application be, as well as its interaction with external actors. All of the above mentioned, along with a series of interviews we took, had their roles in discovering requirements for our project. Furthermore, we have followed the Scrum model for planning and organizing our work on the design and implementation. When the application reached its final form, we have performed different types of testing on it, such as White Box and Black Box. The reason for the testings were both to check if the application runs as it was supposed to and to find what improvements could be made in the future.

2.1 State of the art

The State of Art represents, in our case, all the applications from which we drew inspiration, in terms of design and implementation. This section is very important because it offers a better understanding on which are the most popular applications that offer services that are similar to our idea and what are the common features and design guidelines that we should adapt to our project as well. Moreover, this research is a good source for finding requirements.

2.2 Software specifications

The software specifications describe what the system should do and how the interaction between the user and system should be, before the actual implementation. For this purpose, we set a series of requirements that we had brainstormed about, got from people we interviewed, and from the application design.

2.2.1 Software requirement and its categorization.

Software requirements are conditions regarding operation of the system, called non-functional, and regarding the interaction between the user and the system, focusing on the system's behaviour, called functional[9]. They are usually created after an agreement with the stakeholders is made.

Out of all methods, we chose to use the MOSCOW[10] method to order the requirements by their importance: M for must, S for should, C for could and W for won't. Suggestively, the four categories organize the requirements from the ones without which the application would not work, to the ones which are important, but not vital, to the ones which are desirable, but not important and lastly, to the ones which would fit the ideas of the app, but there are no resources and time to implement them.

The reason for using MoSCoW is to have a good structure for the order in which we should implement the requirements we have. This way, we make sure we do not forget any important features.

Non-functional requirements will be categorized following Ian Sommersville categories.[11] Trying to find

non-functional requirements on each of the categories will ensure that all aspect of the software have been analyzed when creating the requirement.

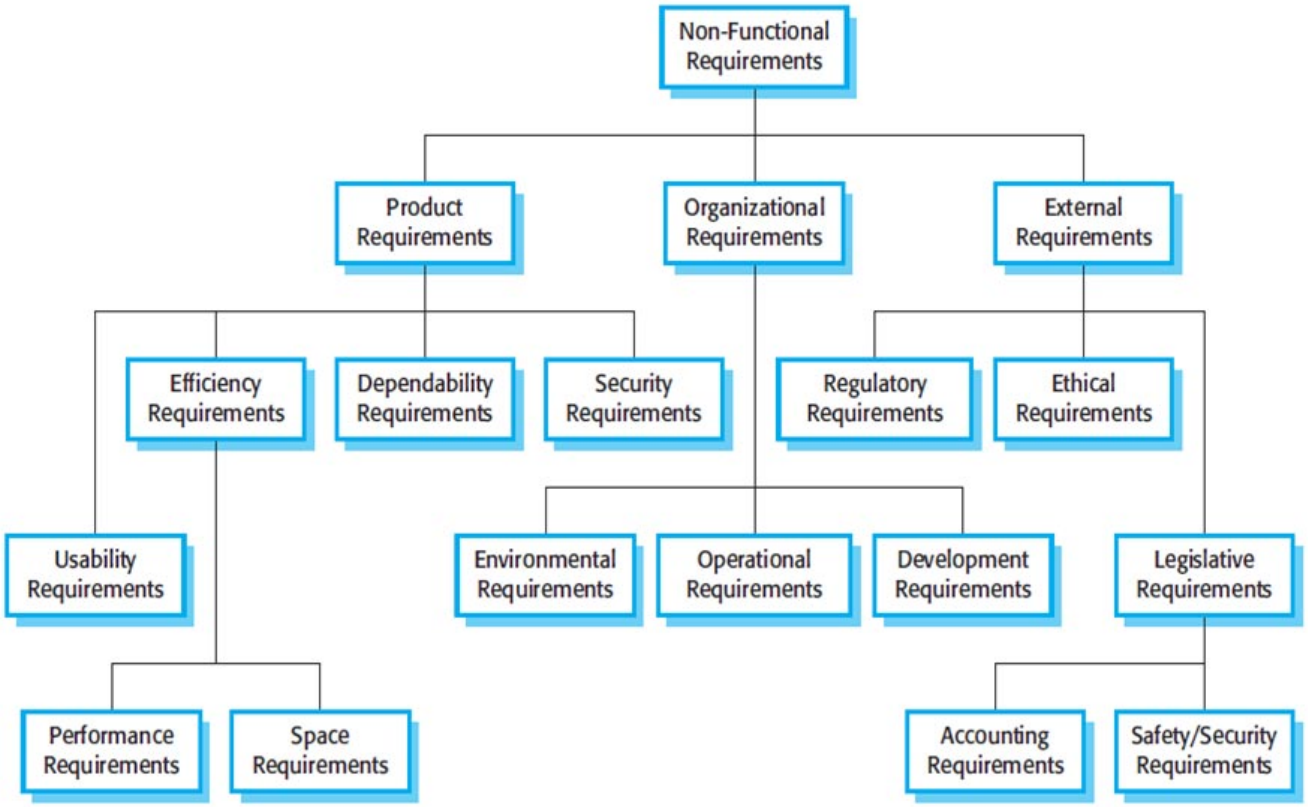


Figure 2.1: Non-functional requirements categorization
[1]

2.2.2 User stories, user scenarios and use cases.

In order to have an organized description of the application, it is a good idea to describe what the users can do and how the system is responding to their actions through user stories, user scenarios and use cases.

The use cases[12] are events described step by step, that a user can perform on an application, as-well as worst case scenarios, where the event does not work as expected. The user stories[13] are short descriptions of features as seen from the user's view. Lastly, user scenarios are similar to user stories, but are more descriptive, supplementing traditional functional requirements[14].
All of these are very useful to us in various ways, such as discovering worst case scenarios, getting more requirements based on them and scheduling sprints.

2.2.3 Interview

We decided to interview people in order for us to find out, what a potential user would expect from our application. From gathering this information, we can get some new requirements and ideas for features and design.

[1] Sommerville, 2011, 9th

We did several interviews with this purpose, and we tried to find people who had previous experience in renting room from perspective of a tenant and a landlord.

In order to cover all the topics we need, we made this set of questions. The following will be used in all interviews with tenants and landlords.

1. What is the experience of the interviewed person as a tenant or landlord? How many rooms did they have to contact as a tenant and how many tenant contacted them as a landlord?
2. What are the top 5 features that you are looking when searching for a new room or tenant?
3. Could you explain how was the experiences in the past finding new room/tenant? Did you use any familiar platform or application? What are the features that you liked in this platform, and what features do you think that you were missing from the platform.
4. How was the communication between you and the landlord or tenant candidates? Did you exchange e-mails? Was it a chat based communication? Phone call etc?
5. What is the opinion about a swiping method to create a match between two users in an application? Are you aware of any application that uses this feature?

   Once introduced our application idea to the interviewed person:

6. What is your first idea with the presented application, does our presented system improve the finding of a new room or tenant? What could be the pros and cons of our presented application?
7. If such an application exist, what features would you like to see in it?
8. What is the most important attributes for a perfect flatmate? Can you tell us something good about previous flatmates how they were, and what you remember from them?

2.3 UML diagrams

In our project we use different kinds of diagrams to get a better overview of our system along with what to prioritize when developing. This gives us a better work-flow, so we make sure that the iterations we are taking are logically consistent.

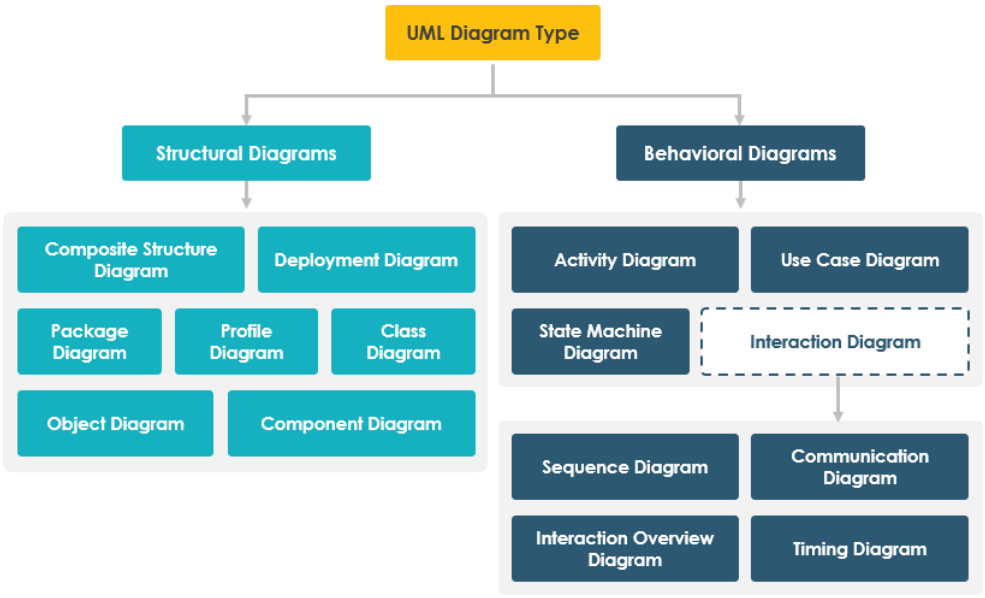


Figure 2.2: Types of UML diagram

In the following paragraphs we will describe the chosen UML diagrams for our project, and we will argument their use.

2.3.1 Context diagram

We use the context diagram to show our system as a whole. It shows the interactions between the system and the terminators which are outside the system, but communicate with the system. The context diagram is created in order to define the boundaries and connections of the designed system. It shows different terminators that interact with the system and describe their functionalities and capabilities.[15]

2.3.2 Use case diagram

The use case diagram is visualizing the functional requirements of our system and gives us an overview of how the actors interact with the system. It lists and organizes the relationship between the interactions that the users might have with the designed system. This will help us identify the internal and external factors, that can influence the system. Moreover, it also helps us to make design choices and development priorities. [16]

2.3.3 Class diagram

We made a class diagram to provide a conceptual model of the system in terms of entities and their relationships. It maps out the static view of our application and the relationship between the different classes. To identify the classes of our application we analyzed the system design text of our application and we identified all the nouns and the relationship between them. Nouns can usually be described as a class in object oriented programming, and describe its content and behavior with another classes in a class diagram.

Class diagrams should be used for: [17]

- Discovering related data and attributes
- Getting a quick overview of the important entities on a system

- Detect if there are too few /many classes in the system
- Identifying if the relationship between the objects is too complex.
- Spotting dependencies between different classes

2.3.4 Sequence diagram

The sequence diagram shows us the interactions between the objects in the order in which the interactions occur. It models a single scenario that is been executed in a system. Sequence diagrams brings a high-level overview of control flow patterns through the designed system. [18]

2.3.5 State diagram

A state diagram is used to describe the behavior of our system with consideration to the possible states of an object when a certain event happens. Usually objects with a long lifetime are chosen to be described with a state diagram, as it can be complex to identify all the possible states that it might have.

2.3.6 Architecture

Software architecture refers to the high level structures of a software system and the discipline of creating such structures and systems. Each structure comprises software elements, relations among them, and properties of both elements and relations[19]

We are following the architectural design, because it creates a link between system design and the software requirements, and therefore having a model that defines how is the system organized, and how its components communicate with each other.

Creating an architectural design brings an overview of the relationship of all the components of our system. We will create a logical architecture that describes all the components used in the design system, and how are the components communicating each other and their dependencies with external services of the implemented code.

The non-functional requirements define the architectural design, making the software easier to maintain, more robust and with higher performance.[20]

The architecture of a software is the blueprint of the system. Architectural changes are costly once the implementation has begun. Therefore, the software architecture plan should be done properly beforehand.

2.4 Process Model

A software process is a set of related activities that leads to the production of the software. These activities may involve the development of the software from the scratch or modifying an existing system.

Ian Sommerville states that all process models follow four activities as follows[21]:

- Software specifications, defining the main requirements of the system that need to be developed.
- Software design and implementation, as the software needs to be designed and afterwards developed
- Software verification and validation, as the programmed software must fulfill the written specification in the previous activity

- Software evolution (Maintenance), as the software can be adjusted to meet the end user and market requirements.

In the development of WeRoom, we will follow a software engineering process model. There are different process models part of the three main groups: Sequential, Iterative and Agile models. We got to the conclusion that an Agile model would fit our work best.

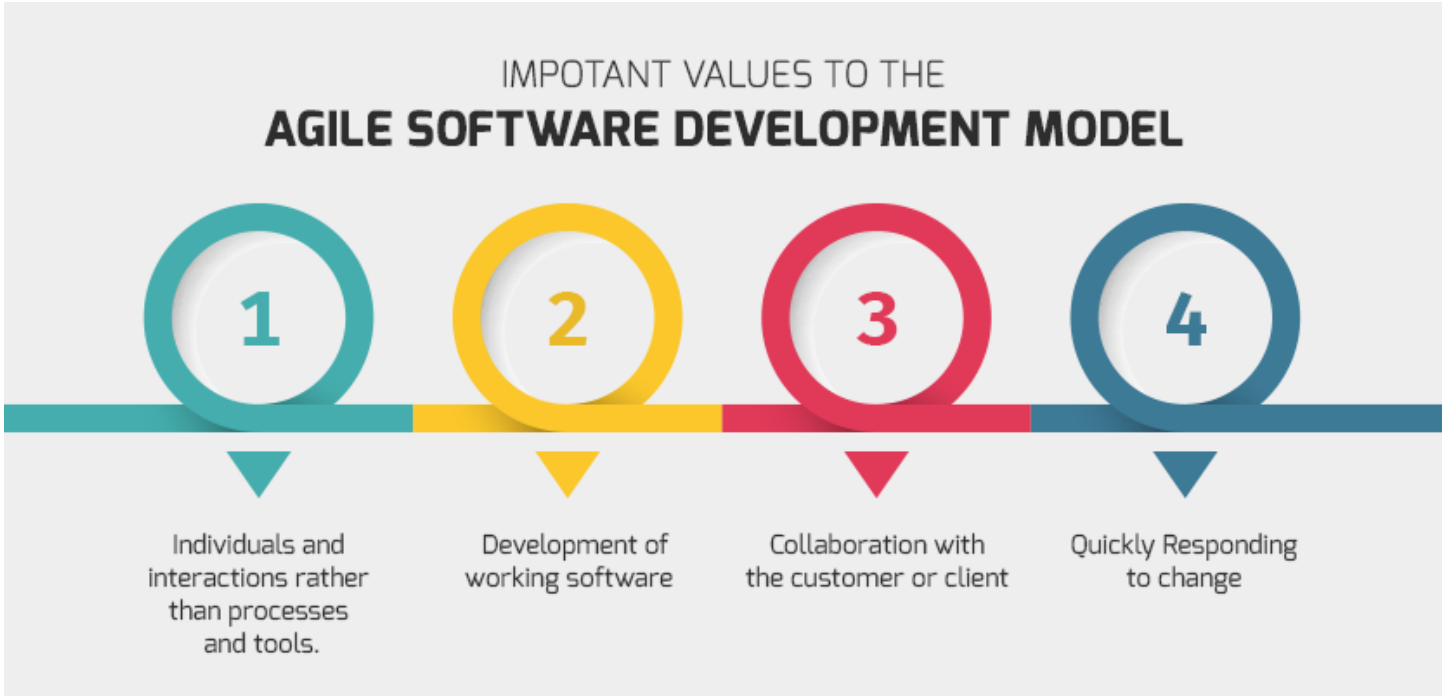


Figure 2.3: Agile software development model[2]

Agile models follow the concept that the software requirements presented at the beginning may change, or even that at the beginning of the development of a system, all requirements would not be known. In order to attempt to solve this issue, the agile models tries to involve the customer of the product in the development, getting feedback on the development periodically.

2.4.1 Scrum

Scrum is a framework which can help a team work together as a unit and organize the work. It is divided into three phases where the first phase is the initial phase. In this phase the planning takes place. The objectives are established for the project along with the design of the software architecture. The second phase is the sprint cycles that span over a given period of time. In the sprint cycles certain tasks should be done within its time span. The last phase is the closing of the project. This phase wraps up the whole project with the completion of the documentation etc. [22] A Scrum usually consist of a product owner, a scrum master to oversee the project and a scrum team to develop the project for the product owner.

Moreover, Scrum has daily meetings to give team members an overview of what the individual member is doing and planning on to finish in the given day. Sprints are also an important part of the scrum. A sprint is a limited time period, which can be from a week to a month depending on the team, where given tasks has to be done within this time period. At the end of every sprint, a sprint review is made.

[2]Agile software process model values: https://www.quickscrum.com/Article/ArticleDetails/3041/3/What-is-Agile-software-development-model

During the review, the team and stakeholders evaluate the work that has been done and identify the next functionalities that should be implemented.

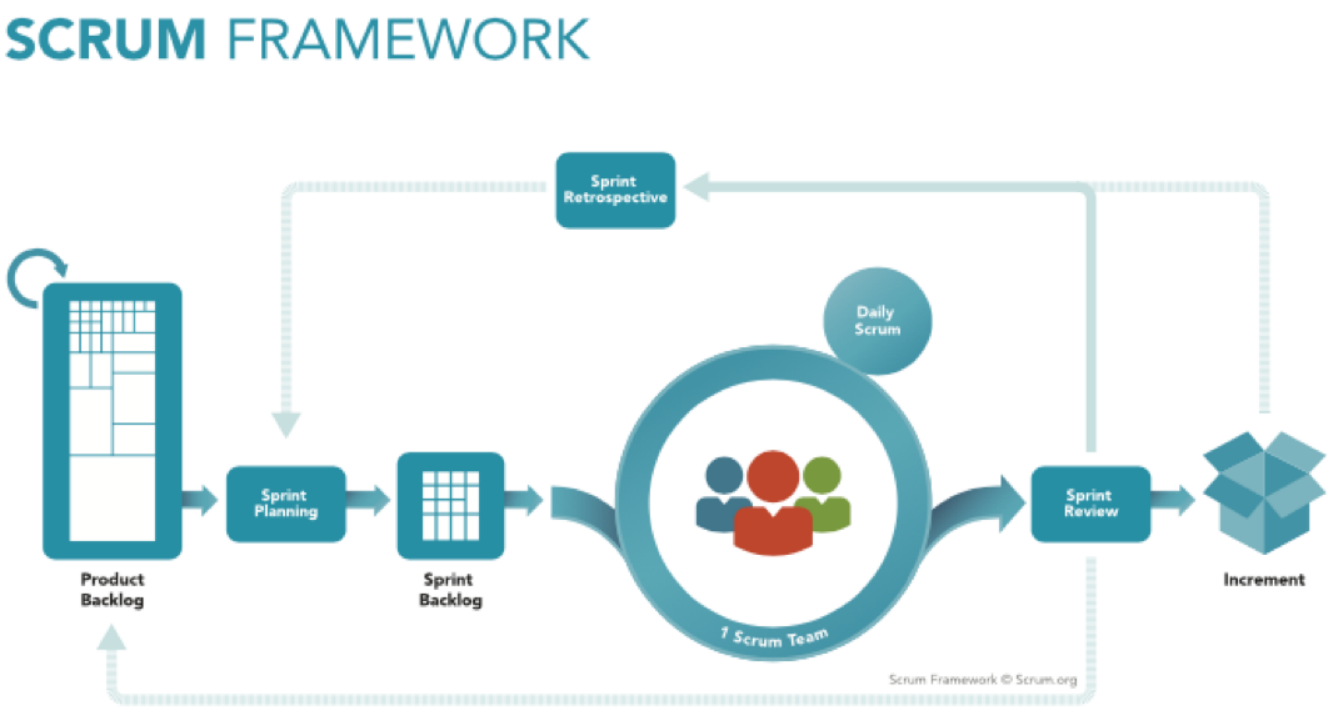


Figure 2.4: Scrum process model illustration [3]

We decided to use the Scrum since it is easy to learn and use. It optimizes the efficiency of the group work because the sprints always put a deadline for the tasks. Moreover it helps with communication within the group. With the daily scrum meetings and sprint-reviews, all team-members are always up to date with who is doing what and when it is planned to be finished.

2.4.2 Kanban

Kanban is a visual system used to manage the work as it goes through a process. Its purpose is to optimize the work-flow and the work among a team. It consists of three steps.

1. To do: An overview of all the tasks that have to be done.
2. Doing: Tasks that the team is currently working on.
3. Done: Tasks that have already been done.

We decided to add a fourth step to our Kanban called *Verify*. Here, the task that has been done can be verified by another team-member before moving it to done, to ensure it complies with our standards.

[3]Scrum framework illustration: https://www.scrum.org/resources/scrum-framework-poster

Figure 2.5: Kanban ToDo board

Figure 2.6: Kanban Calendar board

The work is defined on the Kanban cards and can then those cards are being moved between the different stages. A Kanban card usually includes a description of the tasks and the estimated time it will take to finish that task.

We decided to use the Kanban model to get a visual overview of the tasks so we can manage the work

easier. By using the Kanban model we can see how far in the process the tasks is and which member has to do what before a given deadline.

2.5 Wireframe/prototype

In the early stages of the project, we did the wireframing. With the wireframing, we can have an idea of how the design and layout of our application should look like before we start the implementation. The first part of the wireframing is sketching the ideas on a whiteboard or a piece of paper, and when all members agree on the design, we can begin finalizing it by making a prototype. For this, we have used a program called JustInMind[23], so we can test the prototype on a smartphone.

The main function of the wire framing was to create a blueprint that all developers could follow during the implementation. Wireframing is also used to have a acceptance test for external users, so when we get to the development part of the application we already have validated the flow of the system.

2.6 Design Patterns

Design patterns were developed to solve certain common problems spotted on the design and development of software. The patterns have a big impact on object-oriented programming as they use characteristic from the object-oriented paradigm such as polymorphism and inheritance to bring generality to solve the problems.

Design patterns have became a known part of the basic vocabulary of software developers, allowing different developers to describe a known problem and their corresponding solution to the problem. Those patterns are commonly reused and in some circumstances programming languages include libraries to implements such patterns.[24]

A design pattern consists of four components. This helps the readability and helps developers to detect where they can use one of the design patterns to solve a problem in their software. [25]

1. A meaningful name to identify the pattern.
2. A description and explanation where the pattern can be adapted.
3. A described solution, where a template to the solution is given, describing the components used in the solution and the relationship between the components. This illustrates graphically and shows the relation of the object and classes of the pattern.
4. Description of the trade-offs when implementing the pattern. Developers should know what are the pros and cons of the pattern so they can evaluate in each situation if they should follow it or not.

2.7 Material Design

Material design[26] is used to go more in depth with the design/layout of the application. Many different things have to be taken into consideration regarding the UI (User Interface). Among the things we discussed regarding material design were the colors to use in the application, since each color has a different meaning and impact on the users; what icons to use in the bottom navigation bar for example; movement of the fragments, as we want to implement the swiping feature, and so on. We use the material design as a guideline to design the application's user-interface.

2.8 Software testing

Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use. The testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements
2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification. [27]

In the limits of our project we will perform software testing with the aim to fulfil those two goals.

2.8.1 Acceptance testing

Acceptance testing is software testing where the system is tested for acceptability. The purpose is to evaluate the compliance with the requirements and assess to see if it is acceptable for delivery. [28] Acceptance testing is similar to unit testing since it only has two results, which is pass or fail. The fail does not prove, but can suggest a failure in the product which maybe should be corrected.

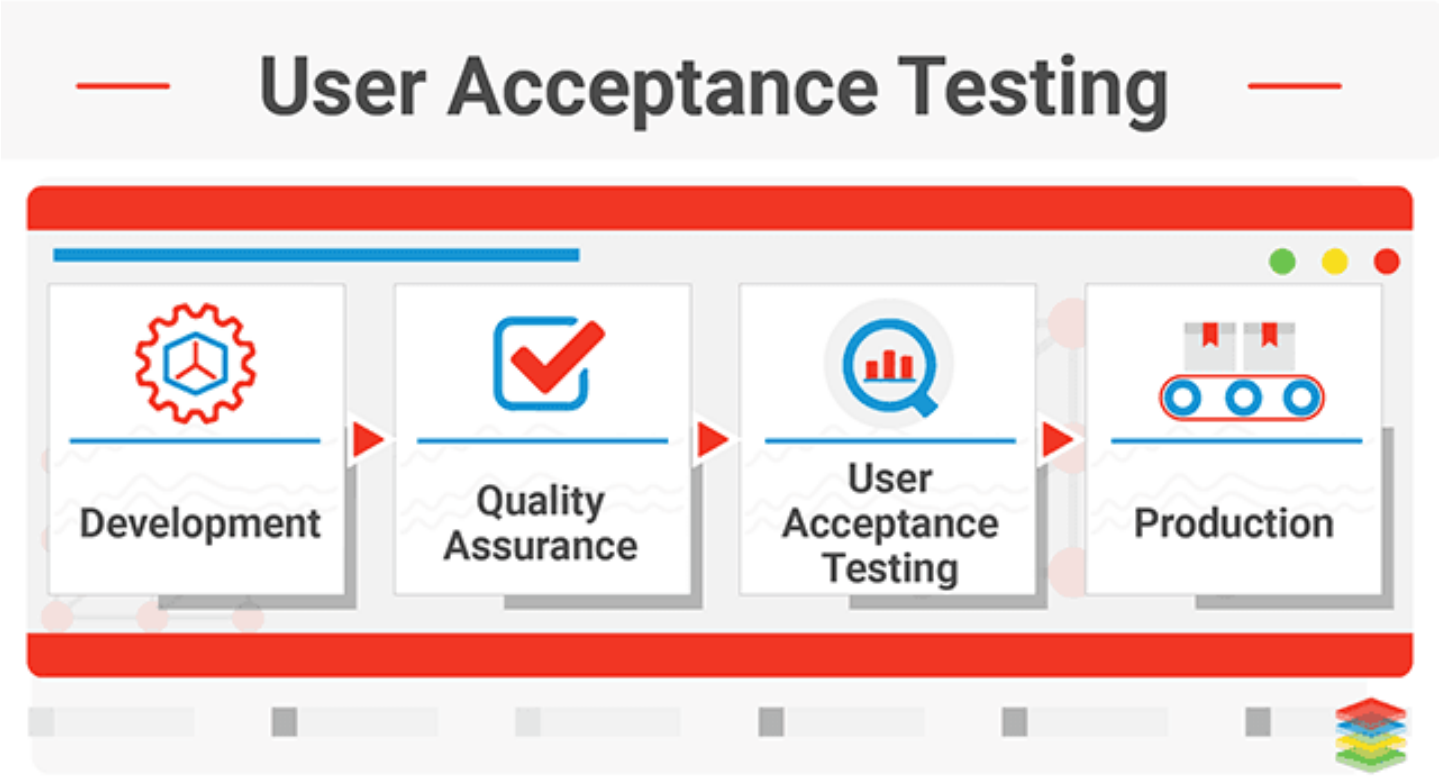


Figure 2.7: User acceptance is the step where code is accepted or rejected before being on production. [4]

There are multiple benefits using acceptance testing, which is why we want to include it in our project. Some of the benefits is that it's helpful with finding bugs, since the users might do something we did not think of. Moreover it can also be helpful with the design. Is there something we are missing? Not enough information displayed? Or it could be something entirely else.

The users testing the application will be from another group in our study line. They will have small idea of the software and its specifications beforehand. An agreement has been made with them, where they

[4]Best practice of user acceptance testing: https://www.xenonstack.com/insights/what-is-user-acceptance-testing/

occasionally test our application so we can get some feedback to see if something should be changed, added or removed.

2.8.2 White Box and Black Box.

Black Box

Black Box testing, also known as behavioral testing, is a way to test the functionalities[29]. Black Box testing, can to some extend, be seen as a type of user acceptance testing (UAT), since it shares the same principles.
It is treated like a black box, where you can only see the "surface" of the system. Therefore, the tester does not need to know about any of the implementation.

Figure 2.8: The focus is on the external testing of the system

Black Box testing can be of any software system with the focus entirely on the software requirements and specifications[30].
There exist different types of testing:

- **Functional Testing** - This type is related to the functional requirements. This is usually done by software testers.
- **Non-Functional Testing** - This type is is to test the non-functional requirements. This can be the usability, performance etc.
- **Regression Testing** - This is done after code fixes, maintenance etc. to check if it has affected the existing code.[31]

We will be using the Functional Testing for the project to see if we have fulfilled the functional requirements. The testing is mostly done by software testers, but in our case we will again use another group from our study.

White Box

Unlike Black Box, White Box testing is the testing of a software's internal structure, code and design. The White Box is "transparent", meaning the code is visible to the tester. Here the non-functional requirements can be tested, with the primary focus on verifying the flow of inputs and outputs through the application.[32]

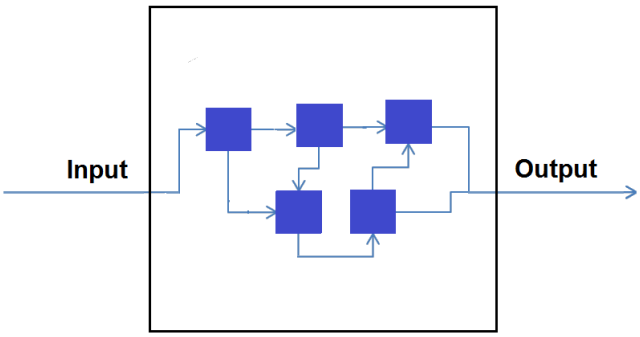


Figure 2.9: Visualization of the process for White box testing

The steps of performing a White Box testing are:

1. **Understanding the source code:** The tester will need to have some knowledge in the programming language used and understand the code implemented in the system.

2. **Test cases and execution:** The code must be tested for proper flow and structure. This is often done by a developer and includes error testing and developing small tests for each process in the application.[33]

White Box testing can be applied to most types of testing, but is mostly used with unit testing, since the method is to check if a particular path/unit of the code is working as intended.
The main advantages of White Box testing is optimization of the code by finding errors and it is more thorough since all of the code paths are usually covered.[34]

2.8.3 Monkey testing

The Monkey Testing is a program meant for verifying how an application reacts to randomly generated actions that can be performed on an emulator or phone by users, such as typing texts, different gestures, clicks and so on.[35]

We are using this test for our software, because it is an easy way of discovering all sorts of exceptions produced by mistakes in the code, which otherwise could remain unnoticed.

2.8.4 Unit Testing

Unit testing is a level of software testing, where different components are tested. Those components are called units, and the units should be the smallest testable part of any software.

Creating unit test for our code ensure that the functionality of the code haven't changed on the release of a new iteration. After having developed a new iteration, the unit testing are run before merging the code into the branch. If any of the unit test fails during this process, some of the new implementation has interfered on the behavior of one of the software's unit. This allow programmers to identify bugs in early stages.[36]

2.9 GDPR

The General Data Protection Regulation (GDPR) is a regulation in EU law on data protection and privacy for all individuals within the European Union (EU) and the European Economic Area (EEA). It also addresses the export of personal data outside the EU and EEA areas. (Data Protection Directive)[37]

The GDPR regulaion is important for our application as we will be working with a lot of personal data to be able to provide to our users the perfect match they are looking for. We will need the consent of the user with the GDPR for their account to be created.

Chapter 3

State of Art

The idea behind WeRoom that we have come up with in order to solve the problem stated in the Problem Formulation consists in an application for room renting, where tenants and landlords can find the desired room to rent and respectively, a tenant to live in the room. Why we chose to exclude other types of accommodation was mainly because of our personal experience of being students, trying to find a room to rent in a foreign country, which has not been easy, as described in the Introduction chapter. There are already a multitude of websites and applications of this kind, so we decided to differentiate our app from others by adding some well-known features inspired from Tinder, namely swiping and matching. Together with the accommodation purpose, we believe we could make the renting process more efficient and fun.

WeRoom allows users to make tenants and landlords profiles for themselves, following to add various filters and information that they require from the room/tenant they are looking for. By having this filtering option, the tenants can only see the rooms that match their preferences and vice versa. And so, we avoid cases where landlords receive messages from tenants that do not meet their requirements.

The next step is swiping, left for disliking a room/tenant and right for liking it. In the case where a tenant and a landlord like each other, a match is being made between them. Also, a chat between the two is being created.

In this chapter we have included all applications that influence our project through features and design. They are separated by the two main ideas we had, as follows: accommodation and communication/networking applications and websites.

Because the applications we chose to discuss have had such success over the years, we believe that we could draw some inspiration from how they chose to respond to the problems they are trying to solve, as well as defining some functional requirements.

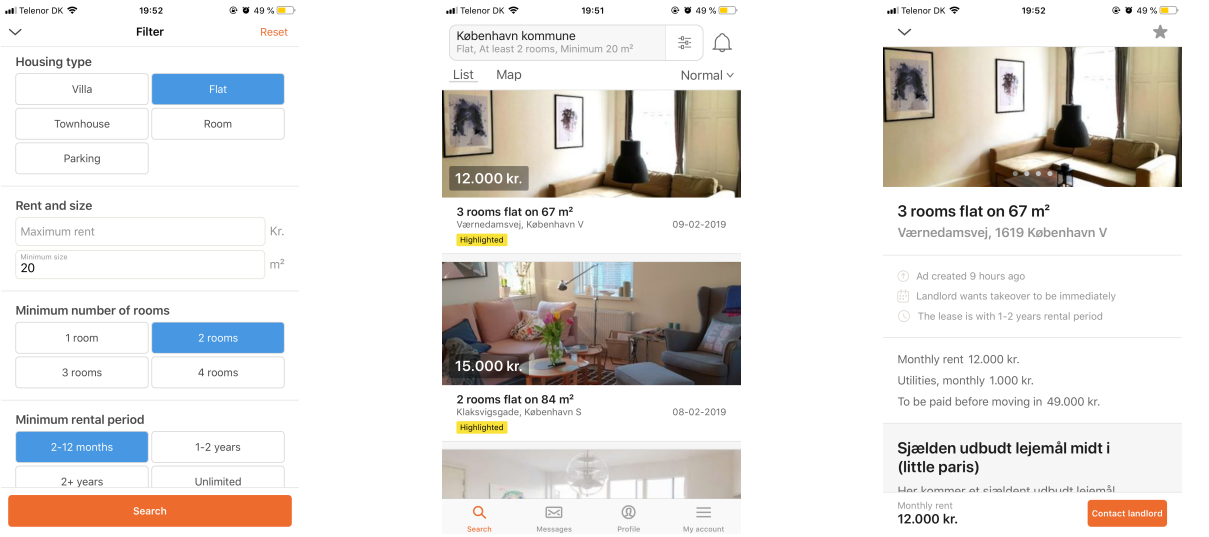
3.1 Real Estate

For our application to try solve the accommodation problem, we have studied a variety of Danish websites and applications for rentals, and the ones that caught our eyes were Findroommate.dk and Bolig Portal, because we share their accommodation concept and because they are among the most popular Danish websites of this kind[38]. In addition to this, the design for our login page is inspired from the international accommodation app Airbnb.

3.1.1 BoligPortal

BoligPortal is a Danish company focused on rentals, that has been founded in 1999. Over the years, their website has become the largest rental site in Denmark, following in 2017 to develop an application for both iOS and Android users.[39]

This application provides a good way for people who search for a place to rent for both short and long periods of time.



(a) Filters (b) Places for rent (c) Room details

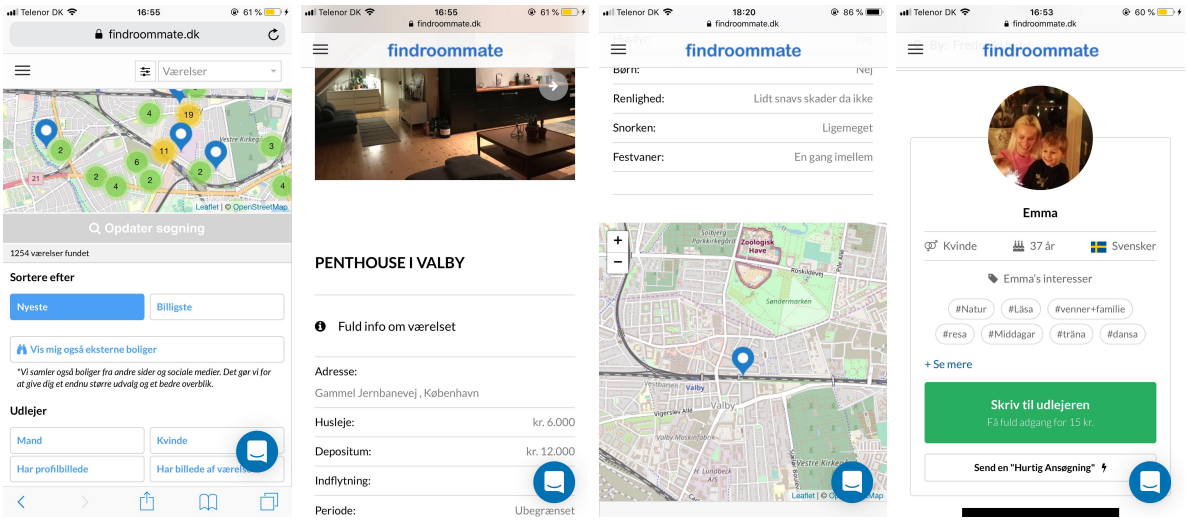
Figure 3.1: Screenshots from Bolig Portal app

The application offers the possibility of an account creation, communication between landlords and tenants, map location for the properties and so on. With the help of filters (see figure 3.1a), the user can choose to see only the places for rent that match their wishes. This feature is very effective and can be seen in various applications and websites of this kind.

3.1.2 Findroommate.dk

Findroommate.dk is a company that has been founded in 2008 and that has one of the most popular rental sites in Denmark.[40] This website is perfect for people looking to share a room from an apartment or house with other people.

As previously mentioned about BoligPortal, Findroommate.dk also has features referring to account creation for both landlords and tenants, filtering, locations on maps, communication between users and so on. What mainly differentiates the two websites is that BoligPortal is focusing on people who look generally for a place to rent, be it a house, an apartment, a villa or just a room, whereas Findroommate.dk is only for people wanting to share a room, as the name suggests.

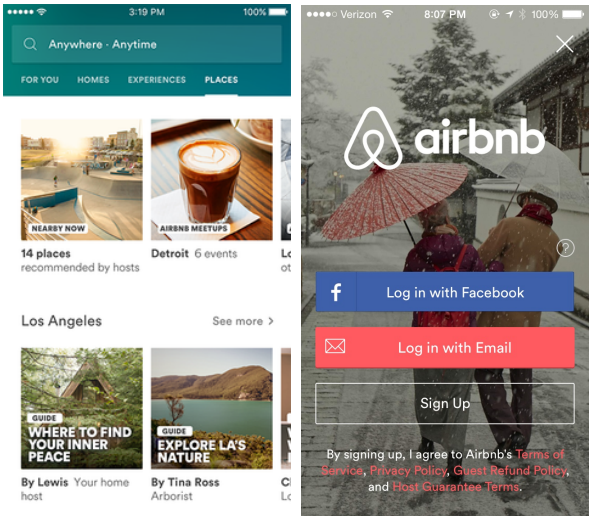


(a) Filters (b) Room details (c) Map location (d) Tags

Figure 3.2: Screenshots from Findroommate.dk

3.1.3 Airbnb

Airbnb is a company for international shared accommodation, founded in 2008 and headquartered in San Francisco. They are accessible through their website and also through moblie apps, both for iOS and Android.[41] Airbnb has become a very popular way of finding a cheaper shared place to rent for a short period of time, for tourists and people who want to rent their rooms to them.



(a) Main page (b) Login page

Figure 3.3: Airbnb

Common attributes

Both BoligPortal and Findroommate.dk have a straightforward use. The login page offers the possibility of accessing the account using the smart login. This feature allows the user to pick an account from an

external application to login, such as Facebook. The next page displays the search bar for typing the desired location, there is a button for filters that has the well-known symbol offered by material design[42], then there is a list of places for rent that match the chosen filters, which can be scrolled down by the user. When clicked, the details of the place appear and at the bottom part, the user can see where the apartment or house is located on a map. In the right-down corner, the user can see a button for messaging the landlord.

Inspired features and design ideas

We decided to take the filtering feature to a more advanced level, by adding many more filters than the rest of the renting apps have, for the purpose of having them match users. We believe that by doing this, we can optimize room/tenant searching, by avoiding cases where the landlords have numerous messages from tenants who would not match his/her requirements and where tenants see rooms which might look appealing at first, but are not actually a valid option because of different attributes they might have/lack.

As mentioned in the common attributes, after clicking on a property, a page of information opens, which has a nice and organized way of displaying the details of the room (see figures: 3.1c and 3.2b). We thought this type of structured layout would fit our information view of the profiles, which can be seen by the users after tapping on a profile from the swiping page.

Findroommate.dk is a very good inspiration source for us, firstly because we also had the idea of having rooms as the only rental option. Moreover, this website has various features we want to implement, such as the map viewing of the properties' location (see figure: 3.2c), which in our app would appear when a landlord creates a room profile. The reason for implementing this is because it is both a security check for the app to make sure the location exists and a visual way for the user to know if the address written is the right one. The description of users using tags is another feature we want to implement, because it is a faster and nicer way of informing other users about yourself in just a few suggestive words (see figure: 3.2d).

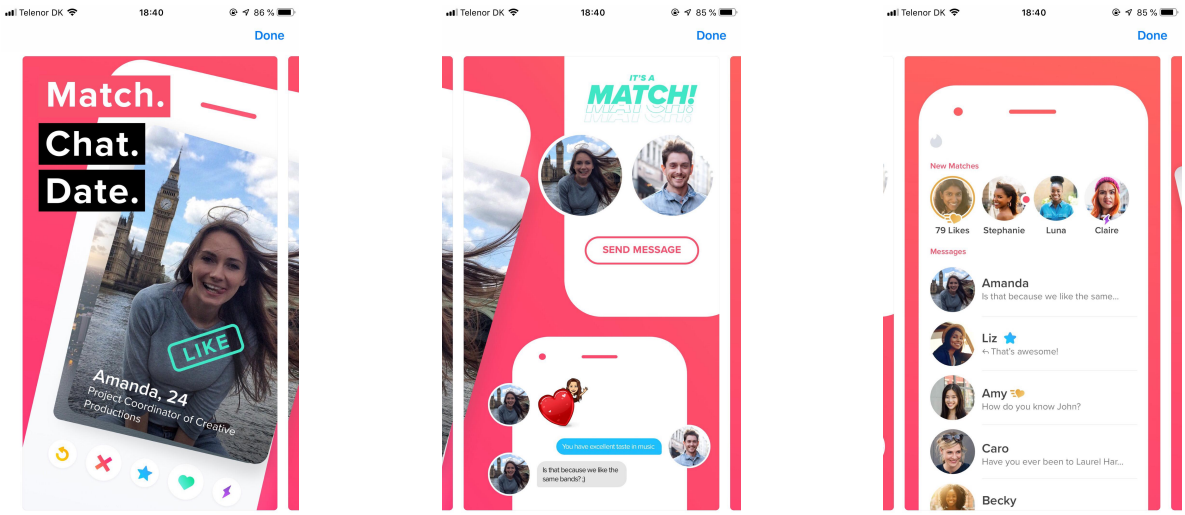
Furthermore, for the design, we drew inspiration for the design of our sign up and login page from Airbnb and Bolig Portal, because both of them have the two mentioned pages with a picture in the background and a few other buttons and fields to be filled in. We decided to follow this design, because it emphasizes the idea of finding a room to rent, ever since the sign up page. As can be seen in figure 3.3b, Airbnb's login page has a connection made between the colour of the umbrella and the email button. This is making the relation between the background and the buttons not so rigid to the eye. This is what we liked most and what led us into having sign up and login pages similar to this one, even by choosing this exact colour as or main theme colour.

3.2 Communication/Connectivity

We made research on world recognized apps with good UI and we especially liked the swiping feature and the matching concept of Tinder. Why we chose these exact features is because they fit our idea of trying to make the process of finding a room/tenant faster and more pleasurable. The swiping feature has been included in many other applications, because it arouses curiosity and it is a fun and quick way of viewing profiles[43]. This is the reason why we, together with the rest of the alike applications, chose these particularities. And so, the connectivity between users will be made through the match system, whereas the communication is covered by allowing the matched users to chat with each other.

3.2.1 Tinder

Tinder is a dating app, with matching primarily by location, released first in 2012. It has gained its popularity fast, first in different college campuses in USA, to come to the point of a worldwide known brand name. The user swipes left or right depending if he/she likes or dislikes other users. If there is a match, the two users can start a conversation. As a social app with that much recognition, psychologists and professors in the field of humanistic sciences have works concerning its influence in nowadays life such as the "Tinder effect"[44].



(a) Swipe (b) Match (c) Chat

Figure 3.4: Screenshots from Tinder on App Store

Moreover, Tinder concept of swiping cards is highly recognized as well. There are already many existing apps based on the "swiping function". Those "Tinder inspired apps" are not only about dating, but they have been introduced to diverse niches of life, which confirms that the model of Tinder is working sufficiently in other spheres as well. Because of that we decided to implement this feature in our app to match rooms with tenants.

3.2.2 MeeW

MeeW is one of those Tinder inspired apps, using the swipe function for the job market. American Medium described the platform as: "An innovative way to find global competencies and talents and a simple way to spread your skills and find job opportunities at your convenience"[45].

MeeW is combining four platforms in one single app: Jobs, Freelance, People and Social. The job seekers can search for jobs by swiping left or right for applying or rejecting a job advertisement, as well as listing their capabilities for the freelance market. The recruiters, on the other side, can post job ads and target thousands of applicants.

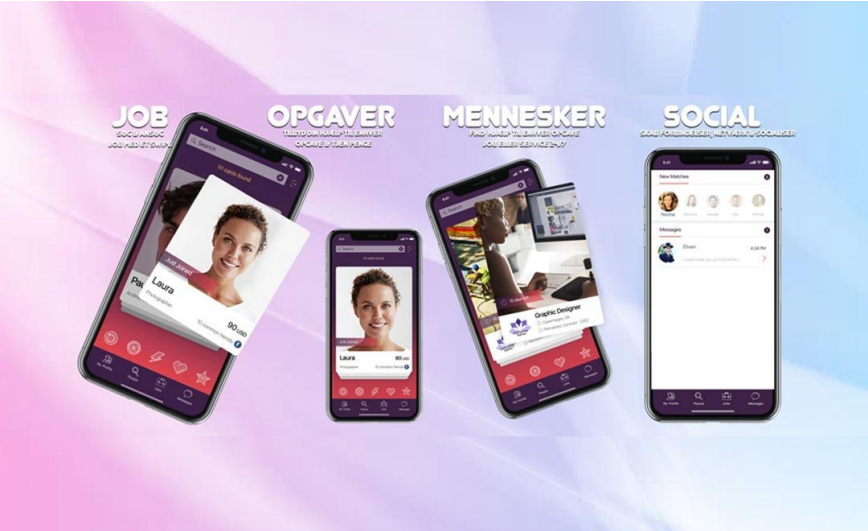


Figure 3.5: MeeW app

The app is targeting people by using tags and then doing the search by those tags. This is another concept we would like to adopt in our application. By doing this, it is possible to reach only the people interested in the job.

When you apply for a job through the app and your application is approved by the recruiter on the other side, then it is possible to start a conversation in the chat. The same process is applied to the freelance market. So, to start a conversation there should be an approval from both sides, same as in Tinder. In our app we plan to use that approach, so we minimize the received messages to a landlord to the minimum possible, and only the right targeted people can see a specific card.

Common attributes

Between the mentioned applications, there are the concepts of swiping and matching, that have been explained so far, and also the chat option that is available only after matching. These are the main features they have in common. Moreover, there are some design similarities, like the UI of the swiping and chat pages, and the motion of the cards that can be swiped. For the swiping page, the user sees the cards with the profiles of the users and some buttons for liking/disliking, in case they do not want to swipe. Below these, there is the navigation bar which leads the user to different destinations, such as their own profile or the chat page. On the same note, the chat pages on these apps look alike, because they both display in the top of the page a list of users they have matched with, showing their profile picture and the name. Below that, there is another list of users with whom they have chatted, having the basic layout of a chat. The conversations are chronologically organized and each of their layouts keeps the view of a usual conversation that can be seen in many other chat applications, such as Messenger, Whatsapp etc. The sent and received messages appear in bubbles and the bottom of the conversation page has the an edit text view where the user writes the messages.

Inspired features and design ideas

Since we liked the concept of swiping and matching, we want to implement them in our application, just like all other Tinder-inspired apps. We took the idea of swiping right for liking the profile and left for disliking it. As we also saw in the other apps of this kind, they provide buttons in the bottom part for liking and disliking, which we will include as well. The chat function and design will also be similar to the ones in Tinder. In order for two users to match and afterwards chat, they have to swipe right to each other, just like in Tinder. In addition to this, we also brainstormed the idea of a user reviewing another user, only if the two have matched in the past and also messaged each other at least once.

For the design, WeRoom keeps the basic layout characteristics for the swiping page and the chat, with all the common features mentioned in the Common Attributes paragraph.

3.3 Conclusion

All the research made on these applications has been a good source of inspiration for defining functional requirements. The functional requirements WeRoom will adopt are as follows:

The first requirements are inspired from the real estate applications. A feature we have discussed is having a map that shows addresses. Together with the map there will be a search bar for typing locations, which will give suggestions according to the typed address.

- The system can show the address of a room on a map.
- The system can give suggestions to the addresses typed in the search bar above the map view.

The following requirements are based on the Tinder-inspired applications. WeRoom will follow the swipe and match concept, therefore, the users will be asked to choose between some series of filters, which will serve as conditions for connecting users. For this purpose, the two requirements below go hand in hand.

- The landlord/tenant is able to choose the filters that describe best the place to rent and the ideal tenant.
- The landlord/tenant is able to see only the tenants/rental properties that match his/her chosen filters.

The next requirements refer to the swiping feature. The users are able to swipe left or right to a profile, left for dislike and right for like. They also have the possibility to reverse their swipe once, on the spot.

- The landlord/tenant is able to swipe either left or right, depending on if he/she dislikes or respectively likes the tenant/property.
- The landlord/tenant is able reverse the swipe only once, immediately after he/she has swiped.

Last requirements regarding Tinder-like apps represent the matching, where two users swipe right to each other, allowing them to communicate through a chat.

- The tenant and landlord who swipe right to each other are able to chat.
- The system notifies the user when the user receives a message.

Chapter 4

Analysis

This chapter includes the analysis of some important factors that influence our project. The starting point is studying the market, in order to understand better which is our target group and whether our application could indeed help, based on the group's demands regarding the accommodation problem. Following to this comes the analyzing of the requirements that we gathered from the State of Art, use cases, user stories and scenarios and from the interviews we made.

4.1 Analysis on the Renting Market

As mentioned before, finding a place to rent can be difficult, especially if the chosen location is in a big city. For example, in cities like Copenhagen or Aarhus renting can be quite a challenge, because both of the them are so called 'university cities'. This means more and more people are looking for a place to live there and because of the demographic pressure, the housing supply can't keep up with the demand.[46] On the other hand, the house prices are increasing and many are forced to find a place in the nearby cities. A study made by the Municipality of Copenhagen, concludes that the current and future demographic changes is driven by national - as-well as international - migration. Copenhagen is trying to stabilize the housing prices by increasing the house supply with around 56.000 homes by 2030. [47]

An analysis on the current rental market in Copenhagen, made by Realkredit Danmark in March 2018 [48], shows that around 30 percent of the housing stock in Denmark is rental housing and about 5 percent is housing cooperatives. Whereas if we look in the Copenhagen and Frederiksberg area, it is the rental and cooperative housing that dominates the market. It represents 60 percent of the housing stock in Copenhagen and Frederiksberg. This can also be seen in figure 4.1.

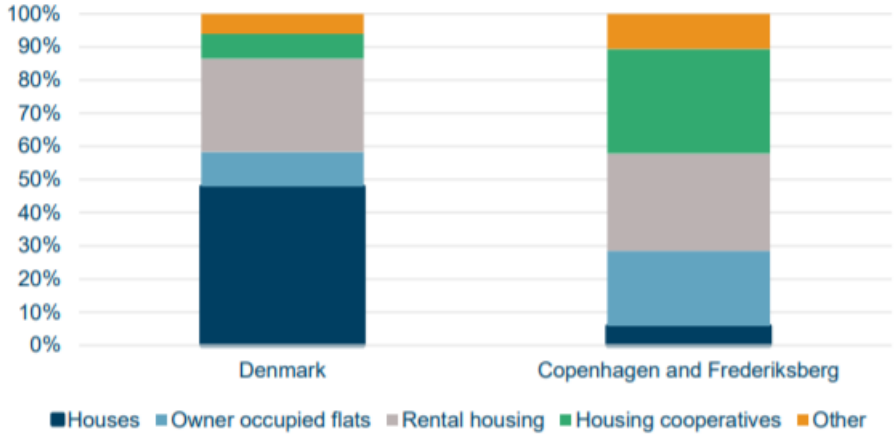


Figure 4.1: This figure shows the housing stock in Denmark and Copenhagen Area
Source: Statistics Denmark and Realkredit Danmark

But even though they are building all around Copenhagen, with the latest official Municipality Plan saying "new properties must average 95m2 (minimum), to meet the needs of families, couples and singles who wish to share accommodation"[49], it can be too expensive to rent an apartment with this size for one person.

Especially for international students it can be hard to find the cheap housing in the bigger cities and often ends up in the expensive part of the rental market. And with the long waiting list for the cheap apartments, it can be a real struggle.

For many, the solution to their housing situation can be to rent a room, while others rent out rooms in their apartment to help their economic situation. To help tenants and landlords connect with each other, there exists many different agencies to do so. Most of them requires a subscription, which can quickly be expensive if the waiting lists are long. Some use social media or other property portals, such as boligportal.dk or findroommate.dk, but it can be confusing when having to look through all these different sites.

To help with these housing problems, we create our application with the goal to establish a connection between landlords and tenants in an easy and a manageable way. With the users choosing their preferences, we find the rooms/tenants that matches these preferences, and create a chat between them, where they can sort out the rest of the details.

4.2 Market Segmentation

We already know there is a huge demand of housing in the bigger cities like Copenhagen, and now we want to find our target audience. Therefore we can make a market segmentation.

With our application, we are targeting everyone within legal age of renting. Therefore it can be hard to make a specific customer segmentation. We can then use the two-sided market business model, also known as two-sided networking, to find our target audience. The two-sided networking model, as defined by The Financial Times Lexicon, is a "meeting place for two sets of agents who interact through an intermediary or platform." [50] With this business model, we can provide benefits for both the businesses, which in our case is the landlords, and for the customers who is the tenants. According to ReasonStreet, which is a company specializing in business models, some of the benefits for the customers and businesses are:

- "Time saved to find the service brokered by the marketplace"
- "Struggle is reduced in finding service"
- "Share of transaction fee widely accepted by both sides of network when the marketplace provides value to both sides" [51]

Another rental platform, who is using the two-sided networking business model, is Airbnb since they both have the guests looking for a room to rent, and the hosts who are renting out their rooms.[52] Beside Airbnb, many other popular companies uses this business models. A few examples can be eBay, which connects buyers and sellers, Twitter and Facebook which connects advertisers with consumers.

For our application, we have the landlords who are renting out their rooms and we have the tenants who are looking for a place to live. The service we provide to the two groups is to make a connection between them and help the landlords rent out their rooms and for the tenants to find a room to rent which matches their preferences. These two segments are interdependent since our application can't function without either of them.

4.3 Requirements

In this section we will discuss and try to find out all the requirements for our application. The requirements that we set will be based on the idea we have for the application, as already described in previous chapters.

4.3.1 System Description

The system allows users to create accounts. These accounts are created using an e-mail and a password or Facebook or Google authentication.

Once the users have an account, the system requires them to create a tenant or a landlord profile. In both cases, users have to fill in information about themselves and the desired tenant/room. Landlords can have up to three room profiles. The system stores all this information in a database. This information is used by the system to filter what kind of rooms the tenants are looking for, and what kind of tenant the landlord is looking for to rent his/hers room.

When users have a complete profile, the application is opened on the swipe page each time. In this screen, users see tenants/rooms that meet the requirements they put when creating their profile. They have two options, to either like or dislike the profile they see by swiping right and respectively left.

The system matches users who swipe right to each other. Afterwards, the system sends a notification to the two users and creates a chat for them. The landlord is able to change their swipe settings, allowing any tenant who swipes right to his/her room profiles to match with them.

Users are able to delete the match with other users. In this case the system deletes the chat between the users.

The users can review others users with whom they have chatted at least once, leaving them a comment and rating them from 1 to 5 points.

4.3.2 Interviews

With the intention of discovering the requirement of the presented system we did some interviews to different tenants and landlords. The focus of the interviews was to ask them about their past experience of being a tenant or a landlord, spot the challenges that they found and how it could be improved. We followed the same guidelines for the interviews to ensure we will cover most of the topics of our application (see Interview section in Methodology chapter for the questions set). All the recorded interviews can be found in an attached file.

Once we had the guideline for our interview we tried to find 2-4 tenant and 2-4 landlords who could give us some feedback about our presented system.

Philip, tenant

Interview with a Dane, Philip, who is renting a student apartment in central Copenhagen. Philip had been spending more than half a year to find a place in the Copenhagen area and has been using various sources to find an apartment.

The interview was done in his appartment in the afternoon and gave us insight in what a tenant is looking for when searching for a place to live.

Philip mostly used findbolig.nu and Københavns Kollegiums website. He ended up calling the agency to ask if he could do something to move up the waiting lists, where he was told to meet up at some brand new apartments with the concept of 'first come, first served', until all apartments were rented out.

The most important things Philip was looking for when searching for a place to live were:

- Good location (Close to education)

- One room apartments
- Wanted to live in some sort of dorm
- Wanted to have own bathroom

He was then asked if anything was missing and what features he liked from the existing websites and explained, when looking through all the different websites, he felt he lost the overview. But apart from that, the features he liked were:

- Pictures of the places
- Could see how long the waiting lists is
- You get behind the "line" if you decline an offered apartment

Tinder and the swipe concept:
Pros: Easy to use and gives a fast look at what you are searching for.
Cons: Only see the top of the surface (Not enough details and needs more description) It gives a delusional overview.

After being introduced to our idea of the application, he was asked what he would like to see in such an application, where he told the most important preferences.
Important features in a flatmate:

- Their behavior
- Cleanliness
- Bring good energy

Minimum information about a place to rent:

- Placement of the apartment
- Rent and deposit
- Bathroom or not
- Pictures

Abel, landlord

Regarding to have some more requirement and see the reaction to our application idea we did an interview to Abel Pestonit. Abel is originally from Spain and have been living in Copenhagen for the last 6 years. He has both experience the cases, of being a tenant and a landlord in the capital of Denmark, and he is subrenting one of the rooms of the apartment that he rents.

The interview was done in his accommodation on the 3 of March, and we used the information collected to redefine the software requirements.
The interview gave us the next ideas:

- Looking for chemistry when choosing a tenant. Share same activities, hobbies or working sector. Looking for a social tenant.
- Tenants with a stable income. (Working people)

- Have been using Facebook to look for new tenants. He likes the social media. Using marketplace from Facebook. They are popular and high amount of users in both places.
- Reaction on our application:
  - Abel likes the idea on filtering the messages that you receive.
  - Was difficult to find a tenant. They have copy-pasted introduction information. The landlord spends a lot of time finding the right tenant, and a lot of time information is missing from most of the tenant.
- Filters for the application:
  - Gender.
  - If the tenant has stable income.
  - Looking for chemistry with the tenant. When introduced to the chat idea, he would prefer to have that than an e-mail. But he still wants that face to face meeting to know if they will have any chemistry.

Morten, landlord

Morten Hvitved is a Dane with different properties. He is actually renting four apartments in Copenhagen and a summer-house in the northern area of Sjælland. The apartments that Morten rents are with long term contracts. We did an interview to Morten to redefine the requirement and see the feedback that he would have with the our application idea before we started the implementation.

The interview was done in his apartment during the night. Morten gave us the next feedback regarding to our project:

- Morten uses boligportal platform to find new tenant. Has been success in the past to find it, is easy to find new tenant, and has no fees for him to list the apartment.
- He makes interview with the possible tenants to decide to which of them he would like to rent the property. Morten gets some data from the tenant and tries to decide if they are trust worthy.
  - Asking for age, if they have kids, if they have a good job/income. They want to know what is their working position.
  - Trustworthy meaning for Morten: job, age, sex, children, divorce. Trying to find stable people.
- Morten's reaction on the swipe concept: He likes the idea of saving time by filtering the information from the tenant. He likes that he can pick and filter by some criteria the tenant that could contact his property.
  - Relevant criteria: smokers, jobs, in relationship, children, pets, age, nationality, pictures.
  - He suggested to introduce reviews in our application like Uber or Airbnb.
- Morten likes the idea of having a more personal approach of searching for a new tenant. He likes the idea of been involve in the selection of the tenant, by knowing something about them before showing them the property that he is renting.
- New features: upload a picture or have a 360 degree image view for rooms.

Group Interview

This interview was done with a group of two danes, a romanian and a bulgarian. It was made at Aalborg University Copenhagen in the morning.

When asked if any of them have been a landlord before, one replied he has been renting out a room in his apartment mostly for friends, but later on, also for others while he was studying abroad. Another has been a landlord for international students studying at CBS and later on for others as-well.

How they found the tenants:
They have mostly been using DBA to find their tenants where they would get a ton of messages. They then arranged meetings with potential tenants to find the best candidate for the role as their flatmate.
When the girl were renting out for the international students at CBS, she did it through CBS Housing. All she had to do was to write her name, rent and a description in a formula and CBS Housing would do the rest and set up the contact between landlord and tenant via email.

Top things when searching for a tenant?

- Someone easy to live with and clean.
- Depends on personal situation - Single, in a relationship etc.They want someone to match their own personality and preferences.
- Important to have the personal space, but also be social sometimes.
- Good communication.
- A reliable person (Paying bills in time)

The rest of the group replied they have only been tenants, and when asked how their experience with finding a place to live, the responses were:
They were either renting an apartment through an agency or they moved in to a place with friends from previous university. One tenant found an apartment through networking and old colleagues. Some used social network, mostly Facebook, since there would be a faster response time.
They thought there were a lot of messaging and interviews, that makes it too time consuming.
Another tenant found her apartment through DBA. It provided a good overview, there were also a faster response time and it is subscription free, whereas many agencies charges a lot of money with subscription, and it is a too slow of a process to find an apartment/room.

Minimum information for a room?

- The rent
- The size of the room
- Is it in a basement, 1st floor, other?
- Location
- Who they will be living with

After a brief explanation of our application, we asked them what features they would like to see in such an application.
They liked the general idea of our application. They think it is important to have specific requirements

when searching for a room/tenant. It could be 'no smokers' or other preferences like it. Moreover, they would like a way to filter out the people who are not serious, since there is a lot of unserious people and scammers on the market.

Requirements

As understood from the interviews presented above, both landlords and tenants are mostly excited about being able to choose filters. Moreover, a feature pointed out in some of the interviews is having pictures both for rooms and for profiles. For this matter, WeRoom's system will require the users to post pictures.

- The system will ask users to post pictures both for his/her account and for rooms (in case of landlords).
- The system will allow the user to choose either to take a picture with the camera or to pick images from the gallery.

Another feature discussed in the interviews that is very common among applications nowadays is the smart login. The users can create accounts and login by using already existing accounts from applications like Facebook, Google, Twitter and so on. As it is an easy way for users to access their accounts, we have decided to implement the smart login to WeRoom, offering the users the posibility to login or create an account using Facebook/Google. An advantage of the smart login is that the system can access information about the user, like name, age, email, profile photo etc. This way, the users can skip the first step for creating a profile.

- The user is able to create an account using a Facebook/Google account or an Email address and a password.
- The system auto-completes the name, age and profile picture of the users who create accounts using Facebook/Google.

4.3.3 User stories, Scenarios, Use Cases

As a part of the requirement discovery we created different user stories, scenarios and use cases that would represent the system that has been modeled. Creating user stories scenarios and use cases will help spotting new requirement that were not spotted in the first round.

4.3.3.1 User stories

User stories are short and simple description from the end user perspective. In one or two sentence it should describe the desire of how the system would function and react to the end user. User stories are used specially in agile process models and can be used as a backlog to implement the project.
The next user stories were defined to help the members of the project understanding the functionality of the system that we where designing, and further using the user stories as a backlog for the scrum sprints.

- The user can create an account using his/her Facebook/Google account or Email address and a password.
- The user can log in using his/her Facebook, Google or Email account.
- The user can delete his/her account.
- The user can choose if he is creating an account for a tenant or a landlord.

- The tenant/landlord can improve his/her account by adding some information about him/her.
- The landlord can add several rooms for rent.
- The landlord can remove the ads for the rooms he/she rents.
- The landlord can choose between the filters that describe best the place he/she is renting and the tenant(s) he/she is looking for.
- The tenant can choose between the filters that describe best the place he/she wishes to rent.
- The landlord/tenant can modify the filters of their accounts.
- The tenant can see the rental properties that match his/her chosen filters.
- The landlord can see the tenants that match his/her chosen filters.
- The tenant/landlord cannot see the rooms/flat mates that are not matching the set requirements.
- The tenant can swipe either left or right, depending on if he/she dislikes or respectively likes the property.
- The landlord can swipe either left or right, depending on if he/she dislikes or respectively likes the tenant.
- After swiping left a card the card won't appear again in the 'search page'.
- The landlord/tenant can reverse the swipe immediately, if they swiped wrong.
- The tenant and landlord who swipe right to each other can afterwards see their contact details and also chat.

4.3.3.2 Scenarios

Scenarios bring life-based example of how a user interacts with a given system. The scenarios should not be abstract and should describe different interaction within the system where the process is successful or where the system have failed. Creating scenarios will add more details to the first round created requirements.[53]
In order to expand our requirements and give more context to them we created the next scenarios:

- **Scenario 1:**

  Gunnar owns an apartment in Copenhagen and he wishes to rent one of the rooms to share the expenses of the apartment. He had some experience finding tenants and he finds it tedious to get more than 80 message from people who are interested in the room he is renting.

  He found out that the WeRoom application can help landlords to find the right tenant. He downloads the application, he creates a new profile as a landlord including the information about the room he is renting (rent price, deposit, room size, common areas...). He is also asked to fill some information on what kind of tenant he is looking for. Gunnar would like to find a male, who is between 24 to 32 years old, who does not smoke, and is not a student.

  He will be asked if he wants to be active or passive when searching for new tenant. If he decides to be a passive searcher, any tenant that fulfills Gunnar's requirements will be able to contact him, but if he decides to be an active searcher, Gunnar will have to swipe the individual tenants. The swiping

screen will show Gunnar information of possible candidates for his room, with a brief information about them. Using a swiping system, he filters all the interested candidates.

Now Gunnar can chat with the right candidates for his room instead of reading 80 messages that he would get with a traditional way. After chatting with different candidates he finds the right flat mate.

- **Scenario 2:**

  Christiano has moved to London and he is desperate to find a room in the crowded city. He finds that most of his messages, when writing to landlords, does not get answered and he is losing his time.

  A friend tells him to use WeRoom to find his new home and he gives it a try. He makes a profile with a profile picture and information about himself that is required for the application. Afterwards he fills the information what kind of room he is looking for. Christiano is not really picky, but he has a low budget. He fills as a requirement that the room needs to be between 600£ - 900£ with a maximum deposit of 1500£.

  Once his profile is set up, he starts looking at available rooms. The application shows rooms in the middle of the screen with a picture of the room and with basic information about the room such as price, location, is the room including furniture, common areas... Christiano swipes right to all those rooms that he is interested in, and swipe left the rooms that he wants to reject.

  Christiano matches with one of the requirements of a room, and he starts a chat with the landlord that gives him an opportunity to see his new room. After a few days the landlord calls him back and he offers to rent the room from the upcoming month.

- **Scenario 3:**

  Morten uses WeRoom for a week trying to find his new room. He matched with several landlords in the process, but he was not getting any notification from the application when he had a new message. When he opened the application he could see that he had new messages from landlords but when he was talking to them it was already too late... all rooms were already rented by someone else.

- **Scenario 4:**

  Mario has some trouble with his account. He created the account using his private e-mail as a reference but he does not remember his password.

  The application does not have any help button or any reset password mechanism. So Mario decides to quit the application and find a room in another way.

4.3.4 Use cases

Use cases identify single interaction from a user or other system with the designed system. Each of the cases should be documented in detail, describing the case behavior, connection with another cases, actors that are involved etc.

ID	**2**
Title	Smart registration
Description	The user is able to create an account using his/her Facebook or Google account.
Primary Actor	The user
Pre-conditions	The user has downloaded the application. The user has a Facebook or Google account.
Main Flow	1. The user opens the application. 2. The user chooses to create an account. 3. The user creates the account using one of the already existing accounts from Facebook or Google.
Post-conditions	The user can setup his/her account. The user can choose if he/she wants to make an account for a tenant/landlord.
Alternative Flow	Facebook/Google denies the access to create the account and an error message appears.
Priority	1

(a) Use Case 2

ID	**12**
Title	Create room
Description	The landlord can create one or more profiles for the rooms he/she is renting out.
Primary Actor	The landlord
Pre-conditions	The user has a landlord profile.
Main Flow	1. The landlord is asked to create a room after completing information about himself/herself and the ideal tenant. 2. The landlord fills in some details about the room. 3. The landlord is asked to add/take some pictures of the room. 4. The landlord can choose to create profiles for more rooms.
Post-conditions	The profile of the room/rooms can be seen by other users. The landlord is directed to the swiping screen. The landlord can swipe to tenants' profiles. The landlord receives notifications regarding the room. The application creates a separated chat page for each of the rooms.
Alternative Flow	The landlord fills in wrong input and error messages appear.
Priority	1

(b) Use Case 12

Figure 4.2: Use Case examples

Use cases bring more detail than scenarios and user stories by giving them a context. In a proper documented use case, any developer should be able to detect when is the case used, who are the actors that are involved in the case, and the flow of a success cases or a failure cases.

Use cases will be helpfull when implementing the application, giving the context during the code to the developers, and also can be used for testing once the code is finished.

The use cases will follow the next structure, and will be used to create a Use case diagram in the system design.

In figures 4.2a and 4.2b there are only two of our use cases tables that display an organized list of actions that the user can perform (for the rest of the use case tables, see the annex). Another purpose they serve for our project is highlighting functional requirements that we should take into consideration.

According to our idea, we would like to have two types of users: tenants and landlords. Therefore, a set

of requirements refer to the possibility of profile creation. Landlords can have up to three room profiles that he/she use, while tenants can choose to have either a personal profile or a group profile, where there are at least two tenants looking for accommodation.

- The user can choose if he/she is creating an account for a tenant or a landlord
- The user can choose to create a group account for several tenants.

The first requirements that are inspired by the use cases are about allowing the user to edit the information about his/her profile. This is a right that users have in every application or website.

- The user is able to reset his/her password.
- The user can edit the information on his/her profile.

On the same note, the users should be able to delete their accounts. Although, if the users do not necessarily want to delete their accounts, WeRoom will allow them to deactivate their accounts or profiles and respectively, activate them. The reason for this is that, for example, if a landlord has found a tenant for his/her room, that room should not be visible to other tenants for however long the room is taken. This is where the landlord can deactivate the room and then activate it when he/she decides to.

- The user is able to delete his/her account.
- The user is able to activate/deactivate his/her account and profiles.

Moreover, the users will be allowed to review each other with rating from one to five stars and text description. The condition for this is for them to have matched in the past and to have chatted at least once. The reviews can be seen on their profile pages. The user can modify the review and the changes will be updated on the profile.

- The user is able to review another user with whom he/she had matched and chatted at least once.
- The user is able to modify the review.

4.3.5 Requirements

Requirements, as mentioned in the Methodology, are agreements on a project, made between the client and the contractor[54]. For our case, the requirements have been decided and agreed on by the members of the group, because our group has both of these roles. The requirements mentioned below are in the original form, as discussed in the beginning of the project. They are separated in the two main categories: functional and non-functional.

4.3.5.1 Functional

For the functional requirements, we tried to shape the interaction between the users and the system, according to the idea of the project. The main source for these requirements are the State of Art, the interviews, the user stories, scenarios and use cases.

- The user is able to create an account/login using a Facebook/Google account or an Email address and a password.

- The system auto-completes the name, age and profile picture of the users who create accounts using Facebook/Google.
- The user is able to reset his/her password.
- The user is able to delete his/her account.
- The user is able to activate/deactivate his/her account.
- The user can choose if he/she is creating a tenant/landlord profile.
- The landlord can create profiles for up to three rooms.
- The system can show the address of a room on a map.
- The system can give suggestions to the addresses typed in the search bar above the map view.
- The landlord/tenant is able to choose the specifications(filters) that describe best the place to rent or the ideal tenant.
- The system will require users to post pictures both for his/her account and for rooms (in case of landlords).
- The system will allow the user to choose either to take a picture with the camera or to pick images from the gallery.
- The user can edit the information in his/her profile.
- The user can activate/deactivate his/her tenant or room profile(s).
- The landlord/tenant is able to see only the tenants/rental properties that meet his/her requirements.
- The landlord/tenant is able to swipe either left or right, depending on if he/she dislikes or respectively likes the tenant/property.
- The landlord can choose to set his/her profile to no swipes, so that he/she would match with any tenant who swipes right.
- The tenant and landlord who swipe right to each other are able to chat.
- the system notifies the user when he/she matches with another user.
- The system notifies the user when the user receives a message.
- The landlord/tenant is able reverse the swipe only once, immediately after he/she has swiped.
- The user is able to review another user with whom he/she had matched and chatted at least once.
- The user is able to modify the review.
- The system notifies the user if more swipes are available.

4.3.5.2 Non-functional

The non-functional requirements present the features of the system that define its architecture[55], rather than its behaviour.

- The system has a database to store information.
- Log-in connection between app and database is encrypted.
- The system uses Facebook and Gmail API for account creation.
- The device has at least Android 4.4 OS version (SDK 19).
- The system meets all GDPR requirements.
- The user will not wait more than 5 seconds to login.
- The system is able to access camera to take pictures.
- Map showing the address should be disabled if the device's battery is lower than 10%.
- The application re-sizes pictures or accepts only pictures lower size than 8 MB.
- The application auto-fills and verifies addresses when typed into the search bar.

4.4 Conclusion

The evaluation made on the factors mentioned above has an important role in our project. The reason for this affirmation is that there is a need of a thorough understanding on the market, because the potential users and the competition are the ones who make demands and respectively set standards that we have to follow. For our case, the accommodation problem is based on the fast growth of renting demand that people make, whereas the solution we are trying to create is influenced by how other applications already do it.

Another reason is that all sources we can make use of, are a part of the process of gathering requirements, which serve as guidelines on how should we design the system.

Chapter 5

System Design

This chapter presents the process of shaping the system and the methods used for this purpose. More specific, there are discussions on the architectural design of the system and what are the application's connections with external components, discussions on which are the diagrams that dictate the behaviour of the application and also on the requirements that have been redefined after some more in-depth research.

5.1 Architecture

Architectural design gives, during the development of the code, a clear overview on how the code is going to be structured and what are the components that are involve on the application.

Ian Sommersville[56] recommend to have this process on early stage and should be a connection between software requirement and the system design. In an agile process is accepted to create an overall system architecture. Creating a incremental architectural design can lead to future re-factoring, and while code re-factoring can be feasible, re-factoring architecture can be expensive in resources.[57]

WeRoom application follows a Model-View-Control[58] behavioral design pattern. This is a part of the architectural design and describes if a component is a model, a view or a controller and how the layers are structured in the application. The first layer in the app, *view* is regarding to the interface of the application, and contains all the layouts, fragment and activities.

- **Model:** Represent objects in the application, and manage fundamental behavior and data for the application. In our application, the models will represent the profiles for the users, landlords, tenant, rooms, matches and lastly the messages from our application.
- **View:** Provides the data to the user. Will show the models to the application user with a interface that the user can understand. This classes are the one related to display the information in the screen to the user of the application, and make sure that the action performed from the user are recorded in the application.
- **Controller:** Those are the classes that make a connection between models and views. They contain the logic of the software. In our application they will deal with the connection with the databases, external API's, gallery and camera function from the camera, etc.

For small application like WeRoom the architectural design is focused on decomposing the application into smaller modules and showing the connection between them. [59]

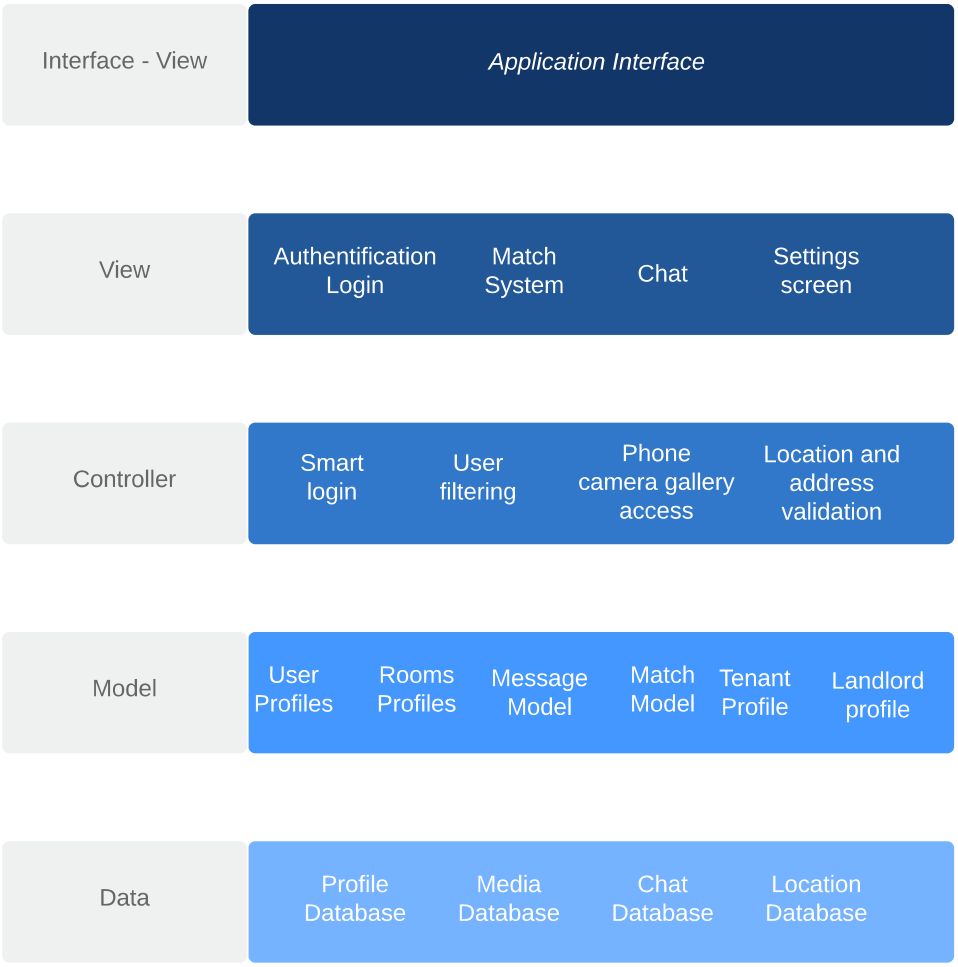


Figure 5.1: MVC model of WeRoom

After structuring the behavior of our application, and how each of the classes will communicate with each other, we started structuring how the components from the application would be related with each other. In early process, we did a data flow diagram that can be found in the annex. This helped us in the early stage to recognize how we could organize the code, and how we should create the package inside our application and spotting some of the externals libraries or API's that our application will use.

In later stages we had a logical representation of our architecture of the application. In this diagram we localize our logic between the modules of our application with the connection with all external libraries, databases, and API's.

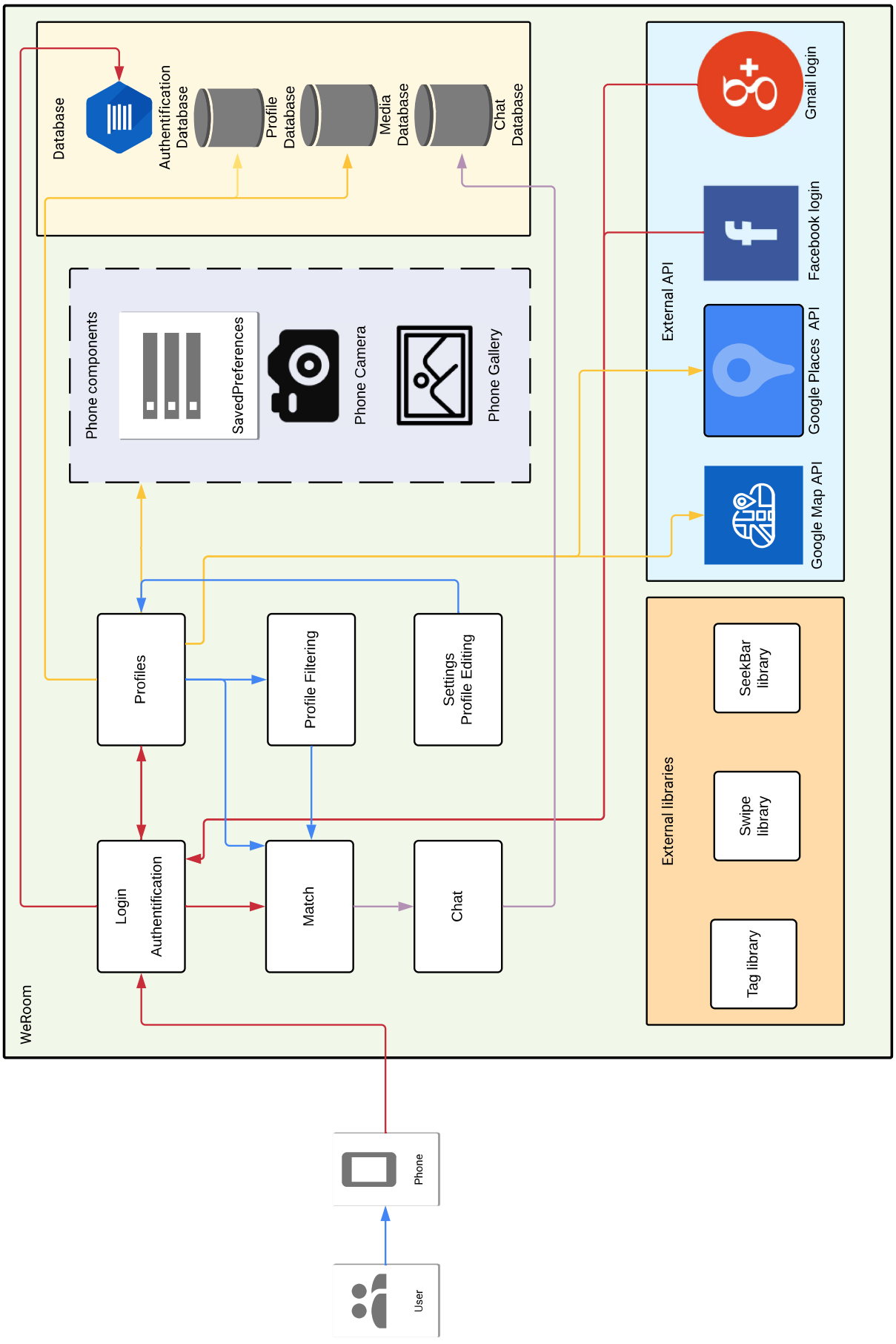


Figure 5.2: WeRoom: Architectural diagram

Figure 5.2 shows the connection between the packages that we have in WeRoom, and what dependencies they have with other external components.

- Authentication package: Will handle the process of the user to get verified into the application, the user is able to login with email and password, Facebook validation or Google validation. It will create the new profiles for the user when setting their tenant or landlord profile.
- Models package: Have all the models of our application and also contains other controllers that will help the views to store and retrieve pictures from a particular profile from the database.
- Profile filtering package: Creates the logic to filter the rooms or tenant before displaying them in the match section to the end-user.
- Swipe package: Displays the rooms or tenant to the user, and will allow them to pick which ones they would like to connect via chat. Once the a room and a tenant like each other a chat should be open between them.
- Profile views package: Allows the user to change some of the settings in the application, and allows the user to modify their profiles or change the role from landlord to tenant or vice-versa.

The diagram also identifies all the externals connections that our application have a dependencies with. All dependencies have been split in the next groups:

- Phone components: Internal function that the android phones offers to other application. We will use the camera and gallery calling another application from android and waiting for a callback when those application are done. This connection between the system and the camera/gallery applications has been implemented by using an open source code.[60]
- Database: Database is needed to store all the data that the application handles. The system need a database to let user create accounts and handle the logins. Also it will need to store information about profiles, store pictures and store all the messages from the databases.
- External API (Access Point Interface): The application will use smart login(Smart Lock[61]), and to be able to do so we will integrate API's from Facebook[62] and Google[63] to achieve this. Also to verify the data from location of the room, the application uses Google Places API, and last it uses Google Maps API to display in a map the introduced address.
- External libraries: Some external libraries from public project where embedded in the project to make the implementation of tags[64], swipe actions[65] and show a seek bar[66] in a easier way.

5.2 Modelling the system

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).[67]

As Sommerville states in his book, developing the models of a system can be done through different perspectives, such as:

1. An external perspective, where you model the context or environment of the system.

2. An interaction perspective where you model the interactions between a system and its environment or between the components of a system.

3. A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.

4. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events

In this section, we will discuss the way we have structured our system and used models to conceptualize it. After making the software architecture and having a better understanding of the system itself, we will produce the models that would be followed during the implementation part. They will be developed through different perspectives as-well, as stated above.

5.2.1 Context Diagram

A context diagram is developed trough an external perspective. We have used it after specifying the requirements in order to understand better the boundaries of our software.

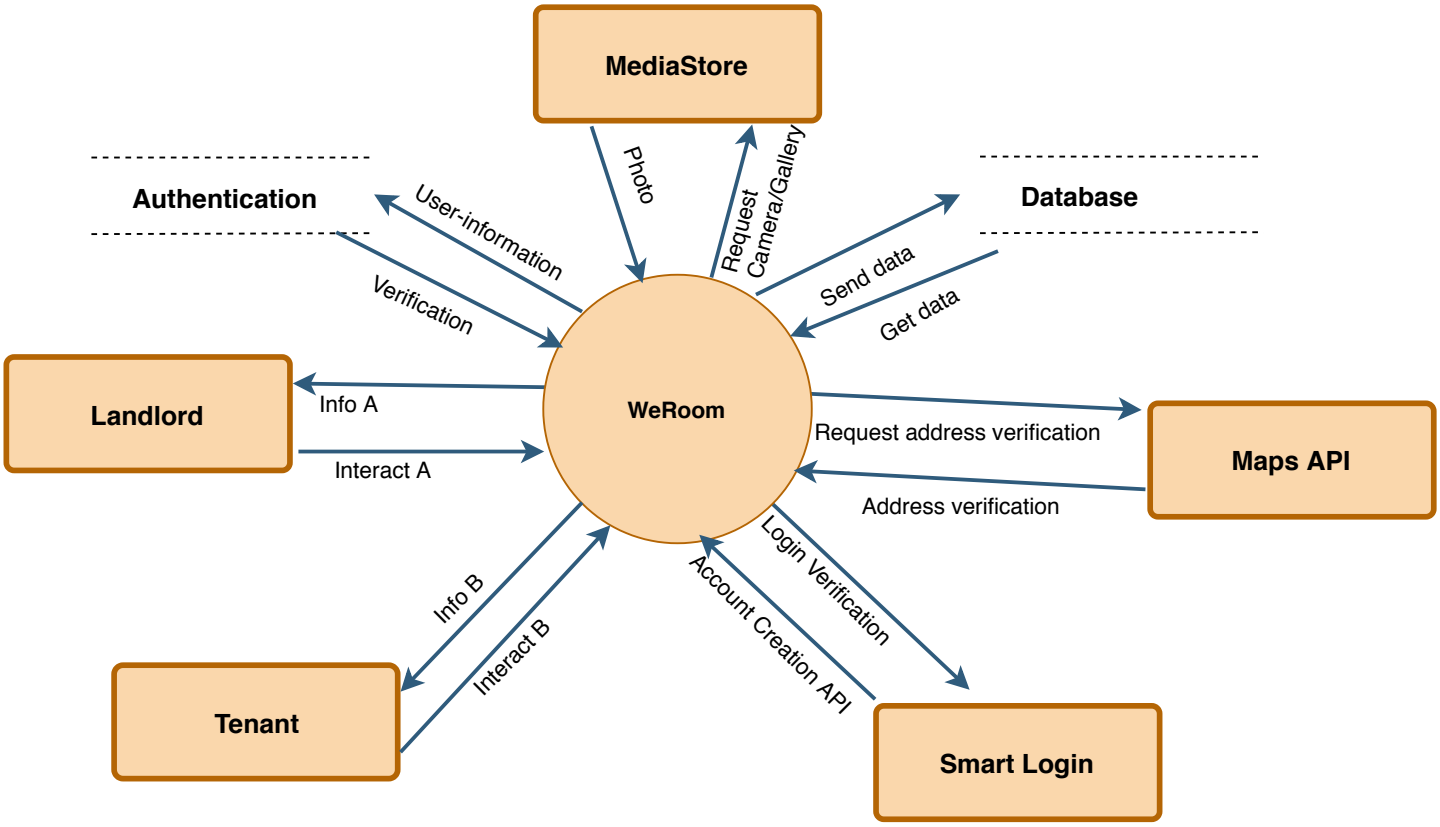


Figure 5.3: Context diagram

When we started modelling our system, we first had to make a context diagram in order to see all of the connections between our application and external terminators. This allowed us to determine clearly the dependencies on external factors.

In the following tables are described all the elements from our context diagram(Fig. 5.3)

In the table 5.4 the terminators are listed and their behavior with the system is being described. A terminator is an external object or function that is a destination or source of an interaction for the system.

Terminators	
Tenant	Tenants look for filtered rooms
Landlord	Landlords put their room up for display for tenants to see
Smart Login	Login with Google or Facebook for faster account creation
Media Store	Take picture with phone camera Choose picture from phone gallery
Maps API	Verify if the address exists Shows the address on a map
Authentication	Stores the User LogIn credentials Verifies the users LogIn credentials
Database	Stores User information

Figure 5.4: Table of terminators

In table 5.5 are listed all the interactions from the context diagram(Fig. 5.3). An interaction is the data flow between a terminator and the system.

Interactions	
Info A	Tenants get information about the different rooms or landlords - Size of room, price, location etc. -- Interests, age, name etc.
Interact A	Tenants can interact with the system - Chat, match etc.
Info B	Landlords get information about the tenants - Interests, age, name etc.
Interact B	Landlords can interact with the system - Chat, match, etc.
Login Verification	Gets information from social media for easier account creation/login - Account can be created with Google and Facebook
Account Creation API	Information when creating an account will be taken from Google and Facebook
Address verification	Verifies if the address exists
Request address verification	After adding an address for a room, it will be checked if the address exists
Get data	Receiving data from the database
Set data	Sending data to the database
Request camera/gallery	Opens the phone's camera or gallery, to take/choose pictures
Photo	Takes the picture into the user's profile
User information	The user credentials are sent in the database
Verification	Verifies if the user credentials exist and are correct

Figure 5.5: Table of Interactions

The context diagram is a use case diagram of the system from a very high level. Logically, after that we focused on the use cases for our app.

5.2.2 Use Cases Diagram

Each use case represents a discrete task that involves external interaction with a system. In its simplest form, a use case is shown as an ellipse with the actors involved in the use case represented as stick figures.[68] A use case diagram is a representation of the system through an interaction perspective.

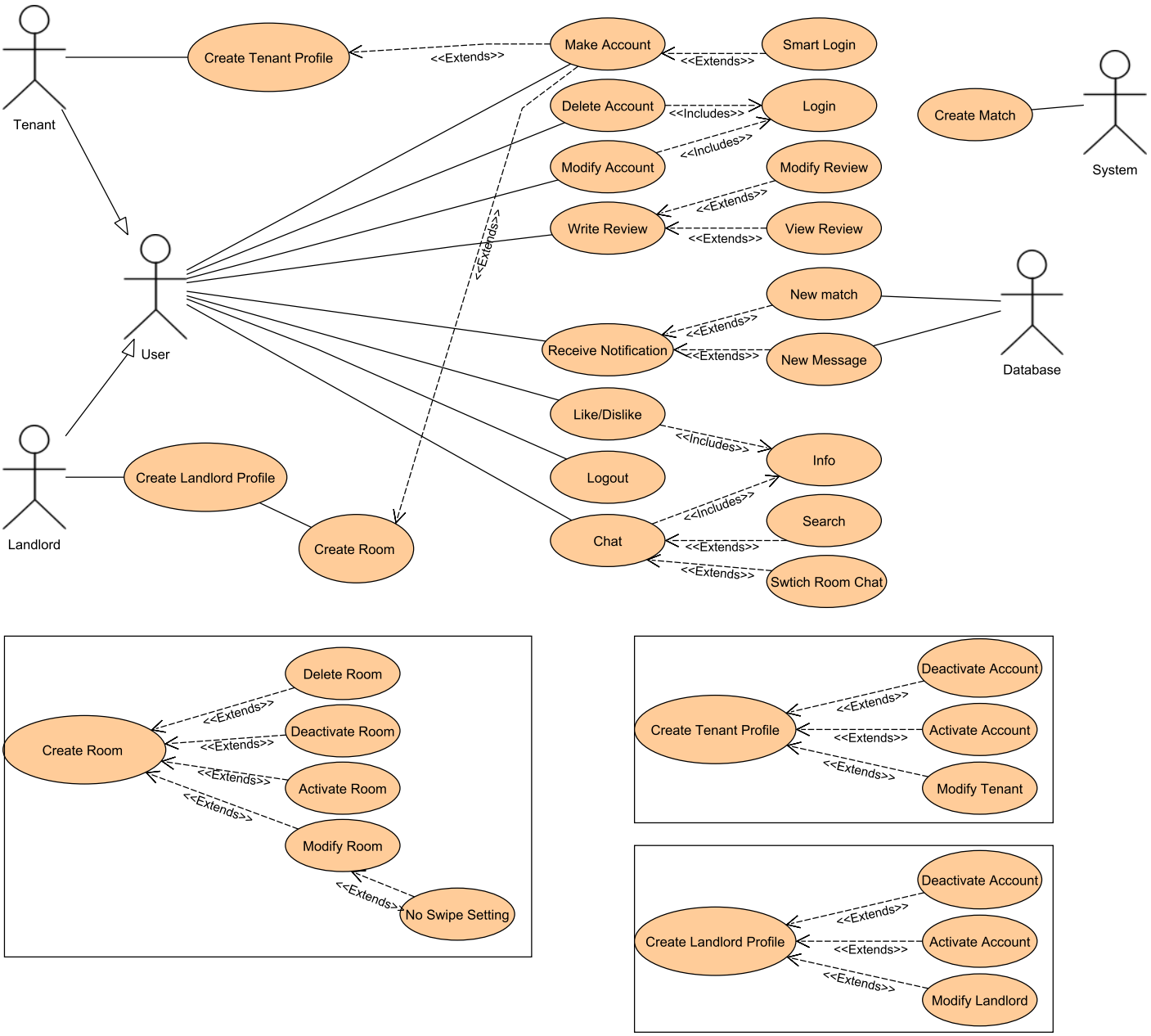


Figure 5.6: Use Case Diagram

We made a use case diagram so we can identify the user interactions with the system. Moreover, it shows us the different relationships between a user and the different use cases in which he/she is involved.

In our application we have two types of users, a Tenant and a Landlord. Those two actors share some of the use cases, but have their own as well. For creating the match, there should be likes from both sides between a room and a tenant. The match is not being created between a tenant and a landlord because a landlord can have up to three rooms and if that is the case we cannot identify which room does the tenant likes and consequently- matches.

In addition, we have put the System as an actor as well because it is the one actually creating the match. Another actor is the Database as it will be listening if there is a new match or new message. In this case there will be a notification for the user to inform him/her about that

In the following table are described all the use cases and their description.

1	Create tenant profile	The tenant can register in the system as a tenant, providing information about itself and room preferences, so the system can filter based on them
2	Create landlord profile	The landlord can register in the system as a landlord, providing information about itself and tenant preferences, so the system can filter based on them
3	Create room	The landlord can create a room and post it.
4	Make account	The user can fill in personal information as name and surname, age, gender, nationality. Additionally, we are asking them for the role they want to take in the app, either they will be a landlord or a tenant. Short personal information can be added using tags, but is not mandatory.
5	Delete account	The user can delete its account and data.
6	Modify account	The user can modify its account. Name, age, gender, nationality, role and description can be edited and changes will be saved.
7	No swipe setting	This setting will allow the Landlord to not be needed to swipe Tenants. Therefore, by activation, he/she will be swiping right (liking) all matching tenants. Subsequently a chat will be created with each tenant that swipes right to a room.
8	Write Review	A user will be able to review other users with a star system and a short description. The possibility to write a review will be given only to users that have exchanged 1 message
9	Modify Review	A user will be able to modify the review that he/she had written.
10	View Review	All users will be able to view the reviews from other users and their own review (if it exists)in the profile of a User.
11	Receive notification	A user will be able to receive notifications from the application. For example, when there has been a new match or a new message has been received.
12	Create match	The system will be creating a match between a Tenant and a Room when there is a like from each side.
13	Like/Dislike	The user will be able to swipe left or right in order to ,accordingly, dislike or like a tenant or a room.
14	Info	The user will be able to see more information of the tenant/room when clicking on a card before swiping it.
15	Chat	A chat will be created between a tenant and a room just after the system has created a match between those.
16	New message	A notification that new message has been received will be sent to the user. The database will be listening if there has been new entries(messages).
17	Search	A user will be able to search by address of room and tenant name in the chat screen.
18	Switch room chat	Each room will have it's own chat screen, because each room will have its own matches. A landlord will be able to switch between different chat screens for each room and see the specific matches for each room.
19	Smart Login	A user will be able to login with Facebook or Google account.
20	Login	A registered user will be able to login in the system with its credentials.

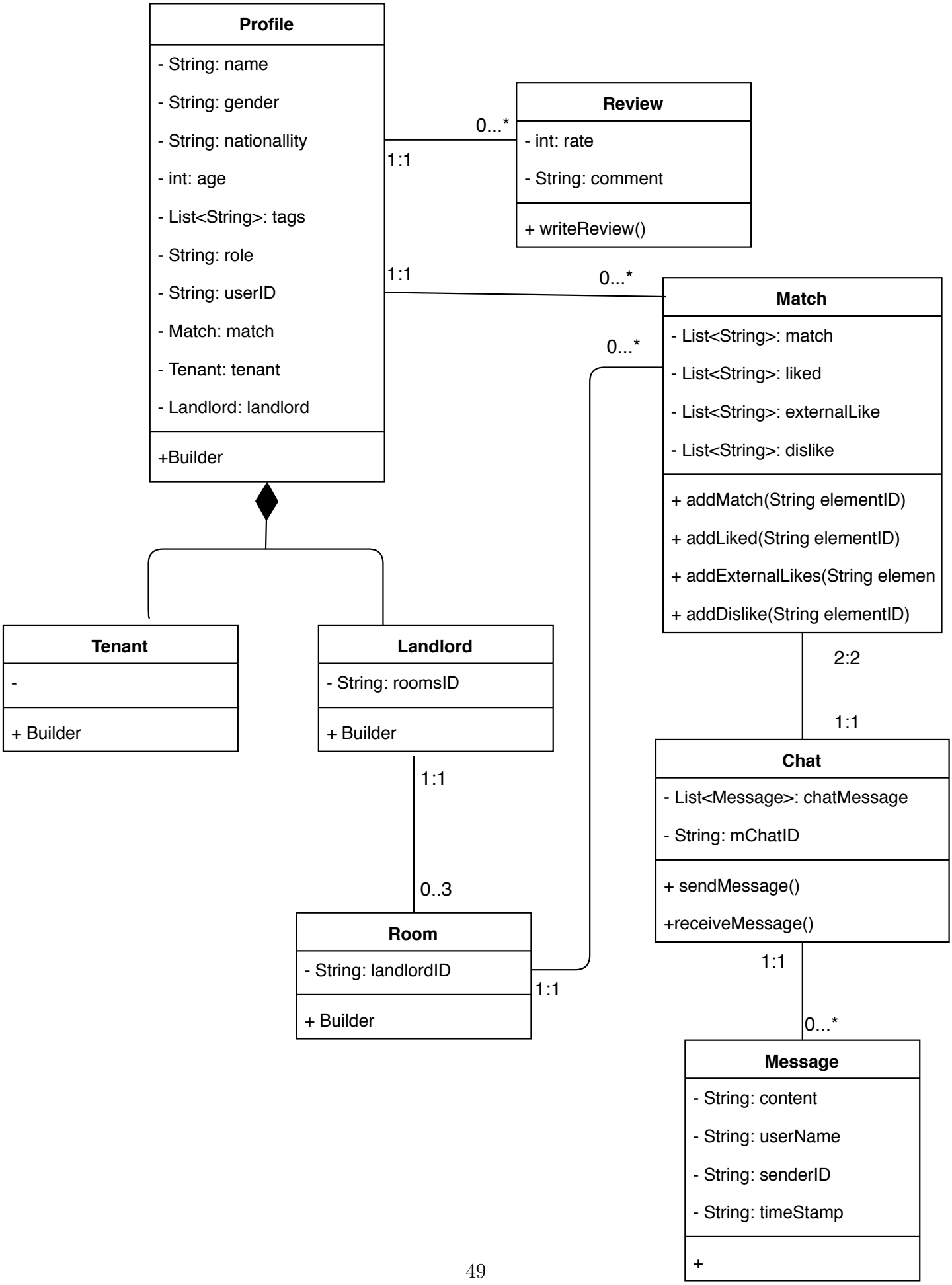
21	Logout	A logged in user will be able to logout of its profile.
22	Delete room	The landlord will be able to delete a room
23	Modify room	The landlord will be able to modify a room,change information about it
24	Activate room	The landlord will be able to activate a room, so it will be showed to possible matching tenants in the app.
25	Deactivate room	The landlord will be able to deactivate a room. The room it wont be showed to possible matching tenants. Any information about the room will be deleted.

5.2.3 Class Diagram

We have made a class diagram to have an overview of the relationships between different classes. Classes are presented in blocks with the name of the class. In the extended form they contain variables and methods of the class. The connection of those blocks shows the relation between those classes.

The following figure is our class diagram. As we are following the MVC model (Model View Controller) in the diagram we have included only the Model classes, with one exception (the class Chat is a controller).

Figure 5.7: Class diagram



As in Figure 5.7 can be seen, we haven't showed the used variables in the class Landlord and Tenant. This is like that because they are used specifically for those classes and are not shared between others. Therefore, its mentioning is unnecessary in order to obtain an overview of the relations in the system.

Moreover, in four of the classes, in the methods section, we haven't included any specific methods, but we wrote only *Builder*. *Builder* is a design pattern of implementation that we have used in our models. For those models we needed to create different representations of its objects. Creating and assembling the parts of a complex object directly within a class is inflexible. It commits the class to creating a particular representation of the complex object and makes it impossible to change the representation later, independently from (without having to change) the class.[69] For example, two instances of class Profile will have not only different values, depending on the user information, but some of them can be even missing, such as the list of tags. Implementing this without the Builder pattern can end up with robust and verbose code. By following the pattern we are able to encapsulate the creation of a complex object in a separate Builder object. So when we want to create an object of Profile class, it is not being directly created, but from that same class it was delegated to the Builder in the class.

Another specification of our systems is the fact that an object of Landlord and an object of Room keep the ID of each other. The ID is specific and unique for the object, so that way we can recognize them. Therefore, we know which instances of Room belong to which Landlord and vice versa.

As mentioned in the introduction of that section, the class Chat is not a model class but a controller. A controller class contains the logic and it's the connection between the model and the view.

An instance of our Chat class is being created as soon as there is new instance of Match. The controller is constantly listening for that. Moreover, it is listening if there has been a new message in the database.

5.2.4 Sequence Diagram

As the name implies, a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.[70]

When modelling the system of WeRoom, we have decided to make a sequence diagram to show how are the sequential steps when swiping and when chatting. The diagram was help-full for having a structured and clear idea. Therefore, later on this sequence of steps could be followed for the implementation of the use case.

The first diagram(Fig.5.8) is representing the process when a Tenant swipes right to a room.

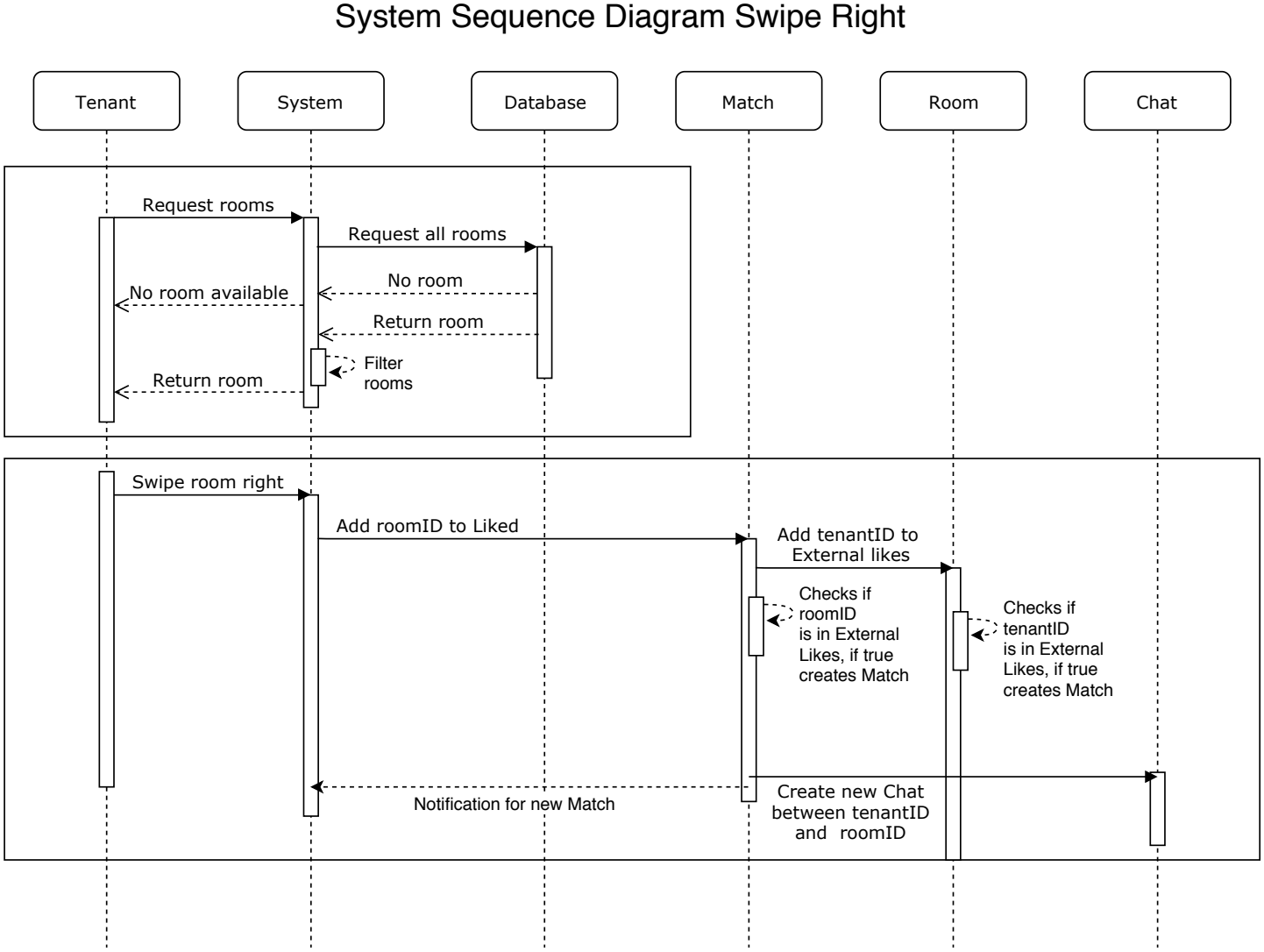


Figure 5.8: Sequence diagram - Swipe

Swiping right in our application is of means of a like, similar to another communication apps that we had reviewed in the state of art chapter.

Let's consider the table from figure 5.8. First of all, to be able to swipe anything, the user should be able to see the filtered rooms regarding to his preferences about it. The system is getting all the rooms from the database and then, if there are any, they are being filtered by the system itself. As we have mentioned in the project limitations section, using the No-SQL database would not let us filter on more than one field at a time. Therefore, at this state of the project, the application would be doing the filtering.

Consequently, when there is already something to be liked, the tenant can swipe left or right to a room. If the tenant swipes right a room, the system adds the roomID to the Liked list for the tenant. Then the tenantID is being added to to the External Likes list of the Room. If the roomID is already in the list of External Likes for the tenant a match is being created for the tenant. If the tenantID is already in the list of External Likes for the room then another match is being created for the room. As soon as there is a match for the tenant and a match for the room, the chat between the tenant and the room is being created. After that a notification is send to both the tenant and the landlord owning the room that they have a new match and they are able to chat with each other.

As it might be expected, the next system sequence diagram will be representing the process of chatting.

The second diagram (Fig, 5.9) is showing the process when a Tenant and a Landlord are sending messages to each other (chatting). Our inspiration to introduce a chat function in our app can be seen in the chapter State of Art. Some of the apps that had a chat were Tinder and MeeW.

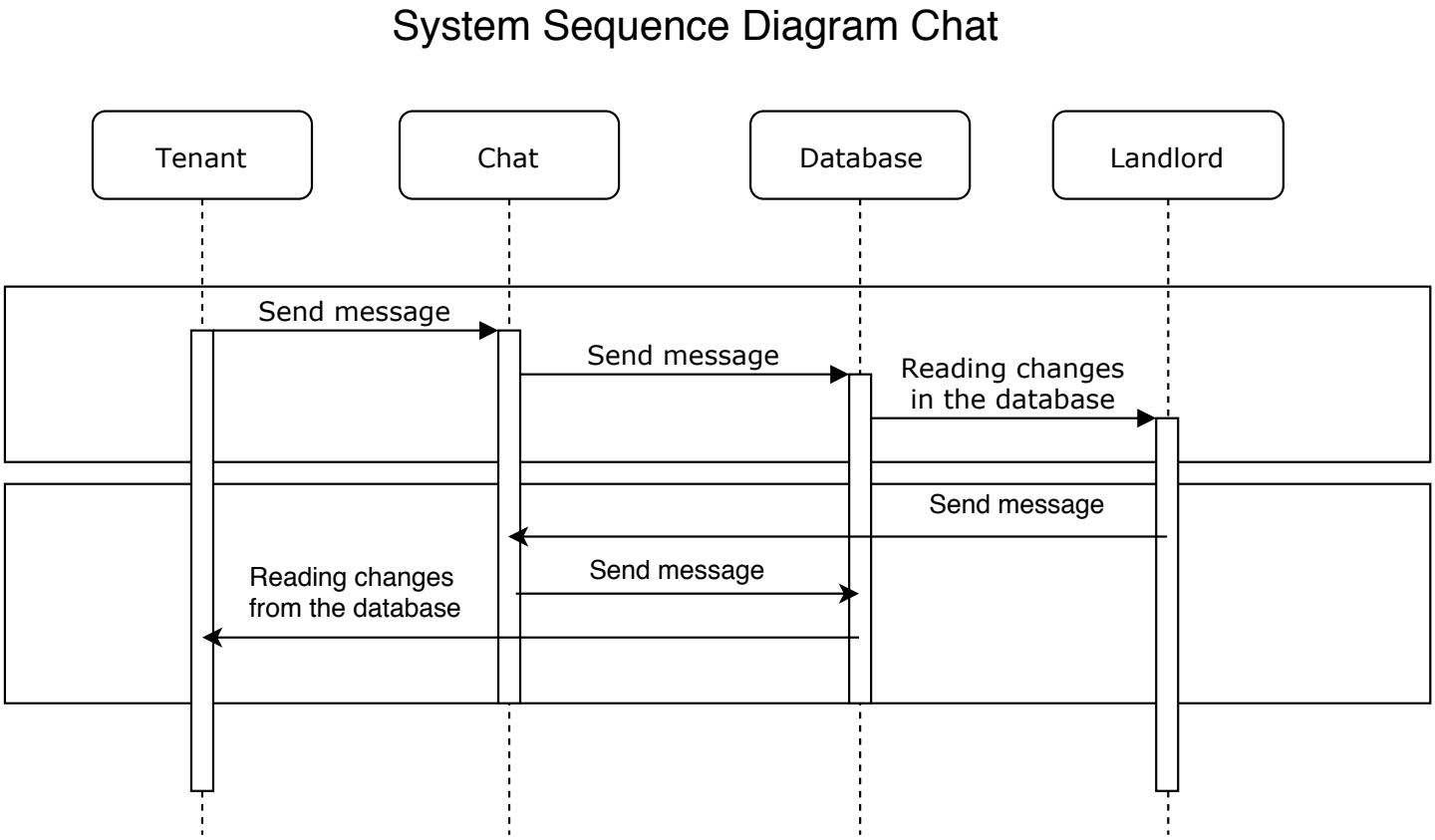


Figure 5.9: Sequence diagram - Chat

When chatting, the landlord and the tenant are exchanging messages. In the diagram can be seen every step of the sequence of that process. We have taken as an example when the tenant writes first in the chat. After writing a message in the Chat screen, the user clicks on the "send" button. His/her message is displayed in the chat screen and sent to the database. If there is any change in the database, a listener is notifying for it and the adapter for the chat is being updated so the landlord now can see the message from the tenant.

In a like manner, the same process is happening when the landlord sends a message to the tenant.

5.2.5 State Diagram

State diagrams are used to model a system's behavior in response to internal or external events.[71]

From a behavioural point of view, we have developed three state diagrams as it can be seen in Fig.5.10. For shortness the Landlord can be seen as *LL* and the tenant as *T*.

The first one from the right are the possible states of an interaction when creating a match from the side of a Landlord. From the starting point we have a choice represented in a diamond shape figure. Depending on the choice of the landlord, he/she can choose to be a Lazy Swiper. Therefore, it is a Lazy Swiper no action from him/her side would be needed to create a match. This landlord would be liking every tenant that has been filtered by his requirements. Thus, the path in state diagram for creating a match for landlord, would be depending on the guard conditions "yes" and "no". In case of a *yes* the landlord

will be waiting for tenants to like his room. If a tenant swipes right to it, then a match is created and they can chat from now on. However, if a tenant swipes it left, the state diagram comes to a direct end, no match being created and no chat.

The sequence of states if a landlord is not a Lazy Swiper is similar from the Tenant point of view. Therefore, this one will be described taking the role of a tenant.

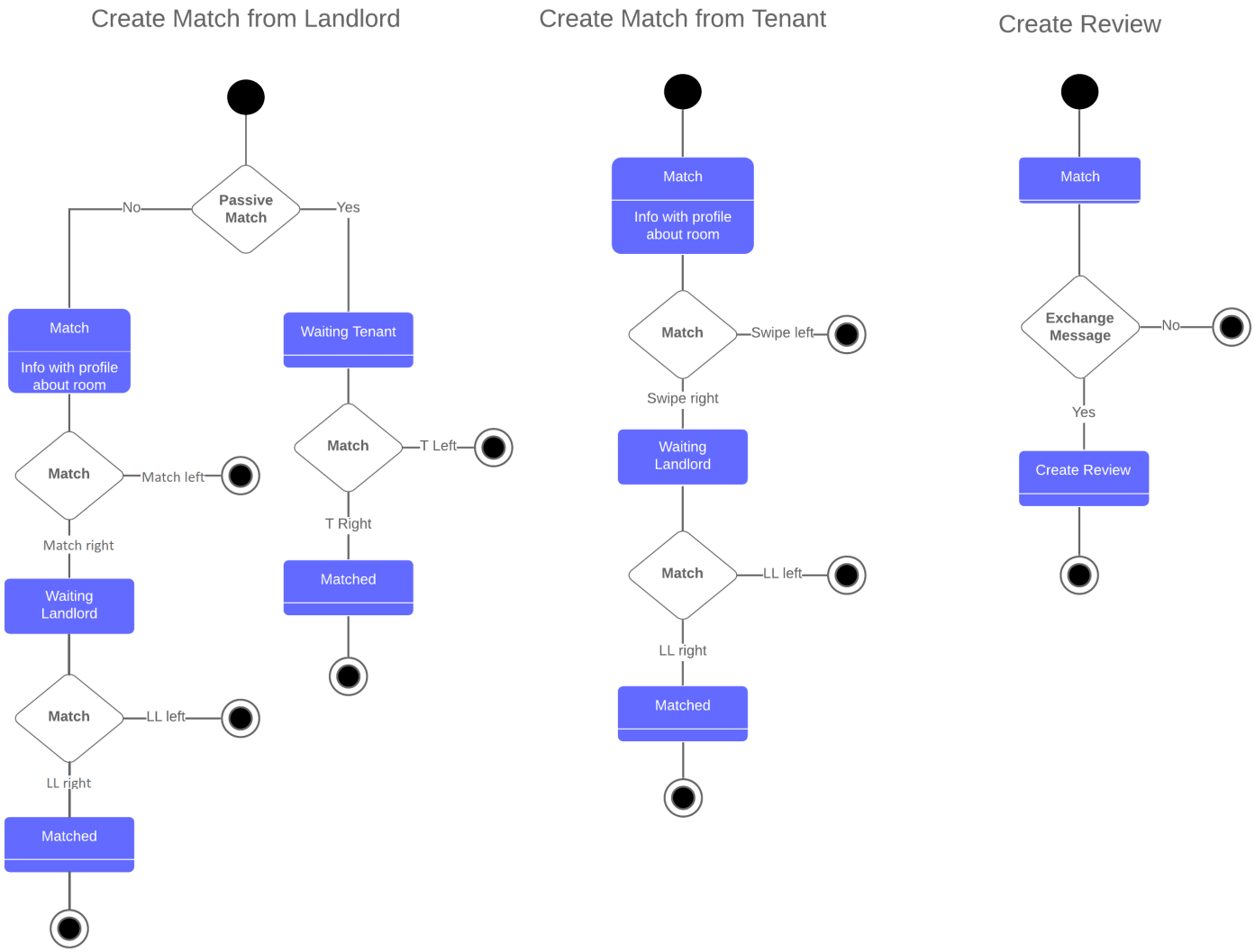


Figure 5.10: System State Diagram

The state diagram in the middle is representing the possible states when creating a match from Tenant side of the app. The tenants is seeing rooms with description and has the ability t swipe left or right to them. If swiped left, this is an end of the match creation, the roomID is being added to the Disliked list and match cannot be created.

However, if the tenant swipes right to a room, the roomID is added to the list of Liked for the tenant. Right now, the system is waiting for the Landlord to like the tenant, so the room ID is added to the list of External Likes of the tenant. If this happens, a match is being created for both of them. Analogical, same process happens from the landlord point of view, as stated above.

The last diagram from the right is showing the states of creating a review. As is already being described the process of creating a match and having a chat between two users, this will be omitted here. Following

the sequence of this diagram we get to know that a review can be created only if both users in a chat has exchanged a message each. We have decided to put this requirements for creating a review, because we wanted to make sure that there is a reasonable reason for reviewing someone.

5.3 Redefined Requirements

This sub-chapter presents the redefined requirements that have been first discovered. To prioritize them, we are using the MoSCoW analysis, whereas to organize them, we follow Sommerville's classification.

5.3.1 Moscow

MoSCoW analysis[72] is a method of organizing software requirements by importance. The capital letters in MoSCoW stand for "must", "should", "could" and "won't". As the words suggest, requirements rated as the crucial ones to have contain the verb must. The requirements that contain "should" are also important, but not necessary for the time being. The next requirements are the ones that contain the word "could", which represent desired, but unnecessary features. The last of the four types is "won't", which are considered the least important, in comparison with the others. [73]

Following this structure, we have tried to prioritize our current functional requirements by how crucial they are for WeRoom and according to the time and resources we have. We are using the MoSCoW method because it offers a logic structure of organizing requirements, so that we can have an overview of the order in which we should implement the features we agreed upon.

Must

The requirements in this category are considered of utmost importance because they shape the basic behaviour of our application. The requirements below represent account and profile creation, filtering, swiping and chatting, which are the main points covered by the idea of our project.

Should

The second category is represents important features that WeRoom could work without. These requirements represent features and actions often encountered in other applications of this kind, like resetting the password or being able to use the camera. However, if we are running short of time or other resources, we would not implement them.

Could

This category represents a series of requirements which we would like to implement, but are not actually necessary. Activating/deactivating an account or profile, as well as reverse swiping actions and notifications are particularities of our ideas, so they are not essential for the application to run. For this reason, these requirements' importance decreases compared to the ones before them.

Won't

The final category is the one that resources and especially time did not allow us to implement. These requirements would fit our ideas, therefore they represent future plans.
Functional requirements

1	The user must be able to create an account/login using a Facebook/Google account or an Email address and a password.	M
2	The user must be able to log out of his/her account.	M

3	The user must be able to choose if he/she is creating a tenant/landlord profile.	M
4	The landlord must be able to create profiles for at least one and up to three rooms.	M
5	The landlord/tenant must be able to choose the filters that describe best the ideal tenant or the place to rent.	M
6	The system must check if the information provided by the user is realistic.	M
7	The landlord/tenant must be able to see only the tenants/rental properties that meet his/her requirements.	M
8	The landlord/tenant must be able to swipe either left or right, depending on if he/she dislikes or respectively likes the tenant/property.	M
9	The users must be able to see more information about a profile, when clicking on the swipe card.	M
10	The tenant and landlord who swipe right to each other must be able to chat.	M
11	The landlord must be able to switch between chat/swipe pages of the room profiles he/she owns.	M
12	The user should be able to delete his/her account.	S
13	The user should able to reset his/her password.	S
14	The user should be able to describe himself/herself using tags.	S
15	The system should show the address of a room on a map.	S
16	The system should require users to post pictures both for his/her account and for rooms (in case of landlords).	S
17	The system should require the landlord to post at least three and up to ten pictures of the room.	S
18	The landlord should be able to choose to set his/her profile to no swipes, so that he/she would match with any tenant who swipes right.	S
19	The system should allow the user to choose either to take a picture with the camera or to pick images from the gallery.	S
20	The system should give suggestions to the addresses typed in the search bar above the map view.	S
21	The user should be able to edit the information on his/her profile.	S
22	The user could able to activate/deactivate his/her account.	C
23	the system could notify the user when he/she matches with another user.	C
24	The landlord/tenant could be able reverse the swipe only once, immediately after he/she has swiped.	C
25	The user could be able to activate/deactivate his/her tenant or room profile.	C
26	The system could notify the user when the user receives a message.	C
27	The system could notifies the user if more swipes are available.	C
28	The user could enable/disable notifications.	C
29	The system won't be able to auto-complete the name, age and profile picture of the users who create accounts using Facebook/Google.	W
30	The user won't be able to choose to create a group account for at least two tenants.	W
31	The user won't be able to review another user with whom he/she had matched and chatted with at least once.	W
32	The user won't be able to modify and create a review.	W
33	A user won't be able to search by address of room and tenant name in the chat screen.	W

5.3.2 Sommerville's categorization

According to Sommerville, the non-functional requirements can be classified by numerous criteria. The three main categories are Product, Organizational and External requirements. We chose to follow Sommerville's categorization because of its detailed structure, that makes organizing the requirements more clear and thorough.

Product requirements define the behaviour of the software through limiting its efficiency, security, usability and dependability. More specific, these requirements have the role of setting precise goals for how the system should work.

Organizational requirements refer to the actual organizing of the developer team in terms of environment, chosen operations and development methods.

External requirements represent all the requirements that come from all external actors that influence our application according to regulations, ethics and laws.

1. Product Requirements:
   (a) Usability requirements
      i. The system uses Facebook and Gmail API for account creation and log-in.
      ii. The application has a database to store information.
      iii. Log-in connection between app and database is encrypted.
      iv. The user is able to filter all tenants and rooms, up to 50km from the desired location.
      v. The user is able to share a picture in the chat.
      vi. Creating a profile does not take more than 5 minutes.
      vii. The system stores the user's settings on the phone, using shared preferences.
   (b) Efficient requirements:
      i. Performance requirements:
         A. The phone does not compute any filtering between tenant and rooms.
         B. If the user has some tenant or rooms to be displayed in the swipe process, the response time to load the first tenant or room cannot exceed 3 seconds.
         C. The system is able to disable notifications if the battery is less than 10%.
         D. The system disables the map view if the user does not have a Wi-Fi connection.
         E. The functions to filter the rooms and tenant will have complexity O(NlogN) or lower.
         F. The system runs asynchronous tasks related to queries in parallel.
      ii. Space requirements:
         A. The size of the application is not bigger than 40MB.[74]
         B. The user must not download from the database any picture bigger than 8MB.
         C. The application re-sizes pictures to avoid using too much memory loading big images (e.g.: 2560*1920 resolution).
      iii. Dependability Requirements
         A. The application must be running 24/7, and in case of a system shutdown for maintenance, the users need to be informed beforehand.
      iv. Security requirements
         A. The databases only allows registered users to make queries.
         B. No passwords are available to read for any database administrator.

2. Organizational Requirements
    (a) Environmental requirements
    (b) Operational requirements
    (c) Development requirements
        i. The development of the application will follow 'Fragment Navigation Pattern'. [75]
        ii. The application is going to be developed in Android from minimum SDK: 19 to max SDK: 28.
        iii. The application will be developed in Java.
        iv. The application's code will have Java doc for the core methods.
        v. The application's code will have no intentional warning from the IDE.
        vi. The application will use the camera and gallery if it gets the permission from the user.
        vii. The application will use 'Builder Design Pattern' to create the models.
        viii. The system will use SharedPreferences to store any application configuration.
3. External Requirements
    (a) Regulatory requirements
    (b) Ethical requirements
    (c) Legislative requirements
        i. The application will show a GDPR consent to the users before creation the account, and the application will follow the GDPR legislation.

5.4 Conclusion

To conclude, this chapter is an overview of how the system is supposed to work, what external libraries, databases and APIs have a role in achieving the application's purpose and how can a user interact with the system. All of these represent a theoretical background of the development, which will be followed when implementing the application.

Chapter 6

Implementation

Following to all the visual representations of the system's design comes the implementation chapter. Here are presented how we worked on the design of the application by following a prototype we created and how we organized the work through KanBan and Scrum. For a better overview of how the application has been implemented, there is a step by step explanation about Scrum logs, namely about how we divided the work and what challenges we met.

6.1 Technologies

In this section we will take all the non-functional requirements and will chose the suitable technologies we have to use in order to fulfill the needs of the application.

6.1.1 Firebase

Firebase is a mobile and web development application platform owned by Google. It consists of many services that can be divided into different categories such as Analytic, Development, Stability and Earn.

Authentication

Firebase Authentication provides back-end services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app. It supports authentication using passwords, phone numbers, popular identity providers like Google, Facebook and Twitter, and more.[76]

The full aspect of functionalities of our app will be accessible only to the logged in users. New users will have the possibility to register and create a personal profile, users with already existing account would be able to log in in their personal account. Therefore, we will use the service of Firebase Authentication, to handle those processes.

Storage

Firebase Storage is used by developers to store all types of media files and other user-generated content.

In WeRoom, we will provide the user the possibility to upload pictures of the rooms, profile picture and send pictures through the chat when communicating with another users.

Smart Lock: Google/Facebook

We decided to add a Smart Lock to our app, so a user can create an account easier and faster. This will be made by using the services that Firebase Authentication provides us with. Smart Lock is a function that connects the application with Facebook/Google and other's API, so that an account can be created by taking directly the information of the user profile in those platforms and returning it to our app. This is benefiting the user experience by making the process more automatically, the user doesn't need to fulfill the fields for name/ email, since this information is taken from the already existing account.

6.1.2 Database

WeRoom follows a MVC architecture as discussed in the architecture chapter in our report. The main goal of the application is managing data that is store in our database and showing it to the user in an interactive way. Then the application handles the logic between the data and the interaction of the user and store it backs in the database.

To store the information for each profile, such as name, age ,nationality, role, etc., including all the preferences for the room/tenant, we are will be using Firetore Database. This is a online cloud database provided by Firebase.

As we have a chat in our app, we have to store each string(message) that has been send from a user, and then, when there is a new message, we have to send it to the other user. To do so, we decided to use Realtime database, provided by Firebase.

Both Firestore and Realtime databses are No-SQL databses that are based on MongDB. As written in the project limitations, to have efficiently working application with this purpose, we should use SQL database to be able to query different fields at the same query, filtering the data.

6.1.3 External API

To fulfil some of the requirements, such as showing an address on a map, verification of an address, etc. we will use external API's.

Google Places API

One of the external API's we have used is the Google Places. Google places brings information about places around the world. A place can be a locale, street, address, monument etc. When connected to the API we can request information about the places like pictures, coordinates, reviews of places from google reviews... The main purpose to integrate this API is to have googles auto-complete box with places. This will create a control in the application verifying all addresses that the users types when creating a room.[77]

Google Maps API

Another one of the external API's we have used is the Google Maps. We have used it in order to implement a Map fragment so we show to the user where exactly is located the address that has been typed. In this way the user can be sure that the right address of the room has been set.

Google maps V2 that is been used in WeRoom requires to have the Play Store installed in the device. When opening WeRoom for the first time it will detect if the Google Maps is installed on Play Store, and if t is not installed in the devide it will require the user to install this software to be able to use WeRoom.[78]

6.2 Design Patterns

Design Patterns[79] are solutions to frequently encountered problems in terms of object oriented programming. Among the advantages brought by design patterns are flexibility, re-usability and shared vocabulary.

In the implementation of WeRoom we decided to follow different design patterns to keep the code simple, reusable and flexible, but at the same time without dropping out functionalities. In this section we will describe the patterns we have been following when implementing.

Fragment Navigation Pattern [80]

The Fragment Navigation Pattern changes the idea of having many activities, each having its own full-screen fragment, into creating one activity which switches between different full-screen fragments. This pattern brings a better performance and user experience.

We are using the fragment navigation pattern because of its less expensive usage, in comparison with navigation between activities and because of the easier communication between fragments.

Builder Pattern [81]

The Builder Pattern is the solution to various problems that might occur while creating complex objects.

We are using this pattern to create profiles for landlords, tenants and rooms. The reason for this is the long series of information that each of the profile contains, which has to be sent and accessed from the database.

MVC[82]

Model-View-Controller, also known as MVC, is an architectural design pattern that defines the three parts an application should follow. The controller manipulates the model that sends information to the view.

In WeRoom, the view is represented by the layout files, the controller is represented by the activities and fragments and the model is represented by classes that create profiles using the builder. The user first sees the view, then gives commands to the controller, which then sends information to the model.The received information is then passed to the database, following to be shown to other users through the view.

Adapter Pattern[83]

The Adapter Pattern has the role of connecting interfaces that normally would not.

For our application, the Adapter Pattern is used to pass the list of tenants/rooms to the swiping cards, to adapt the list of countries to spinners and for the list of messages shown in chats.

Singleton Pattern[84]

A Singletone design pattern is used to store the current profile information. This allow us to have a single point of information on the entire application regarding to this value. This will prevent not having any redundancies information in the application, and will make easier the communication between different fragments.

6.3 Material Design

Material Design[85], as previously mentioned in the Methodology, is the visual language that brings clear guidelines for layouts. Out of all the components of Material Design, we chose to implement the following ones.

Colours

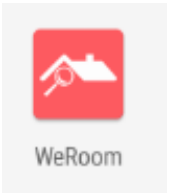
The colour systems are multitudes of colour shades that harmoniously match the chosen main colour. There usually are two palettes, the primary and secondary one. They are used to personalize and emphasize the brand. The main colour is specifically chosen, as each colour can create an emotion or energy in the user.

For WeRoom, we are using only the base colour salmon (FF5A60, see figure 6.1a) a brighter shade (FFC3C5, see figure 6.43) and white. We chose salmon as our main colour because of its optimistic

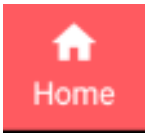
energy[86]. Other big companies that use this main colour are Airbnb, from which we have inspired, and Apple Music.

System Icons

System Icons represent basic actions and destinations. They are very suggestive and easily understandable.



(a) WeRoom's icon



(b) Home icon

Figure 6.1: Icons from WeRoom

We are using some of the icons provided by Material Design (see figure 6.1b), because of their frequent appearance in Android applications. On the same note, our application's icon has a simple design (see figure 6.1a), showing a magnifying glass and a roof above it. Through this logo, we emphasize the idea of WeRoom: the magnifying glass represents the searching, as in all other applications, and the roof represents the accommodation.

Text Fields

Text fields are editable fields that allow the user to enter text.

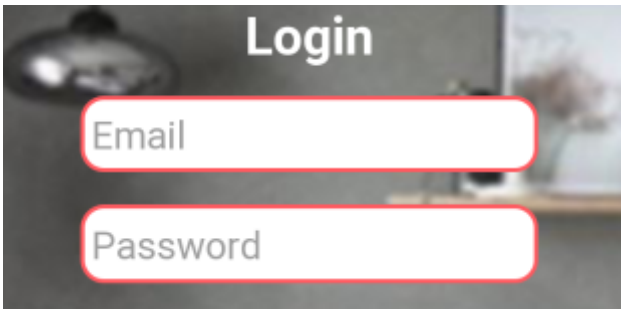


Figure 6.2: Screenshot from the login page

We are using some customized outlined text fields, that have rounded corners and salmon coloured borders (see figure 6.2).

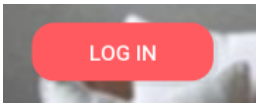
Buttons

Buttons are elements that allows the users to perform an action. There are four types of buttons according to Material Design: text buttons, outlined buttons, contained buttons and toggle buttons.

Text buttons are, as the name suggests, buttons displaying only text and optionally an icon. If the previous button is also framed by a container, it becomes either an outlined button, if the container has no colour, or a contained button if the container is coloured. The last type, namely the toggle buttons, are buttons consisting of an icon inside a container.

(a) Text Button

(b) Outlined Button



(c) Contained Button

(d) Toggle Button

Figure 6.3: Buttons from WeRoom

Our application features buttons of all types mentioned above. As can be observed, figure 6.3a represents a text button from the login page, figure 6.3b is an outlined button from a pop-up message, figure 6.3c displays a contained button, also from the login page and, respectively figure 6.3d is a toggle button from the swiping screen. The majority of them is coloured in salmon for emphasis purposes, while the others are white. The only exceptions are the buttons for liking and disliking, which are green and red, because of their roles. Green for a positive action and red for a negative one are frequently used.

Bottom navigation

The often encountered Bottom Navigation Bar allows the user to switch between the main activities of the application and is present on every screen. There are usually three to five buttons on the bar that display a representative icon and text. The purpose of this bar is to easily access specific destinations.

Following the guide from material.io[87] we find that the Navigation Bar suits perfectly the behaviour of our application, when the user can choose different destination inside of the application. Also its recommended to use a Navigation Bar only when there is 3 to 5 places to be displayed in the bar.

Cards

Cards[88] are surfaces showing brief details on one matter.

We are using cards in the swiping screen. The user is able to swipe between cards that show a possible tenant or room that matches the filters. The cards display a representative picture and a few basic information. If pressed, the cards show the user a thorough description of the tenant/room. The choreography of the cards is simply the left and right swiping motion.

Dialogs

Dialogues[89] are window that pops up, usually in the middle of the screen, in order to inform the user of something. While they are shown, any activity of the application is paused.

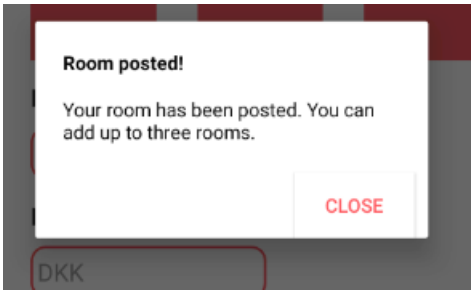


Figure 6.4: Dialog

Our application features a few dialogues, having informational purposes, an example of a dialog from WeRoom being the one from figure 6.4. They display a title, text and a closing button. The button is an outlined one and it is coloured in salmon.

Bottom Sheet

Bottom Sheets are interactive surfaces that display supplementary content.

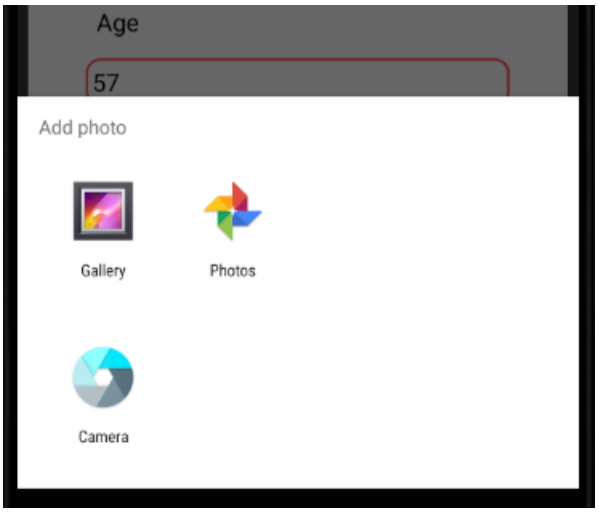


Figure 6.5: Bottom sheet

The Bottom Sheet in our application features is, as shown in figure 6.5, a modal type used as a menu of applications.The user is asked to choose between applications of the same kinds. The sheets display a suggestive title and lists of icons and names of the applications.

6.4 Wireframing

As mentioned in the methodology, wireframing is sketching out the layout and what info/buttons to put and where to put it in the application. We use it to establish the basic structure of our application, before the visual design is being implemented.

Jesse Showalter, who is a designer with specialization in web design, UI and UX design and front-end development, has made a video telling the best approach to wireframe.[90]

1. It is a good idea to identify the amount of content of each page and what type of content it is. Take also the target group and user type into consideration. For example, if it is an older target group, it would be a good idea to put larger images into the design. On the same note, it is important to know what the client's demands are, in order to meet their requirements.

2. The ideas for the design can be written down before starting to work, in order to give an overview of what should be put into the design.

3. Once the requirements are written down, the sketching can begin. Making multiple different sketches of the design is a good idea, since you can then choose what you like and don't like.

4. The ideas can then be combined and make a simple design in a wireframing software.

Taking this into consideration, we can make a prototype.
We started sketching out ideas of how the screens should look. The figure below shows the different ideas we had for the log-in screen.

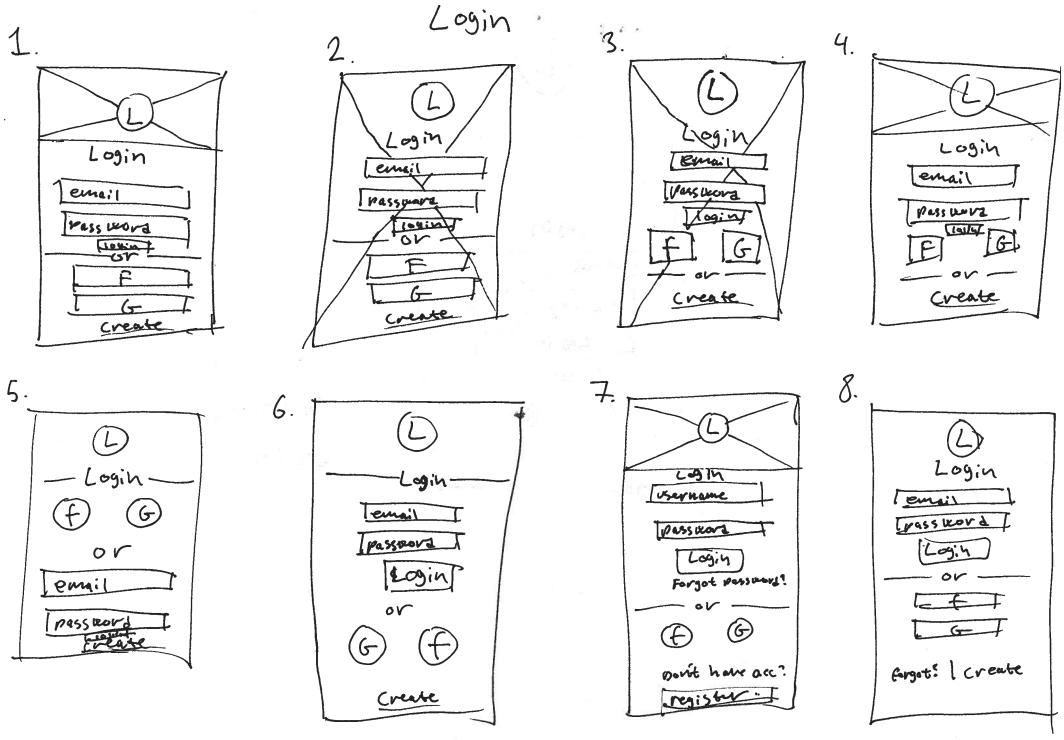


Figure 6.6: First step of login Wireframing

When having the general idea of how our application should look like, we made the prototype using the program 'JustInMind'.

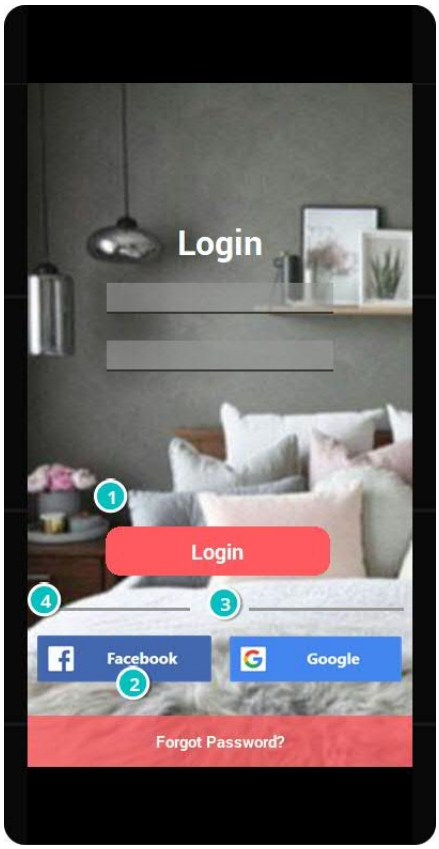


Figure 6.7: Login

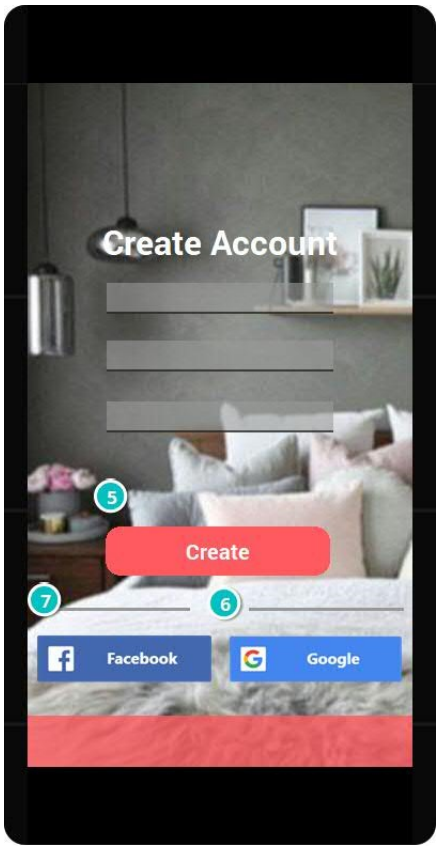


Figure 6.8: Sign Up

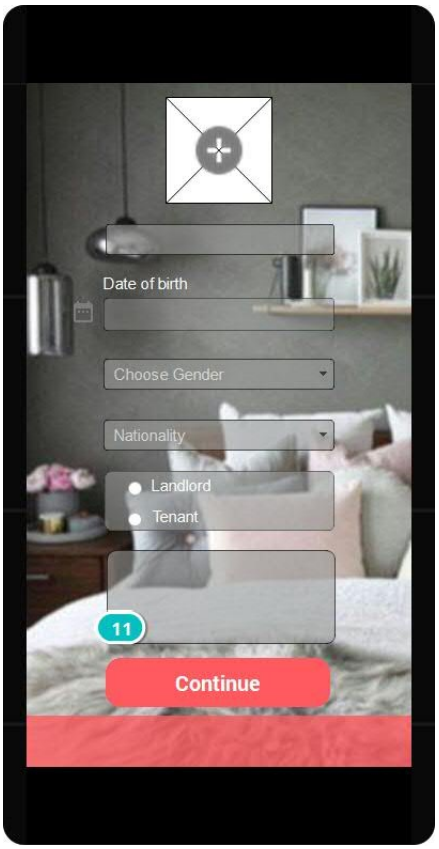


Figure 6.9: Create Account

In the figures above is some of the prototype screens. These might not be the final layouts for our application, since we might find out later if something has to be changed after testing it, but it gives us a good overview of what our application should look like.
With the prototype made, we can then test it with another group to see if we are missing to put in information, if something is needless etc.

6.5 Scrum

In WeRoom project, the role of product owner and developers are shared among the same people. Having multiple roles can be challenging for finding the project requirements. Product owner should focus on finding what problems the application should solve, while the developers main task is to find out how those problems are going to be solved, how the project is going to be designed and what the whole architecture of the product is.
We decided to use the Scrum as a process model for our project. The Scrum will bring the next features for the project:

1. **Agile model:** the development of the project will be done in different steps or in this case of Scrum sprints. During the development of the application the design and requirement can be changed. During the implementation more requirement might be introduced, or some of them can be removed.

2. **Incremental programming:** the project will be released in different version. On each version more functionalities will be included. This could give the project an opportunity to show the application to external actors and get some feedback. Version of the application will be released on each of the sprints during the Scrums.

3. **Daily Scrum:** A 15 minutes daily meeting is being hold every sprint day. The main focus of the meetings is to share the progress and challenges faced every day to make sure that the sprint goal is fulfilled in time.

4. **Sprint review:** At the end of every sprint a review is done. The product owner and main stakeholders are present with the Scrum team. The entire group collaborates on what to do next, so that the Sprint Review provides valuable input to subsequent Sprint Planning. In the reviews each of the developers also gives a feedback on how wide was the sprint. Getting feedback if the members feels if the sprint was to short with the given task or to long, gave us the opportunity to adapt in the future sprints with the amount of task that needs to be developed.

In the Scrum process every member should only have one role. In our case the Scrum team who is going to develop the project also has the role of being the product owner, and the scrum master. Each role has different approach in the project and should solve different tasks. Trying to address this problem in the project we will try to find as many requirement at the beginning of the development of the product, so the team members emphasize more in the product owner role.

During this period of time, we define the requirement of the project, and afterwards we create all the possible user cases needed to fulfill the product requirements. The project has to be developed in a span of 16 weeks. To plan the project and make sure that the product is going to be finalized in time we create a backlog with all the task that we can identify before starting the implementation. Once this step is finalized we can split this backlog in different sprints. Doing so, the role of a Scrum-Master is not biased during the implementation of the product.

We are using an agile model. This means that more requirements or changes can be done during the progress of the implementation. The backlog can be modified: Rewriting, adding or removing some of the task, if the group as a product owner sees this necessity.

During the process each of the sprint will be documented with the spreadsheet. This gives the scrum members the opportunity to have every scrum documented to check the past sprint reviews and if any functionality of the project should be implemented in the next spring that has not been implemented in the previous one. All the documentation regarding to the sprints and the contain information can be found in the annex.

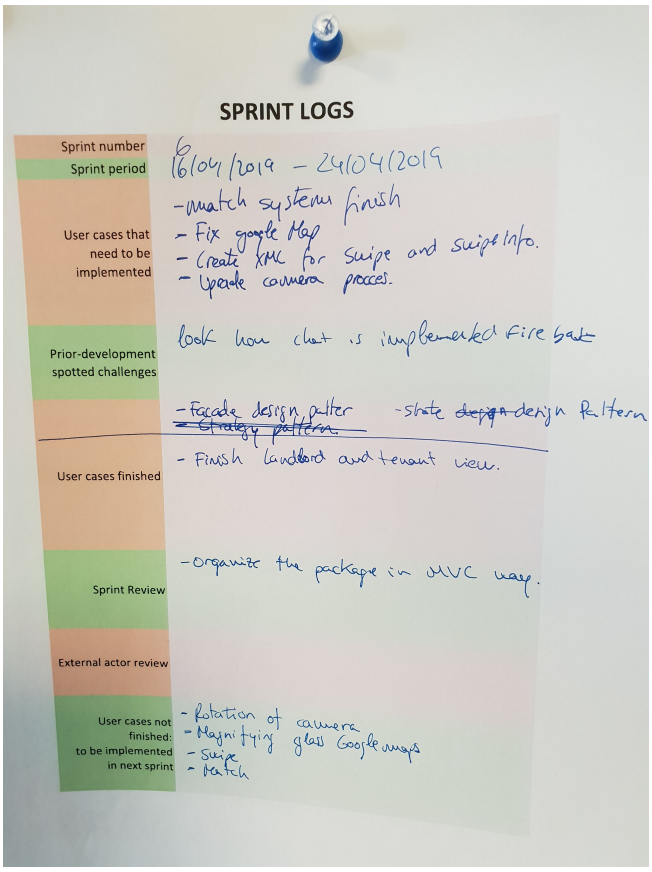


Figure 6.10: Sprint Log - All Sprint Logs can be seen in the Annex

6.5.1 Sprints

We started the project on the 11th of February and the first sprint is documented from the 8th of March. During this three weeks, the members of the project were brainstorming on the idea and requirement of the project.

In this time, we searched for different applications that were doing something similar to our application. We searched inspiration on real estate applications and websites, and also application that were matching people depending on their needs.

After the first week, we started working on the project requirements and we tried to identify as many requirements as we could find for both functional and non-functional. During this week we also started sketching the flow of our application on paper and whiteboard to have a common idea of how the screens of our application should be connected.

Figure 6.11: Designing the flow of the application

In the last week before starting the sprints, we did some interviews to get more requirements and to check that our idea was accepted by external actors to solve the problem formulated in the problem formulation. In this week we also did a system description, created some user stories, scenarios and use cases. In this period of time we worked on creating the first diagrams of our project to have a overview on what we had to implement, and how we would organize the job. After finishing the Context Diagram and Use Case diagram we decided that we would use Scrum from now on to organize the work flow of the project.

Sprint 1

Introduction

The first sprint started on March 8th and ended on March 15th, lasting in total for a week. We had a general overview of what our application should include, so for the first sprint, we decided to do the wireframing. With the wireframing we could make a working prototype and test it with another group to see if we were missing to display information, if it is user-friendly and to give ourselves an overview of all the different screens we need to implement. We used a program called JustInMind, which is a tool for making prototypes for applications, websites etc. We made an interactive prototype which could be displayed on a phone to easier get a feel of how the application should work. In fig.6.12 can be seen a diagram using the wireframing that is representing the possible flow of actions in the application.

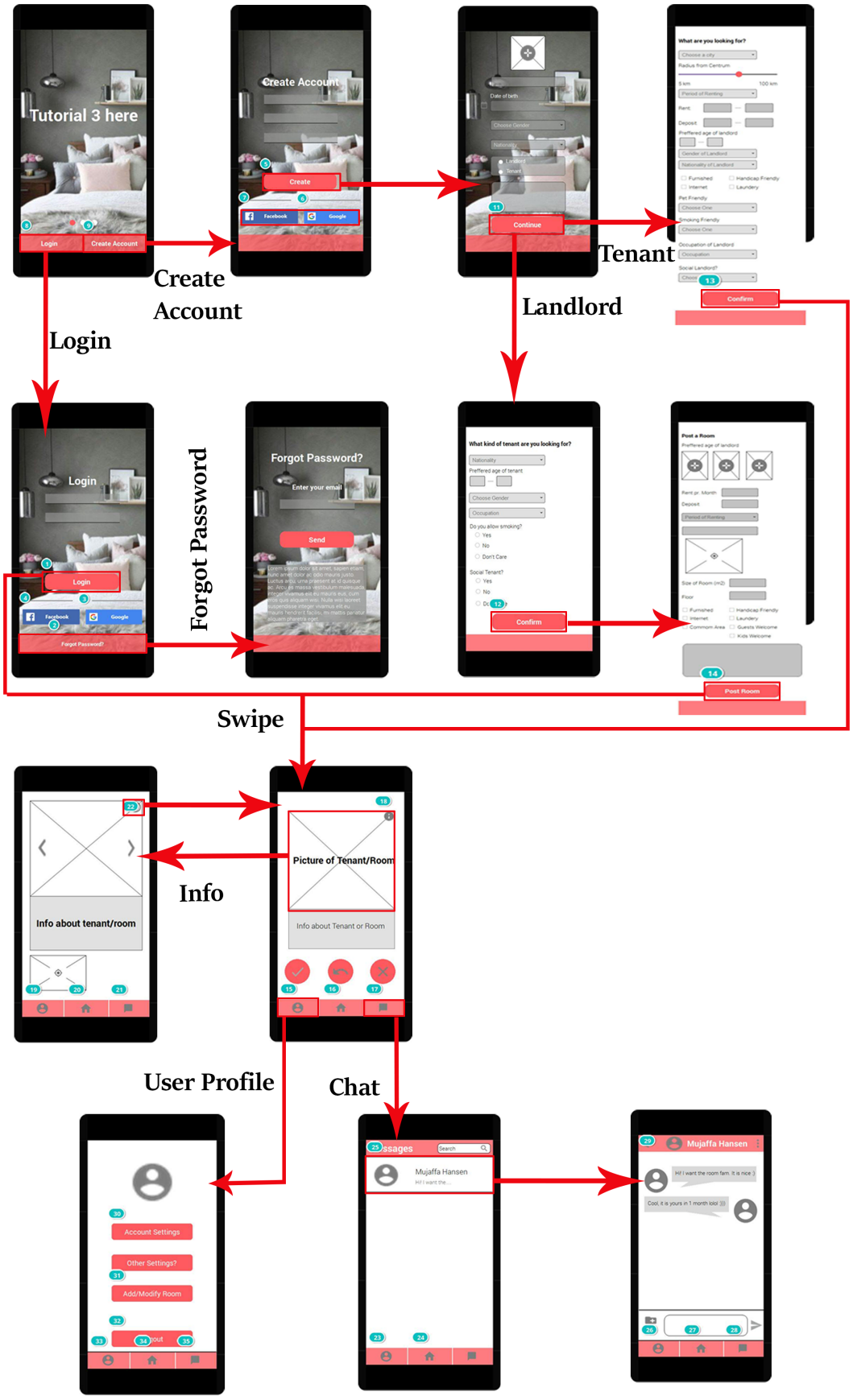


Figure 6.12: Connection of the screens in our application

For the implementation, we had to create the login and account creation for our application and connect it to Firebase. Moreover, we wanted to implement Smart Lock with Facebook and Google API intergration.

Implementation

As seen in the technologies chapter about Firebase and how it functions, we created a Firebase project on their website and had to link it to our application. To do so, we had to install the Google Repository to use the Firebase assistant in android studio, which helps explore and integrate the services Firebase provides. It has tutorials to connect our application to Firebase and provides us with the necessary code to get started with implementing.

Now that we had setup a connection between our application and Firebase, we needed to store the users that gets created. The Firebase Authentication SDK offers manual integration of one or more ways of sign-in methods in an application.

```
mFirebaseAuth.createUserWithEmailAndPassword( mEmail.getText( ).toString( ),
        mPasswd2.getText( ).toString( ) )
        .addOnCompleteListener( Objects.requireNonNull( getActivity( ) ), (task) → {
                if ( task.isSuccessful( ) ) {
                    FirebaseUser mUser = FirebaseAuth.getInstance( )
                            .getCurrentUser( );
                    Objects.requireNonNull( mUser ).getIdToken( b: true )
                            .addOnCompleteListener( (task) → {
                                if ( task.isSuccessful( ) ) {
                                    changeFragment( new ProfileFragment( ) );
                                } else {
                                    Toast.makeText( getActivity( ),
                                            R.string.login_process_error,
                                            Toast.LENGTH_SHORT ).show( );
                                }
                            } );
                } else {
                    Toast.makeText( getActivity( ), "Email already in use",
                            Toast.LENGTH_SHORT ).show( );
                }
        } );
```

Figure 6.13: Part of the implementation of the Firebase Authentication

If we want to sign in a user, we would first need the authentication credentials from the user. For our case, it is the users email and password. As seen in the code above, these credentials are then passed to the Firebase Authentication SDK, which will then verify them and return a response to the user. The user will then know if the email already exists, if some error happened etc. If everything is in order, Firebase will then give the user an ID in the database and save all criterias for the users profile under this ID.

Not only should the user be able to create an account or login with an email, but with Facebook and Google as-well. To do this, we had to use the Google API called Smart Lock. Using this API, the user is able to login or creating an account without entering an email/password combination, but instead do it with Google.[91] The API is not only limited to Google Accounts, but can support Facebook, Twitter,

LinkedIn among others as-well. All with their own buttons in the application of course.[92] But for our application we will stick to Google Accounts and Facebook. The diagram below, describes the typical flow for applications that uses Smart Lock.

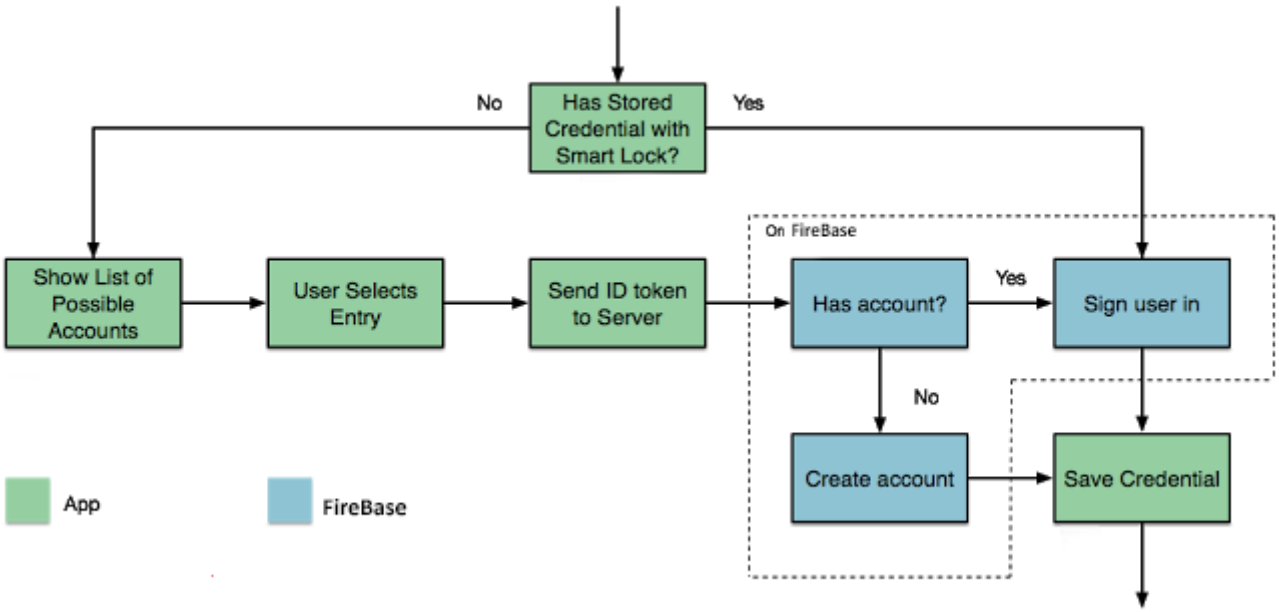


Figure 6.14: Flow for the Smart Lock
Source: https://www.androidauthority.com/how-to-integrate-smart-lock-in-android-apps-702638/

By looking at Figure 6.14, the following code below displays what is happening when the user selects an entry. It lists the Google accounts the user has linked to the phone, and shows them in a fragment. The user chooses an account to log-in with and an ID token then gets sent to the server. If a problem occurs while sending the ID token to the server, an error occurs and a toast is displayed saying the connection failed.

```
public void onActivityResult( int requestCode, int resultCode, Intent data ) {
    super.onActivityResult( requestCode, resultCode, data );
    //Callback for Facebook intent.
    FaceBookConnection.getCallBackManager( ).onActivityResult( requestCode, resultCode, data );
    // Result returned from launching the Intent from GoogleSignInApi.getSignInIntent(...);
    if ( requestCode == GoogleConnection.RC_SIGN_IN ) {
        Task<GoogleSignInAccount> task = GoogleSignIn.getSignedInAccountFromIntent( data );
        try {
            // Google Sign In was successful, authenticate with Firebase
            GoogleSignInAccount account = task.getResult( ApiException.class );
            assert account != null;
            GoogleConnection.firebaseAuthWithGoogle( account, getActivity( ), firebaseAuth,
                    getActivity( ), fragment: this );
            logUser( );
        } catch ( ApiException e ) {
            // Google Sign In failed
            Log.w( TAG, msg: "Google sign in failed", e );
            Toast.makeText( getActivity( ), "Google connection failed", Toast.LENGTH_SHORT ).show( );
        }
    }
}
```

Figure 6.15: Code for the Smart Lock

In continuation of the code above, when logging in with Smart Lock, we need to check if the user has a finished profile or not. To check this, we made the method as seen in Figure 6.16

```
private void logUser( ) {
    FirebaseFirestore db = FirebaseFirestore.getInstance( );
    DocumentReference docRef = db.collection( DataBasePath.USERS.getValue( ) )
            .document( Objects.requireNonNull( FirebaseAuth.getInstance( ).getUid( ) ) );
    docRef.get( ).addOnSuccessListener( (OnSuccessListener) (documentSnapshot) → {
            if ( documentSnapshot != null ) {
                mUserProfile = documentSnapshot.toObject( Profile.class );
                ProfileSingleton.initialize( mUserProfile );
                if ( ProfileSingleton.isFinishedProfile( ) ) {
                    startActivity( InteractionActivity.newIntent( activity ) );
                } else {
                    changeFragment( new ProfileFragment( ) );
                }
            } else {
                changeFragment( new ProfileFragment( ) );
            }
    } );
}
```

Figure 6.16: Method that checks if the user already has an account

This method search in Firebase Firestore if the user has a finished account or not. If the user haven't finished filling out their profile, it will open the ProfileFragment, since it is required . But if they have fulfilled everything, it will open the InteractionActivity instead.

External actor review

When we had the prototype ready, we made the first user acceptance testing. Here we did the testing with the other group. They were asked to mess around with it for a while and see if the flow and information displayed was good enough to start implementing.

The feedback we got from the testers, was to do some layout changes mostly involving the textfields. They should have a description on top of the textfield, describing what they are writing in the given field. Moreover they came with other things to add in the application, such as a star rating of the users and a tutorial the users are forced to go through when they open the app for the first time.

Review

After the first review, we had some feedback to share among the members. We found a GitHub project developed from Android with different example on how to connect an application with different functionalities from FireBase [93]. We also spotted that different members were coding at the same time in a connected task. One member developed the Google identification and another one developed the Facebook identification. Both solutions should have been similar, but when we integrated the connection into our Fragment, we displayed them in different shapes, buttons and colors. The intents were handled in a different way, when we could have had a similar implementation for both. We should aim to have consistency in the code when different developers were working.

Lastly, we over complicated the implementation. We spent many hours trying to manually store the FirebaseAuth credentials of the users in the application, so the user didn't had to insert the email and password every time they opened the application. We found out at the end of the sprint, that Firebase was doing this automatically [94], and was storing the credentials of the user.

Sprint 2

Introduction

The second sprint went from the 15th of March until the 26th of March. For this sprint the main goal was to model our application with class diagrams, sequence diagrams and state diagrams. We also wanted to connect our application to the gallery and phone provided by Android, and let users that have been authenticated with FireBase create a profile and store the information in FireBase databases.

The sprint also covered some research on material design - which can be found in the Material Design chapter, different databases types on FireBase, and design pattern on Java and Android. We realize that it was ambitious to do all this task in a sprint, and that is the main reason why we increased the time for the sprint for 7 days to 11.

Research

FireBase has the possibility to choose three different kind of databases: Real-Time Database, Cloud FireStore and Storage.

- Real-Time Database: offers a JSON[95] tree database. It is complex to create hierarchical data structures and organize them. Simple queries were you can sort or filter on a single property of the JSON. Database hosted in a region, will have high latency for distance regions.
- Cloud FireStore:[96] offers a document based database. Easier to maintain hierarchical data, as it allows to create sub collections with documents. Queries allows you to combine filtering and sorting on a single property. You can query subcollections within a document instead of an entire collection, or even an entire document. Is a cloud multi-region database offering higher global scalability. Cloud FireStore is a paid version compared to Real-Time Database[97].
- Firebase Storage:[98] is a Firebase cloud solution to store fotos and video files for web and phone applications. It gives security access to different users on Firebase Authentification, and it provides a control on the bandwidth when downloading or uploading different medias.

Before starting with the implementation, we did some research on design patterns that were used on Android and Java. We found a useful information on javatpoint website [99] and we printed out a small summary of each of the design pattern described in the website. At the beginning of each of the sprint we used these summaries to spot any implementation that could use a design pattern. For this purpose we did one page summary of each of the design patterns we could found and pin them to the walls (See Figure 6.17) so we could identify which of them we could use at the beginning of each of the sprint. This design patterns summaries can be found in the annex.

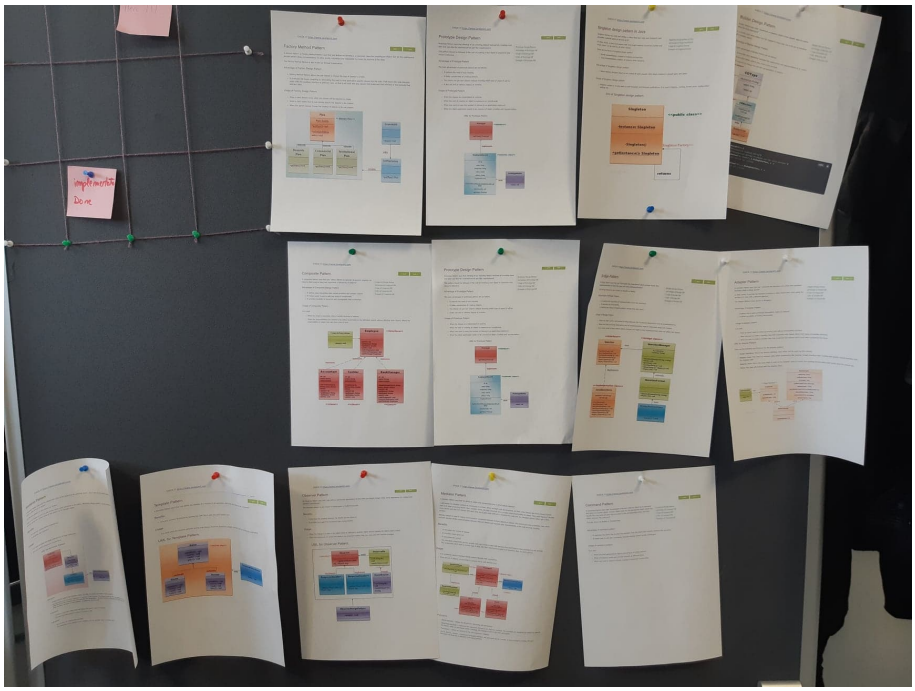


Figure 6.17: Design patterns board

Design

In this sprint we finished modeling our application. We did a class diagram to identify the models of our application, identifying what object we had to store and how we should organize them to have a logical structure in our database. In the next step before starting the implementation we wanted to establish how our models were going to interact between them. We created some sequence diagrams about the match system of our application and how the models will communicate with each other. Those diagrams gave us the understanding on how we should organize our models in the databases. Lastly, we did state diagrams trying to identify what was the state of the match model between tenants and rooms. It provided a list of states that both actors can be with each other. This assisted us creating the methods of the model Match where it should be able to go between all the state of the described model. All this models can be found in the chapter 5.2 Modelling the system.

In this sprint we found out that having a SingleTon design pattern could make easier to manage the data from the user who is using the application. SingleTon as a design pattern brings a single point to get some data on the entire application. It improve the consistency of the application, as we have only one Profile object were the data is stored among all the fragments and activities of the application.

Implementation

The SingleTon will be used among the entire project in almost all the fragments of the application, and it provides a single entry point to the current users Profile when called. This ensures that there will not be duplicated or inconsistent information during the flow of the application. Moreovere, it brings less resource consuming action than getting the user profile every time from the database when needed. Further we implemented a update method for the SingleTon to secure the database will have the same data as the instance of the application, preventing any miss match in the data between the application and the databases. (see 6.18)

```
public class ProfileSingleton {
    private static ProfileSingleton single_instance = null;
    private Profile userProfile;

    private ProfileSingleton( ) {
    }

    /**...*/
    public static synchronized Profile getInstance( ) {
        if ( single_instance == null ) {
            single_instance = new ProfileSingleton( );
        }
        return single_instance.userProfile;
    }

    /**...*/
    public static void initialize( Profile profile ) {
        if (single_instance == null){
            single_instance = new ProfileSingleton();
        }
        single_instance.userProfile = profile;
    }

    /**...*/
    public static void update( Profile profile ) {
        single_instance.userProfile = profile;
        FirebaseFirestore db = FirebaseFirestore.getInstance( );

        db.collection( DataBasePath.USERS.getValue( ) )
                .document( profile.getUserID( ) )
                .set( profile );
    }
```

Figure 6.18: SingleTone implementation for Profile handling in WeRoom

Adapter design pattern was also used in the sprint to create Spinners where the user could select their nationality. Adapters brings objects that allows two previously incompatible objects to interact[100]. In our case the spinners need to display a list of countries. We can get a String of 'Locale' objects using Java API having a String with all existing countries in the world. To be able to display this information in a Spinner they need to be adapted to a structure that the Spinner recognize and here is where the SpinnerAdapter has a key value.

During the implementation of the sprint, we decided to have an Enum class that provided more robustness when accessing to the database. All values that had to be hard-coded to get to folders or subfolder from the database will be represented in this Enum.

```
public enum DataBasePath {
    TENANT( s: "tenant" ),
    USERS( s: "users" ),
    ROOMS( s: "rooms" ),
    PROFILE_PICTURE( s: "profile_picture" ),
    ROOM_PICTURE( s: "room_picture" ),
    IMAGE( s: "images" ),
    CHAT( s: "chat" );

    private final String name;

    DataBasePath( String s ) { name = s; }

    public String getValue( ) { return name; }

}
```

Figure 6.19: Enum class that contains values to access the Databases in WeRoom.

Last at this sprint, we decided to use Real-time database. At the time we could not justify the cost of having a better scalability that Cloud FireStore would bring in the future. We didn't have any need of splitting the database in any sub folder, or making a query on any of the sub collections of the database before later on in the project.

Managing any connection with Firebase Real-time database is quite straight forward. The application has integrated the SDK from Firebase in the project from the first sprint and the credentials are already set up to get validated from Firebase. Inside the code its only needed to initialize a instance connection with the database, to push updates or get data from Real-time Database.

```
DatabaseReference messageRef = FirebaseDatabase.getInstance( )
        .getReference( DataBasePath.CHAT.getValue( ) ).child( mChatID );
String key = messageRef.push( ).getKey( );
if ( key != null ) {
    messageRef.child( key ).setValue( chatMessage );
}
```

Figure 6.20: Writing data to Real-Time Database.

```
DatabaseReference messageRef = FirebaseDatabase.getInstance( )
        .getReference( DataBasePath.CHAT.getValue( ) ).child( mChatID );
messageRef.addChildEventListener( new ChildEventListener( ) {
    @Override
    public void onChildAdded( @NonNull DataSnapshot dataSnapshot, @Nullable String s ) {
        Message message = dataSnapshot.getValue( Message.class );
        chatMessages.add( message );
        adapter.notifyDataSetChanged( );
    }
```

Figure 6.21: Create a listener that reads data from Real-Time Database on any new document added.

Review

For the sprint review, we agreed that it would suit us better to start and finish the sprint on other days than Monday or Friday, so we could use the weekend to work and discuss any research of implementation that we had done.

Sprint 3

Introduction

Third sprint of the project was done between the 26th of March and 2nd of April. We started having Sprint meetings and reviews on Tuesdays as discussed in the last sprint review, to increase the communication and productivity of our weekend work.

For this sprint, the main goal in the implementation was to finish the on boarding process of the authenticated users by creating a Tenant profile or a Landlord profile with at least one room. To accomplish this task we had to storage all the pictures from the profiles and the rooms into Firebase Storage database.

Research

For the research of the sprint we had to learn how to use Firebase Storage and what the best way to store and get back all the pictures for each of the users and rooms was. Also in the room creation as a requirement in the project we had to verify that the users type a existing address in the room. After some research we found that Google API Places offer a auto-complete box with existing addresses.

Using Google Maps and Google Places requires to integrate Google API and having a working payment plan to be able to use them. Google offers 300$[101] credit when enabling the cloud platform for the first time and can be spent in the following 12 month since the creation. To enable a Cloud platform for the project only a Google account is needed, and after accepting the free trial plan you have access to their platform. The platform offers different API and services that need to be activated individually. Once the Maps and Places were activated we have to get a credential that is used on the application.

Figure 6.22: Example of a API key from Google Cloud Platform

Design

Regarding to the design of this sprint, all models will be created with a builder design pattern. Using this pattern ensures readability and makes the process of initializing an object easier. Builder patterns also brings more flexibility in the future if more variables will be added to the models, we don't have to re-factor all the constructor available for the model, we only have to create one more method for the builder.

Implementation

Firebase Storage works in a similar way as Real-Time Database explained in the sprint 2. When uploading a picture into the Storage we had two ways to store the reference of the picture. One option is that we could have created a string in the Profile models with a URL where the picture is stored in the database. The second option that we used was to structure the storage with buckets for each of the profiles. Each of the buckets will be named with their FireBase Authentification unique ID. This way we didn't have to store any extra information in the models and we would create a controller that could store or get any picture on a specific bucket giving only the Bitmap picture capture from the application and the user authentication ID.

```
public static void setProfilePicture( String userID, Bitmap bmp ) {
    StorageReference reference = FirebaseStorage.getInstance( ).getReference( );

    reference
            .child( DataBasePath.IMAGE.getValue( ) )
            .child( userID )
            .child( DataBasePath.PROFILE_PICTURE.getValue( ) )
            .putBytes( PictureConversion.bitmapToByteArray( bmp ) );
}

/**...*/
public static Task getProfilePicture( String userID ) {

    StorageReference reference = FirebaseStorage.getInstance( ).getReference( );
    StorageReference downloadRef = reference
            .child( DataBasePath.IMAGE.getValue( ) )
            .child( userID )
            .child( DataBasePath.PROFILE_PICTURE.getValue( ) );

    return downloadRef.getBytes( Long.MAX_VALUE ).addOnSuccessListener( (OnSuccessListener) (bytes)
            byteArray = bytes;
    } );
}
```

Figure 6.23: Method to upload and download profile pictures from the class ImageController.

As seen in figure 6.23, the getProfilePicture method doesn't return a Bitmap picture, but a task related to it. All the queries done to any of FireBase databases are asynchronous task and can not be performed in the UI thread of the application[102]. The UI thread or main thread of the application can not be stopped until we get the answer from a database (if we manage to do this on the UI thread, the phones operative system will close our application immediately). Therefore, if we create a method that creates a query and returns the answer of that query, the UI thread will always get a null value from the method. Example of this hypothetical method signature 6.24.

```
public static Bitmap getProfilePicture( String userID )
```

Figure 6.24: This method will always return null if the Bitmap is given from a Firebase database query.

We can fix this issue working with tasks. When we call the method getProfilePicture we have to handle the task on the fragment asynchronously. We can create a listener on the task and execute some code when the task is being completed, in our case get the desire picture and update some values in the fragment 6.25

```
Task t = ImageController.getProfilePicture( p.getUserID( ) );

t.addOnSuccessListener( new OnSuccessListener<byte[]>( ) {
    @Override
    public void onSuccess( final byte[] bytes ) {
        mPicture = PictureConversion.byteArrayToBitmap( bytes );
        mProfilePhoto.setImageBitmap( mPicture );
    }
} );
```

Figure 6.25: Creates a onCompleteListener and when the query is success it will execute some code with the snapshot gotten from he database.

For enabling Google Places API in the application we have to initialize the instance of Google Places with the API key we got from Google Platform. The key is being store on R.string as a value in case that it is been needed in future fragments/activities. The autocompleteFragment allows the user to type any address and have it auto-completed. The fragment works in the way that you can only have a valid address when typing, and won't accept any address that is not being recognized by Google.

```
// Initialize Places. Validates with Google API key.
Places.initialize( getApplicationContext( ), getString( R.string.google_cloud_api_key ) );

// Create a new Places client instance.
Places.createClient( getActivity( ) );

// Initialize the AutocompleteSupportFragment.
AutocompleteSupportFragment autocompleteFragment = ( AutocompleteSupportFragment )
        getChildFragmentManager( ).findFragmentById( R.id.autocomplete_fragment );
```

Figure 6.26: Get validated on Google API and initialize Places Autocomplete to verify WeRoom typed addresses.

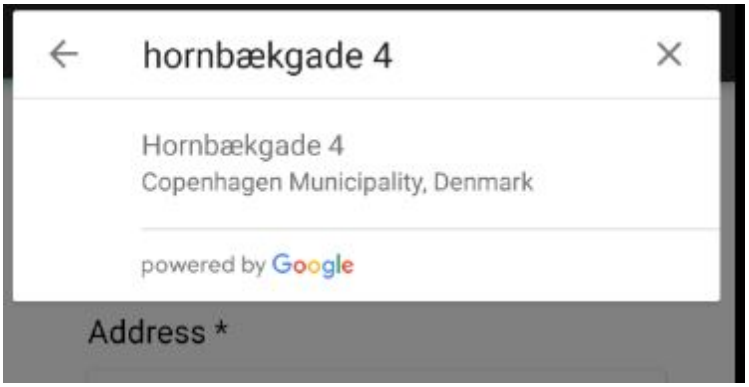


Figure 6.27: AutocompleteFragment working on WeRoom application.

This sprint we decided that we would use Builder design pattern for our profiles models in the application. The usage of the Builder has been already explained in the chapter 5.2.3 when the class diagram was introduced (See figure 5.7). During this sprint we discussed how we could ensure that all the data we

introduced in the databases had meaningful value on the models. We stated as an example, that it would not be consistent to allow users to have a room price with a negative number for example. To ensure that our models were relevant we decided that we would throw exception on run time if we introduced inconsistent information on the models as seen in Figure 6.28.

```
public Builder withRentingPeriod( int period ) {
    if ( period != LESS_THAN_THREE_MONTH && period != THREE_TO_SIX_MONTH &&
            period != SIX_TO_TWELVE_MONTH && period != OVER_TWELVE_MONTH )
        throw new InputMismatchException( "Wrong period time" );
    sPeriodOfRent = period;
    return this;
}
```

Figure 6.28: Example of TenantProfile Model. We only allow certain values in our models to ensure consistency on our database. The views have to ensure that the models never throw an exception.

Review

Implementing this exception on the builder would ensure that all data introduced to the databases were correct during the implementation of the application. The Views in the application (in this case, the fragment that creates the profiles on the package on-boarding') should ensure to validate the data of the user before pushing an object to the database. Furthermore, in future sprints when testing our application we will run monkey tests on the application, to ensure that we are handling all users input and detect if any invalid data has been introduced.

Sprint 4

Introduction

Sprint number 4 started on 2nd of April and ended on the 9th of April. When we started the sprint, we still had to finish CreateRoom implementation from the previous sprint. For that week we had taken some task from our backlog as-well, such as implementation of a swipeable card and information card, that should be opened on click of a swipe card.

Research

To be able to implement the swipeable cards we needed to make some research on how actually Android is dealing with gestures.[103] Moreover, we knew we wanted to use Recycler View in that part of the app.[104] The Recycler View is usually used to display scrolling list of elements, with the difference from the Scroll View that the Recycler is creating a certain amount of elements at a time and not all the elements from the beginning. It should be used when there is a data collection whose elements change at runtime based on user action or network events. The Recycler needs to have a layout manager and an adapter to be instantiated. A layout manager positions item views inside a RecyclerView and determines when to reuse item views that are no longer visible to the user.[105]. Knowing this, in our SwipeFragment, where we are showing the matching tenants/rooms to the user, we didn't wanted to create the view of all of the possible matches at the beginning as this would result in increased response time and bad performance of the app.

After that, we wanted to adopt the idea of a *card* looking view as in Tinder. To do so we decided to use Card View, which is used to wrap the layout, elevate it and make the corners round. It is often the container is used in a layout for each item within a ListView or RecyclerView.[106] Therefore, this was the perfect thing to use to meet our goal.

The Card itself is coming from Material Design[107]. From there we have taken some inspiration as well, such as the way we wanted to look the transitions between cards, and gestures with cards. This can also be seen in the Material Design chapter.

The swipeable cards are the focus aspect of WeRoom. Because of that we wanted to make them functional and well looking at the same time. To do so, we included an external library[108] to help us deal with having a stack of cards in recycler view and only allow swiping gesture to left and right direction. It is a simple clean library providing us with just the basic functionalities. We used that to build on top of it all the functions that we needed.

Implementation

After researching how to do the CardView, as we said at the beginning of the sprint, we had some unfinished work from the previous sprint. We needed to finalize the flow for account creation, both for tenants and landlords. Therefore, we finished the implementation in the class CreateRoomFragment. We needed to validate all the data that a user is writing, if it was between the parameters put in the builder class for RoomPosted class. To do so we used if-else statements in the method postRoom before pushing all the data to the database (see in fig 6.29).At the end of that method we are pushing all the data, because we are sure that is meeting the parameters in the builder class. If by any chance the parameters are not met despite passing the if-else checking at the beginning, at the last else in the method we are using the Builder to create the room, which will throw an exception and crash the app. (see fig 6.30)

```
private void postRoom( final boolean once ) {
    if ( mRent.length( ) == 0 || Integer.parseInt( mRent.getText( ).toString( ) ) < 0 ) {
        mRent.setError( "Please type rent" );
        mRent.requestFocus( );
    } else if ( mDeposit.length( ) == 0 ) {
        mDeposit.setError( "Please type deposit" );
        mDeposit.requestFocus( );
    } else if ( mPeriodRenting.getSelectedItemPosition( ) == 0 ) {
        TextView errorText = ( TextView ) mPeriodRenting.getSelectedView( );
        errorText.setError( "" );
        errorText.setTextColor( Color.RED );
        errorText.setText( "Choose period of renting *" );
    } else if ( mAddressName == null ) {
        Toast.makeText( getContext( ), "Please type the address", Toast.LENGTH_SHORT ).show( );
    } else if ( mRoomSize.length( ) == 0 ) {
        mRoomSize.setError( "Please type the size of the room" );
        mRoomSize.requestFocus( );
    } else if ( mRoomDescription.length( ) == 0 ) {
        Toast.makeText( getContext( ), "Please describe your room", Toast.LENGTH_SHORT ).show( );
    } else if ( mRoomPictures.size( ) < 3 ) {
        Toast.makeText( getContext( ), "The rooms needs three pictures", Toast.LENGTH_SHORT ).show( );
    } else {
        final RoomPosted input = new RoomPosted.Builder( mUserId )
                .hasCommonAreas( mCommonArea.isChecked( ) )
                .hasInternet( mInternet.isChecked( ) )
                .hasLaundry( mLaundry.isChecked( ) )
                .isFurnished( mFurnished.isChecked( ) )
                .withAddress( mAddressID, mAddressName, mAddressLatitude, mAddressLongitude )
                .withDeposit( Integer.parseInt( mDeposit.getText( ).toString( ) ) )
```

Figure 6.29: Method for posting a room

```
public Builder withRent(int rent){
    if(rent<0)
        throw new InputMismatchException("Wrong rent");
    sRent = rent;
    return this;
}

public Builder withDescription(String s){
    if(s.equals(""))
        throw new InputMismatchException("Wrong description");
    sDescription = s;
    return this;
}
public Builder withDeposit(int deposit){
    if (deposit < 0)
        throw new InputMismatchException("Wrong deposit");
    sDeposit = deposit;
    return this;
}
public Builder withPeriodRenting(int position){
    if(position < 0 || position > 4)
        throw new InputMismatchException("Wrong period");
    sPeriodOfRenting = position;
    return this;
}
```

Figure 6.30: Builder for room with exceptions

Lastly, we wanted to let the user know that the room had been posted when the post room button is clicked. As seen in the chapter on Material Design, we implemented a dialog, saying the room has been posted. From here the user can press 'close', and the user can then either continue to the swipe screen or post another room.

We finalized the flow for creating an account by finishing the code for posting a room. As well, the research for the swipeable cards was done. Therefore we could continue with its proper implementation in WeRoom. As we have mentioned, the Recycler View was working with adapters. We wanted to use the same card layout for showing information of rooms and tenants. To do this we used custom adapter with multiple ViewHolders. In the adapter class we were checking if the role of the logged in profile is a landlord or a tenant (see fig. 6.31). If the user is a landlord the method will return 0, and if the user is a tenant it will return 2. Taking this return value, in the method onCreateViewHolder we were inflating the relative layout depending on the role.

```
@Override
public int getItemViewType( int position ) {
    if ( p.getRole( ).equals( "Landlord" ) ) {
        return 0;
    } else {
        return 2;
    }
}
```

Figure 6.31: Changing the holder depending on the role

In the method onBindViewHolder, again depending on the return value from getItemViewType, we are initializing the Holder and then using the method bind(see fig6.33) to bind the data from the holder with the card layout.

```
@NonNull
@Override
public RecyclerView.ViewHolder onCreateViewHolder( @NonNull ViewGroup parent, int viewType ) {
    switch ( viewType ) {
        case 0:
            return new TenantHolder( LayoutInflater.from( parent.getContext( ) )
                    .inflate( R.layout.swipe_card_landlord, parent,  attachToRoot: false ) );

        case 2:
            return new RoomHolder( LayoutInflater.from( parent.getContext( ) )
                    .inflate( R.layout.swipe_card_tenant, parent,  attachToRoot: false ) );
    }
    return null;
}

@Override
public void onBindViewHolder( @NonNull RecyclerView.ViewHolder viewHolder, int i ) {
    switch ( viewHolder.getItemViewType( ) ) {
        case 0:
            TenantHolder viewLandlord = ( TenantHolder ) viewHolder;
            viewLandlord.bind( mTenantList.get( i ) );
            break;

        case 2:
            RoomHolder viewTenant = ( RoomHolder ) viewHolder;
            viewTenant.bind( mAllRoomPosted.get( i ) );
            break;
    }
}
```

Figure 6.32: Custom Adapter, depends on the role

```
public void bind( Profile profile ) {
    mProfile = profile;

    Locale l = new Locale( language: "", mProfile.getCountry( ) );

    mTenantName.setText( mProfile.getName( ) );
    mTenantNationality.setText( l.getDisplayCountry( ) );
    mTenantAge.setText( String.format( Locale.getDefault( ),  format: "%d", mProfile.getAge( ) ) );

    Task t = ImageController.getProfilePicture( mProfile.getUserID( ) );
    t.addOnSuccessListener( (OnSuccessListener) (bytes) → {
            Bitmap bmp = PictureConversion.byteArrayToBitmap( bytes );
            mPhoto.setImageBitmap( bmp );
    } );
}
```

Figure 6.33: Method for binding the data from profile to the holder

In the bind method in fig 6.33 we are binding all the information that need to be showed in the swipeable card: name, age, nationality, photo. More on how we retrieve that information will be presented in a later sprint.

The next step for this sprint was to open a new fragment with the missing information from the swipeable card about the tenant/room. And after doing that, the user should be able to go back to the swipe screen and actually swipe on that same card, that he/she looked more information for.

```
public TenantHolder( @NonNull final View itemView ) {
    super( itemView );

    mTenantName = itemView.findViewById( R.id.text );
    mTenantNationality = itemView.findViewById( R.id.nationalityOfTenant );
    mTenantAge = itemView.findViewById( R.id.ageOfTenant );

    mPhoto = itemView.findViewById( R.id.photoCard );
    mPhoto.setOnClickListener( (v) → {
            AppCompatActivity activity = ( AppCompatActivity ) itemView.getContext( );
            Fragment myFragment = CardInfoTenantFragment.newInstance( mProfile );
            activity.getSupportFragmentManager( ) FragmentManager
                    .beginTransaction( ) FragmentTransaction
                    .add( R.id.fragment_container_top, myFragment ) FragmentTransaction
                    .addToBackStack( null ) FragmentTransaction
                    .commit( );
    } );
}
```

Figure 6.34: Holder Constructor

As it can be seen at fig. 6.34, we are initializing the new fragment with the static method newIntent from CardInfoTenantFragment which requires a profile in order to bundle the information and pass it through the intent (with the interface Parcelable). To solve the problem with going back from the information fragment to the same swipe-able card, we implemented a button in the pop-up fragment. This button is using the same code as the default return button for going back in Android.

Review

In the sprint review for this sprint, we discussed that we had to try to have the scrum meetings even if people are missing. We would try to have a scrum meeting using Skype or any other video chat application. We also spoke about productivity. We agreed that we should be more productive when meeting at the group rooms. Time could have used more efficiently in the past weeks, and we had a general feeling that in the past 3 sprints we always had some unfinished tasks.

Sprint 5

Introduction

Sprint number five started on the 9th and ended on the 16th of April. During that period of time we had several task to do. After implementing the Swipeable Cards and the fragment for Card Info, it was a logical continuation to make them function as they should, to query the right information from the database and populate those cards.

Research and Database Design

To query information from the database we need to apply to the query the requirements for the room/tenant that the tenant/landlord had put when creating the profile. Said in other words, we need to filter the database, so the data we get is within the implied limitations. To do that we need to query on more than one field at a time. This means that we should be able to query for example rent above 100 and bellow 200 and age above 20 and bellow 50. Unfortunately we found out that Non-SQL databases do not support that functionality. This restriction has been discussed in the project limitation. To be able to deal with that, without changing the database at a so late point of the project, we decided to query a list of all

the tenants/rooms and execute the filtering in the phone. We are aware of the fact that this is not the best solution of the problem, but given the amount of time left and number of tasks in the backlog, the app would be functioning. However,it will not be a finished market product, ready to work with bigger amount of data.

As discussed in sprint 2, there are some differences when comparing Realtime database and Firestore database, both Non-SQL and provided by Firebase. Until sprint 5, the project has been storing all its data in a Realtime database. The way we found best to structure the data and query it, was to build a hierarchical order. In this order we can store as document every user and then in it as a field will be the information if it is a landlord or a tenant and in a subfield the relevant information for the role. The reason behind that structure was:

- every user has a profile with basic information in it
- the user should be able to change role

But when researching more about Firestore Cloud, we found that it is better supported and it allows you to make more complex queries. For example, you can query subcollections within a document instead of an entire collection[109]. Because of that, we have decided to move to Firestore and use that database from now on. We have not migrated any data, all the data in the previously used database was deleted.

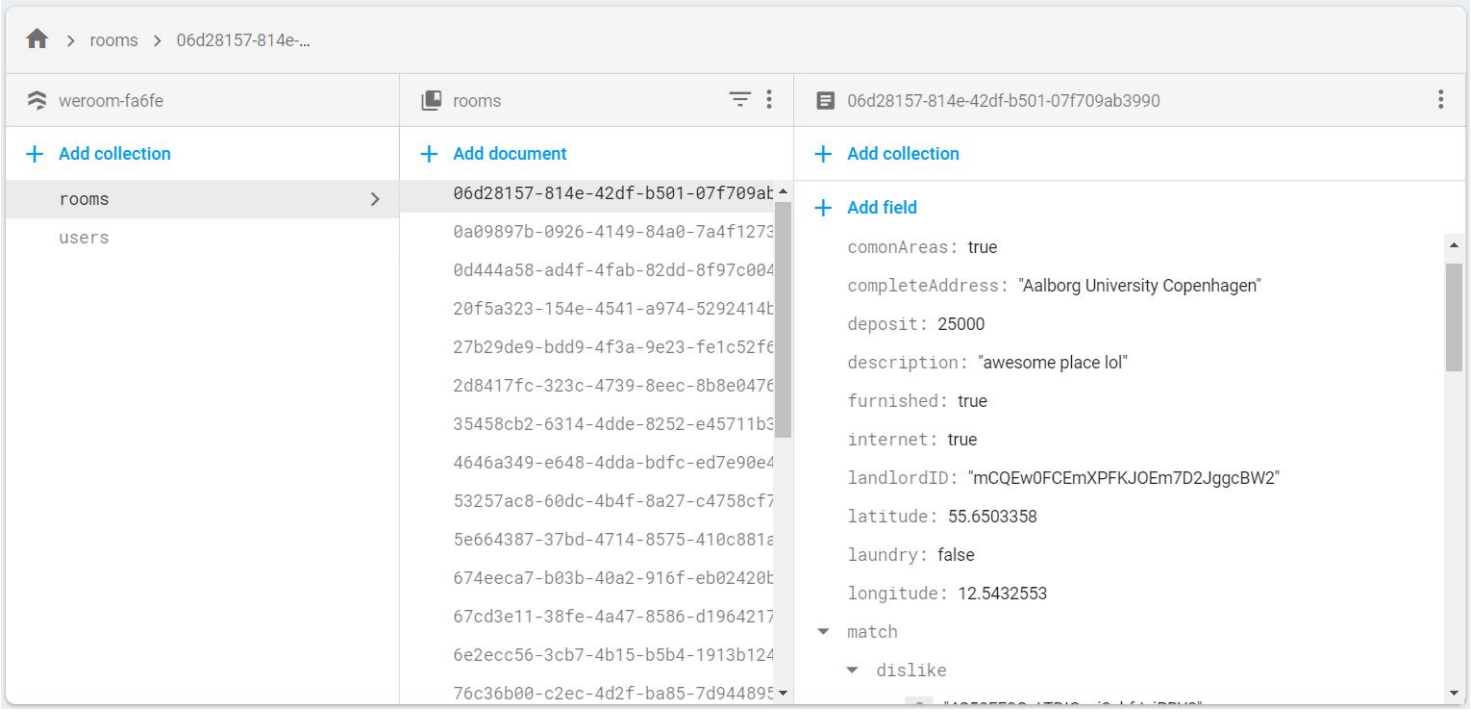


Figure 6.35: FireCloud console for WeRoom application

Implementation

As the database changed, the queries changed accordingly. We had to query on a child added before, and from now on - on a document.

```
private void startSwipeFragmentFromLandlord( ) {
    Profile p = ProfileSingleton.getInstance( );

    Query tenantCandidateQuery = FirebaseFirestore.getInstance( )
            .collection( DataBasePath.USERS.getValue( ) );

    Query removeNonTenant = FirebaseFirestore.getInstance( )
            .collection( DataBasePath.USERS.getValue( ) )
            .whereEqualTo( field: "tenant", value: null );

    Query getLandlordsRooms = FirebaseFirestore.getInstance( )
            .collection( DataBasePath.ROOMS.getValue( ) )
            .whereEqualTo( field: "landlordID", p.getUserID( ) );
```

Figure 6.36: Query data for Landlord

In figure 6.36 can be seen queries. They are in the method startSwipeFragmentFromLandlord. As the name says, if the role of the user is a Landlord, in the swipeable cards should be shown rooms. Thus, to do that we get a list of all the rooms and their relevant information. We have three queries and three according tasks. From the query tenantCandidateQuery we are getting all the users. From removeNonTenant we are getting all the users where the value for the field "tenant" is null. For getLandlordRooms we are getting all the rooms for that specific landlord, by comparing the landlordID for all the fields in the document *rooms*.

```
Task t1 = tenantCandidateQuery.get( ).addOnSuccessListener( (OnSuccessListener) (queryDocumentSnapshots) → {
        for ( DocumentSnapshot d : queryDocumentSnapshots ) {
            mAllProfilesFromQuery.add( d.toObject( Profile.class ) );
        }
} );
Task t2 = removeNonTenant.get( ).addOnSuccessListener( (OnSuccessListener) (queryDocumentSnapshots) → {
        for ( DocumentSnapshot d : queryDocumentSnapshots ) {
            mNonTenantProfiles.add( d.toObject( Profile.class ) );
        }
} );

Task t3 = getLandlordsRooms.get( ).addOnSuccessListener( (OnSuccessListener) (queryDocumentSnapshots) → {
        for ( DocumentSnapshot d : queryDocumentSnapshots ) {
            mLandlordsRooms.add( d.toObject( RoomPosted.class ) );
        }
} );

Tasks.whenAllSuccess( t1, t2, t3 ).addOnSuccessListener( (OnSuccessListener) (list) → {
        startFragmentFromLandlord( );
} );
```

Figure 6.37: Tasks for queries for Landlord

In fig.6.37 can be seen the three tasks related to the three queries in fig.6.36. The first task t1 is adding all the user data from tenantCandidateQuery as a Profile in an ArrayList of Profiles mAllProfilesFromQuery. Task t2 and t3 are analogous to t1. We end up with three ArrayLists- one with all the users, one with all the landlords and one with all the rooms that the specific landlord has. At the end of fig.6.37 can bee seen a listener which is waiting for all of the three tasks to finish execution and creating the fragment with swipeable cards for landlord. This is improving the user experience because it is reducing the time that is

needed to open the fragment, because all three tasks are executed in parallel and are not waiting for each other.

```
private void startFragmentFromLandlord( ) {
    startNotificationService( );
    mAllProfilesFromQuery.removeAll( mNonTenantProfiles );
    Bundle bundle = new Bundle( );
    ArrayList<Profile> filteredTenants = FilterController
            .filterProfilesFromLandlord( ProfileSingleton.getInstance( ), mAllProfilesFromQuery );
    bundle.putParcelableArrayList( SwipeFragment.KEY_TENANT_LIST, filteredTenants );
    bundle.putParcelableArrayList( SwipeFragment.KEY_ROOM_LIST_LANDLORD, mLandlordsRooms );


    FragmentManager fm = getSupportFragmentManager( );

    if ( swipingFragment == null ) {
        swipingFragment = new SwipeFragment( );
    }
    swipingFragment.setArguments( bundle );
    fm.beginTransaction( )
            .add( R.id.fragment_container_top, swipingFragment )
            .commit( );

}
```

Figure 6.38: Start fragment for Landlord

In fig.6.38 can be seen the method for starting the InteractionActivity. We use the two array lists of all the users and all the landlords that we obtained(see fig.6.36 and fig.6.37). We remove all the landlords from the all users list, so we end up with a list of tenants. We had to it in that way, because in Firestore it is not possible to query if a field has some data, but if a field is null. Hence we got the landlord list by taking all the users where the field "tenant" is *null*.

To pass the ArrayList of tenants/rooms through the intent of the new fragment we need to use the interface *Parcelable*. This interface is similar to *Serializable*, but faster in execution. Serializable is a standard Java tagging interface. We have been using it to serialize objects when we push the data to the database. However, reflection is used during the process and lots of additional objects are created along the way. This can cause lot's of garbage collection resulting in poor performance and battery drain[110]. Therefore, we implemented Parcelable specially to use when we are passing objects through intents. It works in a similar way, serializing objects, but it is part of Android SDK, no reflection needed. Now when we have explained that, the Parcelable interface requires the use of an ArrayList to pass a list of objects through an intent. Because of that fact, we could not declare the profiles list as a List interface, although it would be a better implementation. After that in the class CardInfoTenantFragment we pass that ArrayList in a bundle. Every element in that list is a profile. From that object we take all the relevant information for the specific tenant and display it on the card.

External actor review

Moreover, at the end of that sprint we made Acceptance testing. The testing group did not have any previous information about the application and its implementation. The acceptance criteria was for the testers to be able to navigate freely, without assistance in the app, to create a profile nevertheless the role. As a feedback we got that the purpose MapFragment in RoomCreationFragment is confusing for the user. The map is only to display the address that had been typed, not to search an address on the map itself.

Hence the group has decided to hide the MapFragment and show it only when an address is typed and verified.

Review

In this sprint we agreed that we should do some more research before implementing tasks. It would also be a good idea to research on the tasks the sprint before it should be implemented. We use one sprint to research about a topic, and we implement it in the next sprint. We want to avoid implementing the same task several time like we did with the databases. Maybe if we had researched more about the databases and the difference between SQL and non-sql databases we could have solved a lot of problems and code re-factoring.

The code still gives some exceptions when working with the models on the on boarding of new users process. We should run some monkey testing in that area of the code and solve all the data error before continuing.

Sprint 6

Introduction

The sixth sprint started on the 16th of April and ended on the 24th of April. For this period, the goals we set are regarding both new features to implement and unfinished work from the previous sprint.

The first task was to save the matches between tenants and rooms in the database. For this, we had to use the model from MVC design pattern, that we have previously discussed in the Chapter 5.1 Architecture.

The next task referred to fixing a few aspects about the map view we have in the RoomCreationFragment class. The map shows the exact location of the address written in the search bar above it, but a problem we have to deal with is that the map does not work on all Android phones and emulators because of limits regarding the Android version and whether or not the Google play store services are installed. The other problem that needed to be fixed about the map was that it displayed a small magnifying glass symbol, which when pressed, crashed the application.

Another task for this sprint was finding a solution to have only one button for adding pictures either with the camera or from the gallery. By that time, we had two separate buttons for accessing the two options.

The last task referred to some changes that had to be made on the layout of the swiping cards. These changes were to be done on the XML files of the cards.

For our first task, we have created the class; Match, which handles the matches from our application. There are several user-actions that have to be taken into account for this matter: Liking and disliking.

Research

In the last sprint we agree that it might be a good idea to have some prior research about task that will be implemented in the next sprint. We only had 4 weeks remaining to finish the project and the implementation of the chat and how to create a chat room will be in one of the coming sprints.

We found an open project on git-hub, about how a chat is displayed.[111]. The project itself does not have any connection with any database, but offers a overview how we can display the messages and how a nine patch[112] image is being used to wrap a message in a mobile application.

Implementation

The liking action has to be saved from both ways, so we have created the methods addLiked (see figure 6.39a) and addExternalLikes, which is very similar to addLiked. The role of the two are as follows. The addLiked method stores the id of the user in the list of strings called 'liked'. It then checks if the profile likes the user back in the list of strings called externalLikes. Whereas addExternalLikes is resposible for storing the id's of the profiles that like the user in externalLikes. For both methods, if there is like from both sides, the match is created through the method addMatch (see figure 6.39b).

The disliking action also has its own method, namely addDislike, which either stores the id of the disliked person to the list of strings called dislike or removes a match, if the user's id is saved in the list of strings that saves matches, suggestively called match.

Just like any other model, the class Match is used by other classes for sending the information to our database.

```
public void addLiked(String elementID) {
    if (!liked.contains(elementID))
        liked.add(elementID);
    if (externalLikes.contains(elementID))
        addMatch(elementID);

}
```

(a) addLiked

```
private void addMatch(String elementID) {
    if (!match.contains(elementID))
        match.add(elementID);
}
```

(b) addMatch

Figure 6.39: Methods from Match class

For the tasks regarding the map, the easiest way of deciding when to show the location of the address on the map was to include the map into a separate layout, called layoutMap, which initially has its visibility set to gone (see figure 6.40a). Afterwards, there is an if statement checking if the search bar above the map has any real location selected (see figure 6.40b) and if so, then the system tries to set layoutMap's visibility to visible and to update the map by passing the latitude and longitude of the location, all through a try and catch block.

```
try {
    layoutMap.setVisibility( View.GONE );
} catch ( Exception e ) {
    Log.d( TAG,  msg: "layout visible" );
}
```

(a) Setting layoutMap' visibility to gone

```
if ( autocompleteFragment != null ) {
    autocompleteFragment.setPlaceFields( Arrays.asList( Place.Field.LAT_LNG, Place.Field.NAME ) );
    autocompleteFragment.setOnPlaceSelectedListener( new PlaceSelectionListener( ) {
        @Override
        public void onPlaceSelected( @NonNull Place place ) {
            mAddressName = place.getName( );
            Log.i( TAG,  msg: "Place: " + place.getName( ) + ", " + place.getLatLng( ) );

            if ( wifi.isConnected( ) ) {
                try {
                    layoutMap.setVisibility( View.VISIBLE );
                    mapFragment.updateSite( place.getLatLng( ) );
                } catch ( Exception e ) {
                    layoutMap.setVisibility( View.GONE );
                }
            }
            mAddressID = place.getId( );

            mAddressLatitude = Objects.requireNonNull( place.getLatLng( ) ).latitude;
            mAddressLongitude = place.getLatLng( ).longitude;
        }
```

(b) Showing the map

Figure 6.40: The map in RoomCreationFragment class

After some research, we have decided to implement a bottom sheet[113] from Material Design, for the photo picking problem. As seen in the chapter on Material Design, the bottom Sheets are interactive surfaces that displays supplementary content, which in our case presents a menu of applications, organized in two lists. The top list contains the gallery of the phone and other such applications, like Google Photos, while the bottom list contains the main camera app of the phone and other camera-like applications.

The method "getPickImageIntent" (see figure 6.41) creates the bottom sheet with the applications to choose from. The way it works is it creates a chooser intent, which displays lists with all available external applications camera-like/gallery-like of the device[114].

The system gets the picture from the chosen intent through the method getImageFromResult (see figure 6.42). After checking with the boolean variable "isCamera" the source of the picture, the method then resizes, rotates and eventually returns the picture in Bitmap format.

```
public static Intent getPickImageIntent(Context context) {
    if (Build.VERSION.SDK_INT >= 24) {
        try {
            Method m = StrictMode.class.getMethod( name: "disableDeathOnFileUriExposure");
            m.invoke( obj: null);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    Intent chooserIntent = null;
    List<Intent> intentList = new ArrayList<>();
    Intent pickIntent = new Intent(Intent.ACTION_PICK,
            android.provider.MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
    Intent takePhotoIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    takePhotoIntent.putExtra( name: "return-data",  value: true);
    takePhotoIntent.putExtra(MediaStore.EXTRA_OUTPUT, Uri.fromFile(getTempFile(context)));
    intentList = addIntentsToList(context, intentList, pickIntent);
    intentList = addIntentsToList(context, intentList, takePhotoIntent);

    if (intentList.size() > 0) {
        chooserIntent = Intent.createChooser(intentList.remove( index: intentList.size() - 1),
                "Add photo");
        chooserIntent.putExtra(Intent.EXTRA_INITIAL_INTENTS, intentList.toArray(new Parcelable[]{}));
    }

    return chooserIntent;
}
```

Figure 6.41: Method getPickImageIntent from class ImagePicker

```
public static Bitmap getImageFromResult(Context context, int resultCode,
                                        Intent imageReturnedIntent) throws IOException {
    Log.d(TAG,  msg: "getImageFromResult, resultCode: " + resultCode);
    Bitmap bm = null;
    File imageFile = getTempFile(context);
    if (resultCode == Activity.RESULT_OK) {
        Uri selectedImage;
        boolean isCamera = (imageReturnedIntent == null ||
                imageReturnedIntent.getData() == null ||
                imageReturnedIntent.getData().toString().contains(imageFile.toString()));
        if (isCamera) {     //camera

            selectedImage = Uri.fromFile(imageFile);
        } else {            //gallery
            selectedImage = imageReturnedIntent.getData();
        }
        Log.d(TAG,  msg: "selectedImage: " + selectedImage);

        bm = getImageResized(context, selectedImage);
        int rotation = getRotation(context, selectedImage, isCamera);
        if (bm.getHeight() >= bm.getWidth()) {
            if (isCamera) {
                bm = rotate(bm,  rotation: rotation + 180);
            } else bm = rotate(bm, rotation);
        } else {

            bm = rotate(bm,  rotation: rotation + 90);
        }


    }
    return bm;
}
```

Figure 6.42: Method getImageFromResult from class ImagePicker

Review

For this sprint we have encountered two main problems. The first one is referred to the map in RoomCreationFragment, which had a magnifying glass icon that we did not find a solution for. Because we could not afford to spend anymore time with this issue, we decided to leave it as it was. This was also the case for the pictures coming from the camera intent. We could not find a way for checking which way the picture is rotated when first getting it from the camera and therefore, we did not know which way to rotate it again. Each emulator and phone's camera would act completely different, this being the reason for giving up on this problem.

While reviewing this sprint, we all agreed that we should not spend too much time on the not so important things that we cannot solve. There were many other features which were crucial to implement. This decision was also made, based on the fact that we still had other problems to solve from previous sprints and on the two problems mentioned above.

Sprint 7

Introduction

At the sprint number 7, the application is missing few functionalities. The ones that we implemented between 24th of April and 3rd of May were:

- Chat Selection screen where the users could open a new chat with another matched user.

- Create a chat room where the users can chat with a match, also the chat rooms should display all the old messages.
- Create a navigation bar for the interaction activity so the user can move between profile settings, match screen and chats.
- When a landlords swipe in the match screen, they should be able to choose from which of their rooms they are liking a tenant, and also when they are in a chat selection screen they should be able to choose the matches corresponding to their room. This will be implemented with a TabLayout.

Research

Regarding to the last two tasks, we followed the Android Material designs best practices. As seen in chapter 6.3 on Material Design, the navigation bar usually contains three to five icons and each of them represent a destination inside the application.

Figure 6.43: Bottom Navigation Bar

For our case, the Bottom Navigation Bar (see 6.43) contains three buttons: First to the left, brings the user to their profile, the middle one to SwipeFragment and the right one to the ChatSelectionFragment. The icons are suggestive, offered by Material Design, and below each of them, there is one-word texts. The Navigation Bar displays the three shades of our main colour as follows.The container is salmon coloured, the inactive buttons display the brighter shade and the active button is white.

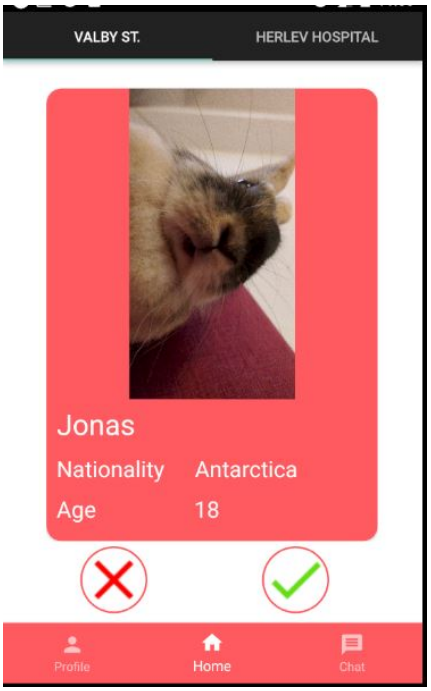


Figure 6.44: TabLayout allows to choose different rooms for the Landlord.

Tablayout was the chosen tool for the landlord to change the selected room. The Tablayout offers the best usage when the user has to change the view of the application, when the views are related and have the

same hierarchy[115]. The landlord can have up two 3 rooms in the application in the same time, and each of them gives the opportunity to the landlord to match with new tenants. Each of the rooms might have different tenant requirements, so we let the landlord decide match with tenant for individual rooms.

Implementation

Androids embeds the TabLayout in their API and can be created easily without any external library or project. In our case we have to decide in execution time how many tabs will be created and what is their name. In the SwipingFragment we are getting the landlords rooms as an argument and they can be uses when we create the instance of the TabLayout as seen in figure 6.45.

```
if ( ProfileSingleton.getInstance( ).getRole( ).equals( "Landlord" ) && getArguments( ) != null ) {
    mTenantProfiles = getArguments( ).getParcelableArrayList( KEY_TENANT_LIST );
    mLandlordsRooms = getArguments( ).getParcelableArrayList( KEY_ROOM_LIST_LANDLORD );
    List<String> rooms = getRoomsStrings( );
    mCurrentSelectedRoom = mLandlordsRooms.get( 0 );

    for ( String s : rooms ) {
        tabLayout.addTab( tabLayout.newTab( ).setText( s ) );
        mLandlordsAdapter.add( new ListAdapter( new ArrayList<>( mTenantProfiles ),
                roomPosted: null,  allRooms: null, mThisProfile ) );
    }
    mCurrentAdapter = mLandlordsAdapter.get( 0 );
    tabLayout.setTabGravity( TabLayout.GRAVITY_FILL );
    tabLayout.addOnTabSelectedListener( new TabLayout.OnTabSelectedListener( ) {
        @Override
        public void onTabSelected( TabLayout.Tab tab ) {
            mCurrentSelectedRoom = mLandlordsRooms.get( tab.getPosition( ) );
            //mCurrentAdapter = mLandlordsAdapter.get(tab.getPosition());
            mRecyclerView.swapAdapter( mLandlordsAdapter.get( tab.getPosition( ) ),
                    removeAndRecycleExistingViews: true );
        }
```

Figure 6.45: TabLayout implementation on SwipeFragment

The tenants does not need this Tablayout. The Swiping fragment has the same layout for both tenants and landlord, so we decided to hide the TabLayout for the tenant as seen in figure 6.46

```
if ( ProfileSingleton.getInstance( ).getRole( ).equals( "Tenant" )
        && getArguments( ) != null ) {
    mAllRooms = getArguments( ).getParcelableArrayList( KEY_ROOM_LIST_ALL );
    tabLayout.setVisibility( View.GONE );
    mCurrentAdapter = new ListAdapter(  tenantList: null, mLandlordsRooms,
            mAllRooms, mThisProfile );
}
```

Figure 6.46: Tablayout been hidden for Tenants.

Before creating a chat screen, we had to create a chat selection screen where each of the user should have a list of their matches to get to their corresponding chat room. We allow communication between any user that have matched with any other user, and each user has its own user ID and the corresponding user ID to their match.

Having this in mind, we need to create a bucket for each of the pair communication possible, and this buckets should be easy to identify in the future and should be unique for both pair of users that are involve in the chat. To obtain this, we decided to identify a chat conversation using both a tenant and a Room ID, and to make them unique and easy to find in the future we will concatenate them having the smallest string first. The implementation of creating the chatID can be seen in Figure 6.47

```
private String createChatID( String ID_one, String ID_two ) {
    String chat_ID;
    if ( ID_one.compareTo( ID_two ) < 0 )
        chat_ID = ID_one + "_" + ID_two;
    else
        chat_ID = ID_two + "_" + ID_one;
    return chat_ID;
}
```

Figure 6.47: How WeRoom identify each of the chat rooms, creating a unique chatID for each of the pairs: Tenant and Rooms.

Now that all chats have a unique way to identify, we had to display the possible chat rooms to the users. If the user is a tenant we will get the matches from the Match object in their profile using the ProfileSingleton. If the role is a landlord we will display each of the matches of his/her corresponding room. In this case we also implement a TabLayout explained in the SwipingFragment, giving the landlord a clear view which chat correspond to each room.

We display this information using a RecyclerView. The RecyclerView has two different holders, one in case that we display a room, and another one in case that we display a tenant. Each of the holders will have a ClickListener that will change the current fragment to a ChatFragment. When calling this new ChatFragment it passes the ChatID so it can start listening to the right bucket in the Real-Time database.

The ChatFragment has two major tasks to perform. One will be listening to the bucket that corresponds to the chat in the database if any new message has been written. The second one, will be writting new message to the database in the corresponding bucket as seen in Figure 6.48

```
public void newMessageListener( ) {
    DatabaseReference messageRef = FirebaseDatabase.getInstance( )
            .getReference( DataBasePath.CHAT.getValue( ) ).child( mChatID );
    messageRef.addChildEventListener( new ChildEventListener( ) {
        @Override
        public void onChildAdded( @NonNull DataSnapshot dataSnapshot,
                                  @Nullable String s ) {
            Message message = dataSnapshot.getValue( Message.class );
            chatMessages.add( message );
            adapter.notifyDataSetChanged( );
        }
```

Figure 6.48: ChatRoom listening to new message from Real-Time database.

ChatFragment display all messages in a ListView.All the messages displayed need be added to a message adapter. The message adapter reads all the message objects and checks if the message senders ID matches with the current profile. This way we can distinguish them from incoming and outgoing messages. The only difference between incoming and outgoing messages is that they use a different layout when displaying

the message. The incoming messages will be displayed at the left in grey color, and the outgoing messages will be oriented at the right in a salmon color.

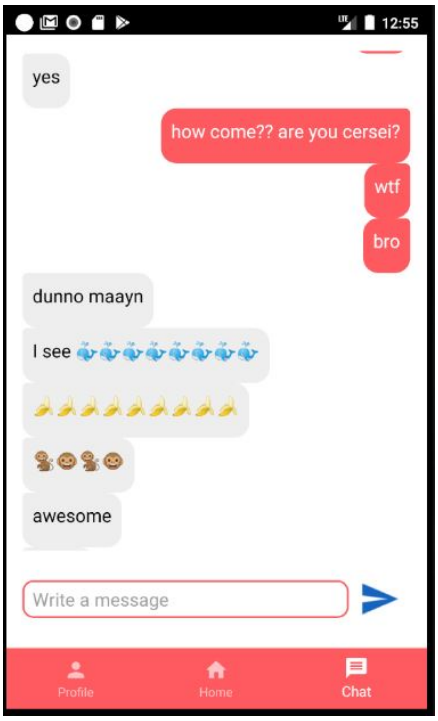


Figure 6.49: Chat between a Tenant and a Landlord.

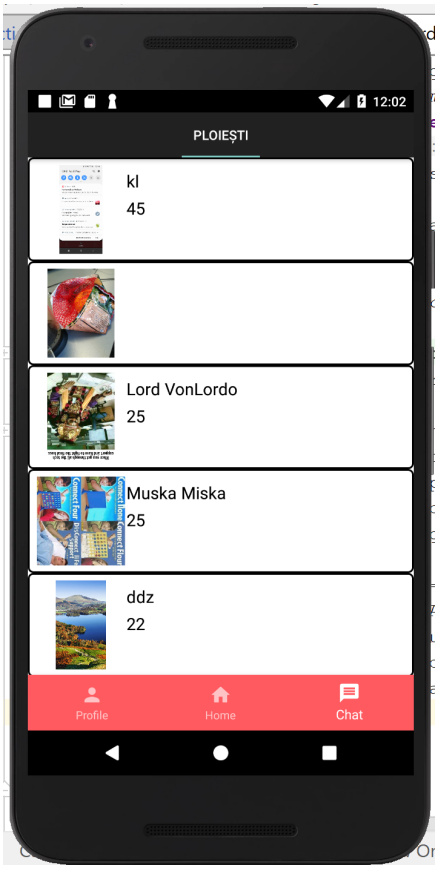


Figure 6.50: ListView of users that can be chatted with

Review

Finalizing the sprint, we decided to focus less on having a market ready product with perfect layouts. In the last semester meeting with our supervisor, we got some general feedback that we should show we can program in android and manage a project from the visualization of the idea, the design, implementation and project management.

We had only 3 more weeks from here before the hand in deadline. Some of the implementation done in the next sprint had to be prioritized, and some of the implementation will not be finished completely (for example not binding the entire information of the Profile in the screen).

Sprint 8

Introduction

The team decided that the sprint 8 should finish all the implementation of the application so we could have a week for testing the application, and a last sprint before the handing to fix any encountered bugs or improvement found in the user acceptance test.

During the period between the 3rd of May and the 10th of May we analyzed all the missing functionalities, and after cutting out some of the functional requirements we implemented the next tasks:

- Profile information and settings: The user need to be able to see the information regarding to their profile, and edit any of their profiles (tenant, landlord and rooms are also included). The Settings the user needs to be able to logout from the application and delete their account if interested. Last, the user can choose to deactivate the notifications from the application when they get a new match.
- The application needs to create a notification when they have a new match.
- The landlord needs to be able to enable the a lazy swiper mode for their room. If this is activated any tenant that liked their room will automatically create a match.
- If the WiFi is not enabled, the application won't show the map fragment at the room creation fragment.
- The TabLayout from the swiping fragment is missing the logic. Create a different adapter with the tenant for each of the rooms and change the adapter on execution time when the landlord changes to another room.
- All the Tenant and Rooms that are been read from the database need to be filtered regarding to the Tenant or Landlord's requirement that is loaded at the application.

Some of the requirement were excluded in the application. All the functionalities that were taken out, are extra functionalities that would make the application more appealing, but wont affect any of the main functionalities of the application.

- WeRoom wont have any review system for tenant and landlords.
- The tenant and landlords wont be able to unmatch.
- The user wont be able to switch the role from landlord to tenant and vice versa.
- The application wont create any notification when the user has a new message.

Research

One of the core functionalities of the application is to filters the users that the tenants and landlords are available to match with. Until now when a user was matching on our application we where displaying all the possible candidates in the screen. This behavior had to change and we had to filters the candidates regarding to the users profile settings.

The initial plan of the implementation was to filter the candidates when making a query on the databases. We had a small discussion in the sprint 5, about Firebase databases query limitations. FireCloud and Real-Time Database doesn't give us the opportunity to create a single query that check if multiple fields are within a given range.

The proposed solution was to handle the filtering logic on the phone. We were aware that this is not the best implementation, since we can overload the application with logic that should be handled by a back-end server. To make sure that this solution would create a scalable application we introduced a non-functional requirement that the complexity of the filtering should not be bigger than O(NlogN). We decided to have a upper limit on O(NlogN), since implementing a filtering for all existing users from the database on O(N^2) complexity would increase the time required to show the users on the phone drastically when the databases had big amount of users.

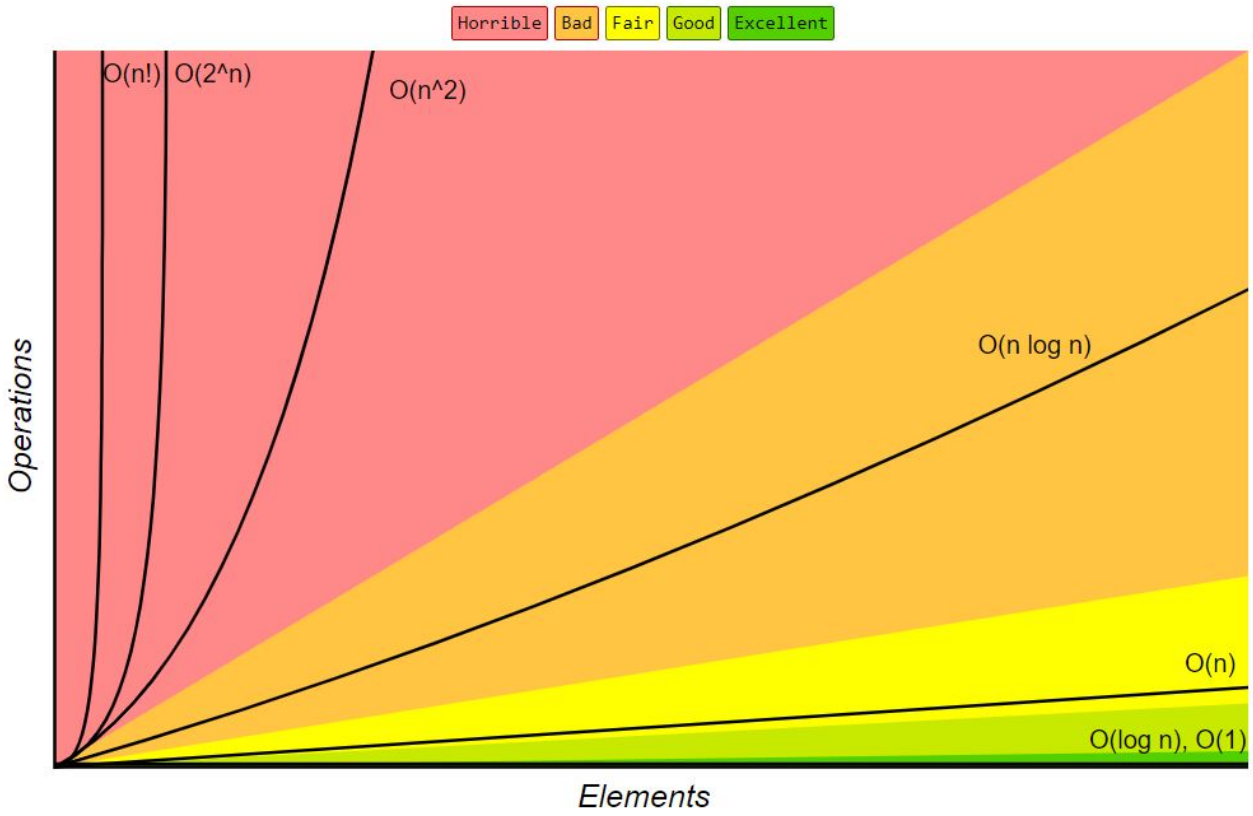


Figure 6.51: Big - O complexity chart
Source: Big O notation illustration and sheet: http://bigocheatsheet.com/

Implementation

Candidate filtering will have two different filters. One in case of tenant users (they will filter rooms) and another one for landlords users (they will filter the tenants profiles).

We will explain the implementation behind the tenants view filtering. The tenants should get a list of room candidates, and filter them by rent, deposit, location, if smoking is allowed, if internet is included etc. We found that on our development environment if we introduced all this filtering it would be really hard to create data to verify if the application was working. Furthermore, if we had to filter all the information that both roles had set up in their account it would take long time to implement. We decided that for now on we would lower the filter requirements and filter the rooms only by rent, deposit and location.

The filter get a list of Rooms and discard all rooms that are outside of the tenants rent, deposit and location range. Rent and deposit filtering is trivial, we can just verify the rooms rent and deposit between the maximum and minimum values given by the tenant.

The location of the room and the desired location of the tenant are stored in coordinates. Using the Haversine formula [116] we can check the distance between those two coordinates and check if its inside the radius distance where the tenant wants to live.

```
public static ArrayList<RoomPosted> filterRoomsFromTenant( Profile p, ArrayList<RoomPosted> rooms ) {
    if ( p.getTenant( ) == null ) {
        return null;
    }
    ArrayList<RoomPosted> result = new ArrayList<>( rooms );
    ArrayList<RoomPosted> removeRooms = new ArrayList<>( );
    TenantProfile tenant = p.getTenant( );

    for ( RoomPosted r : result ) {
        if ( HaversineCalculator.distance( tenant.getCityLatitude( ), tenant.getCityLongitude( ),
                r.getLatitude( ), r.getLongitude( ) ) > tenant.getDistanceCenter( ) ) {
            removeRooms.add( r );
            continue;
        }
        if ( tenant.getMinDeposit( ) > r.getDeposit( ) || tenant.getMAxDeposit( ) < r.getDeposit( ) ) {
            removeRooms.add( r );
            continue;
        }
        if ( tenant.getMinRent( ) > r.getRent( ) || tenant.getMaxRent( ) < r.getRent( ) ) {
            removeRooms.add( r );
        }
    }
    //TODO implement all other filters in future, for testing it will only filter
    //distance, deposit and rent.
    result.removeAll( removeRooms );
    return result;
}
```

Figure 6.52: Implementation of filterRoomsFromTenant.

In this filters we managed to get a O(N) complexity, having a better performance than the O(NlogN) set in the non-functional requirements. Also note that the filters are not discarding the users that has been already swiped in previous sessions. The filters in the future should check if the room ID has is contained in any of the list of the Profiles Match object. If we find the room ID on Match, Liked or Dislike we should remove the room from the list as the tenant has already interacted with that room.

Modifying any existing profile is similar to create a new profile. In all cases but modifying a room we re-used the existing fragments from the onboarding package.

```
if ( getArguments( ) != null && getArguments( )
        .getBoolean( KEY_IS_EDIT ) )
    setProfile( ProfileSingleton.getInstance( ) );
return v;
```

Figure 6.53: Modify the behavior of ProfileFragment to edit a profile instead of creating a new one.

Therefore, LandlordProfileFragment, ProfileTenantFragment and RoomCreationFragment were modified from the original implementation. All three cases in the method 'onCreateView' they will check if any argument were passed when creating the fragment. If there was an argument it would be a profile that the fragment had to display in the text field and modify the check boxes regarding to the information obtained from the argument (See Figure 6.53). Also it will create a flag to modify the flow the profile fragments, so it would return to the ProfileSettings instead of continuing the on boarding flow (see Figure 6.54)

```
if ( mEditable ) {
    ( ( InteractionActivity ) Objects.requireNonNull( getActivity( ) ) )
            .changeToPorifleFragment( );
} else {
    if ( mRole.getSelectedItemId( ) == 0 ) {
        changeFragment( new LandlordProfileFragment( ) );
    } else if ( mRole.getSelectedItemId( ) == 1 ) {
        changeFragment( new ProfileTenantFragment( ) );
    }
}
```

Figure 6.54: Modifying the flow of the ProfileFragment when we edit a Profile.

To create notification on new matches, we had to listen to any changes on the database. If the role of the user was a Tenant we had to listen to any new match created in their profile, but if the role was a Landlord, we had to create up to three listeners for each of their rooms.

The best approach to create those listeners, and having them working independently from current fragments or activities, we used Android services[117]. The services are used to handle long-running operations that doesn't need any user interface. The service will be created to be listening to any changes on the database and when any change has been detected on the matches it will create a notification for the user, letting them know that a new match have been created.

The NotificationService is been initialized once we are in the InteractiveActivity (it does not make any sense to initialize it before as we are not able to have any match without a finished profile). When the service is created and executed, it will read the argument from the intent and will initialize the listener to detect any changes on the database as seen in the Figure 6.55.

```
DocumentReference docRef = db.collection( DataBasePath.USERS.getValue( ) )
        .document( mIds.get( 0 ) );
mRegistration.add( docRef.addSnapshotListener( (documentSnapshot, e) → {
        if ( documentSnapshot != null ) {
            Profile p = documentSnapshot.toObject( Profile.class );
            Match newMatch;
            if ( p != null ) {
                newMatch = p.getMatch( );
                newMatch.getMatch( ).removeAll( mMatches.get( 0 ).getMatch( ) );
                for ( String s : newMatch.getMatch( ) ) {
                    mMatches.get( 0 ).addLiked( s );
                    mMatches.get( 0 ).addExternalLikes( s );
                    createNotification( );
                }
            }
        }
```

Figure 6.55: Database Listener to new matches for the Tenant role.

Once the listeners detect that a new match has been added to the database, it will trigger a new notification. To be able to create a new notification we have to add a minimum information to the notification before sending it to the user[118]:

1. Set a small icon for the notification. We will show the notification with the application logo.
2. Set a title for the notification.

3. Set a content text in the notification.

4. Set a notification channel for the notification.

The implementation for the lazy swiper for the tenant rooms will be done modifying the match model. We will have a boolean that will state if the match is lazy or not. By default it will be false, but when we change its value to true we will change the behavior of addExternalLikes method so it creates a match in the model without the need of creation any like from the Room's Match as seen in Figure 6.56.

```
public boolean addExternalLikes( String elementID ) {
    if ( ! externalLikes.contains( elementID ) )
        externalLikes.add( elementID );
    if ( lazySwipe ) {
        addMatch( elementID );
    } else if ( liked.contains( elementID ) )
        addMatch( elementID );
    return lazySwipe;
}
```

Figure 6.56: New implementation of addExternalLikes to behave in case that the match is on lazy mode.

Also notice, that we are returning in the method addExternalLikes the value of lazySwipe. This now is usefull to let the tenant know if they have liked a lazy swiper so they have to create the match in their own Match model as seen in Figute 6.57.

```
if ( room.getMatch( ).addExternalLikes( mThisProfile.getUserID( ) ) )
    mThisProfile.getMatch( ).addExternalLikes( mCurrentAdapter.returnTopItemID( ) );
```

Figure 6.57: How the tenant identify that have liked a lazy swiper Tenant.

One of the program requirement for the project was to disable a functionality if the application was not connected to WiFi. We could not find any functionality in our application that could follow this requirement, therefore, we forced the application to not show the map when creating a new room, if the application was not connected to WiFi.

First step is to declare in the application manifest that its need permission to the Network_State. This will enable the application to check on execution time if the user is connected in WiFi or not. We can see in the Figure 6.58 that we will display the map on the fragment only if the WiFi is connected.

```
ConnectivityManager connManager = ( ConnectivityManager ) Objects.requireNonNull( getActivity( ) )
        .getSystemService( Context.CONNECTIVITY_SERVICE );
final NetworkInfo wifi = connManager.getNetworkInfo( ConnectivityManager.TYPE_WIFI );

if ( wifi.isConnected( ) ) {
    mapFragment = new MapFragment( );
    final FragmentTransaction transaction = getChildFragmentManager( ).beginTransaction( );
    transaction.replace( R.id.showmapfragment, mapFragment ).commit( );
}
```

Figure 6.58: Opening the map fragment, only if the WiFi is connected

Review

In the sprint review we recognized that we had too many task to implement in the sprint, and some of the quality of the code had declined compared to the previous sprint. We intended to have a look and clean up the code in the next sprint. Regarding to the implementation, only a few layouts were missing to be able to make a user acceptance test.

Sprint 9

Introduction

For the ninth sprint, we decided to perform testings on the system, in order to get feedback from potential users and also to check if everything works as expected. For this matter, we chose to use the following tests:

- User Acceptance testing
- Black Box
- Monkey Testing
- White Box

The reason for having these tests is to find and resolve as many bugs and errors as possible, to validate the functional and respectively non-functional requirements and to have the system verified by users. This sprint had an extended period, from the 10th of May to the 20th of May, because, as expected, we found a variety of faults in the code.

Research

White box testing techniques analyze the internal structures the used data structures, internal design, code structure and the working of the software, rather than just the functionality as in black box testing[119]. White box testing helped us identify possible bugs and improve the durability of the software.

As mentioned in the Methodology chapter, we are using the Monkey Run tool for stress-testing our application[120]. The reason for this is finding as many bugs and mistakes in the code as possible, so that they can be solved.

The way of using this method is by introducing a one-line command in Command Prompt, afterwards the Monkey Testing displays the exceptions found and then stops due to finding an error, if it has not been ordered otherwise.

Testing

We began with testings performed by users on our application. What we did was to write down all the functional requirements from the MOSCOW and put them into an Excel spreadsheet, so we can check if they are fulfilled or not. To help us with this, we used the agreement we have with another group from our study line. They were given two phones and told to create an account as a tenant and another should create an account as a landlord. They then had to set up their accounts and try to match and chat with each other. They were also told to try and edit their account information. By doing all of these, we used both the Black Box testing for input-output testing and the User Acceptance testing for checking the requirements[121].

The group found both errors we were aware about, along some new errors. Through the new ones we discovered that after deleting the account, the user could not open the application anymore. Other

mistakes in the code were regarding the like and dislike button, which sometimes worked and sometimes crashed the application, the slow loading of pictures form the database, the filters switched to "does not matter" in the profile editing page and so on. As for the ones we were aware about, the application crashed when the magnifying glass icon near the map was pressed in the room creation page and the pictures uploaded wither with camera or from the gallery were rotated by different degrees.

While testing the application, we also marked on our spreadsheet with true/false at every functional requirement, so that we have an overview of whether they are fulfilled. All the requirements from Must and Should in the MOSCOW are met, along with most of the requirements in the Could. The checked spreadsheet can be found in the annex.
The next tool we used was the Monkey Run for testing the system from the perspective of a automatic user performing pseudo-randomly generated actions. For this purpose, we put all of the screens to a test, using 20.000 events for each of them.

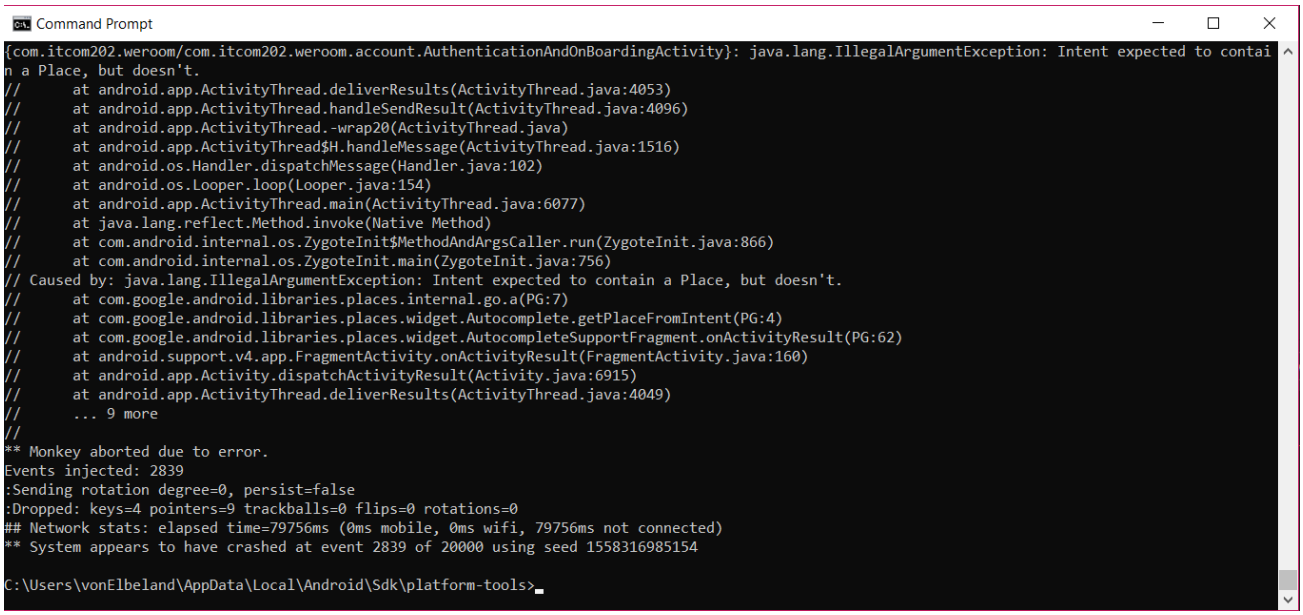


Figure 6.59: Monkey Testing screenshot

As can be seen in figure 6.59, the test found only one exception regarding the search bar for Places in RoomCreationFragment class and then stopped after performing 2839 events out of 20000. Therefore, this test proved that the Builders work properly and the application does not crash from wrong input.

External Actor Review

The improvements the testers requested were to display some kind of text when they run out of users to swipe. Since we were not displaying anything, they were led into thinking the application could not load the rest of the users' profiles. Other modifications they suggested were regarding to the chat. In the conversation page, the messages do not display the name of the sender and neither does the name appear in the top of page, as it was supposed to be.

Review

After using the three mentioned tools, we organized a long list of bugs and errors found during the testing. The work on fixing the errors was continued in the next sprint, along with the White Box testing for which we did not have time to do.

Sprint 10

Introduction

Sprint number 10 was the final one for the project. The start of the sprint was on the 20th of May and it ended on the 26th of May. This sprint was dedicated to testing our application and fixing bugs.

Testing results

At the end of the previous sprint we had performed Black box testing from which we found out several bugs to be fixed.

1. Google login crashes the app
2. In the chat should be added name of the user you are chatting with
3. Activating a lazy swiper setting crashes the app
4. Deleting the account crashes the app and cannot be reopened

Those were the major identified bugs that we decided to fix during the last sprint. Additionally, the problem with reversing the photos to 180 degrees was pointed out, but as discussed previously, further implementation won't be done for that.

Moreover, we have performed White box testing and we went through the Non-functional requirements and we found out the following to be done and tested:

1. The application will show a GDPR consent to the users before creation the account
2. When out of matches, the user should be informed
3. The application must be running 24/7, and in case of a system shutdown for maintenance,need to be informed to the users beforehand
4. The user should not wait more 5 seconds server response time to login, having at least 80% of the Wi-Fi
5. If the user has some tenant or rooms to be displayed in the match process, the response time to load the first tenant or room cannot exceed 3 seconds, while having at least 80% of the Wifi connection
6. The size of the application should not be bigger than 40MB.
7. The user must not download from the database any picture bigger than 8MB

Implementation

We first started fixing the bugs found from doing the Black Box testing. As mentioned, when we tried to log-in with a Google account the app was crashing. We found that this problem is only happening when the application was killed and started again. After debugging, we found the reason. When calling the Google Lock function we get an authentication token from the callback. After that, this token is used by Firebase Authentication. However, at the same time we are executing a method to start the activity, which is getting the user ID from Firebase Authentication. We found that the callback with token from the Google Lock function and the query for the user ID are asynchronous tasks. Therefore, to fix that we have put an OnCompleteListener for the Google Token.(see fig.6.60) This way we can continue to the log-in method only if we had received the Google Lock Tocken.

```
@Override
public void onActivityResult( int requestCode, int resultCode, Intent data ) {
    super.onActivityResult( requestCode, resultCode, data );
    //Callback for Facebook intent.
    FaceBookConnection.getCallBackManager( ).onActivityResult( requestCode, resultCode, data );
    // Result returned from launching the Intent from GoogleSignInApi.getSignInIntent(...);
    if ( requestCode == GoogleConnection.RC_SIGN_IN ) {
        Task<GoogleSignInAccount> task = GoogleSignIn.getSignedInAccountFromIntent( data );
        try {
            // Google Sign In was successful, authenticate with Firebase
            GoogleSignInAccount account = task.getResult( ApiException.class );
            if (account != null) {
                GoogleConnection.firebaseAuthWithGoogle( account, getActivity( ), firebaseAuth,
                        getActivity( ), fragment: this ).addOnCompleteListener((task) → {
                        logUser( );
                });
```

Figure 6.60: login function is started on completion of the Google Lock

After that we fixed the recommendations we got from the testing. We showed the name of the user you are chatting with in the ChatFragment and included a back button redirecting the user to the SelectChatFargment. The View was implemented by using a CardView, following the Material Design[122].

Next bug found by testing was regarding the Lazy Swiper function for the Landlord. As we have mentioned, the landlord will have the possibility to activate the LazySwiper after posting a room, from the setting for the room. However, in the setting of the room we use TabLayout from where the Landlord can post another room if it is not exceeding the permited amount. In the layout for EditRoomFragment the switch button for activation was visible for both the room that was already posted and for the create new room. Therefore, when activating the LazySwiper for the new room, we were having a NullPointer exception, since such a room does not exist yet. The way to fix this was to set the visibility to gone of the LazySwiper setting from the layout if the room has not been initialized yet. The method for initializing room is putting the values of the room passed as an argument in a bundle. If there are no arguments, then the room has not been initialized and it does not exist. (see fig.6.61)

```
if ( getArguments( ) != null ) {
    mThisPostedRoom = getArguments( ).getParcelable( KEY_ROOM );
    initializeRoom( );
    initializePictures( );
    mLazySwiperLinear.setVisibility(View.VISIBLE);
} else {
    mEditRoom.setText( "Confirm" );
    mDeleteRoom.setVisibility( View.GONE );
    mLazySwiperLinear.setVisibility(View.GONE);
```

Figure 6.61: LazySwiper Option Setting visibility depending if room exists or not

Afterwards, the last thing to fix from the testing was to be able to delete the account. After the testers tried to delete thir account, they could not reopen the application. This was occurring because we were trying to delete from the database all the rooms related to the user, even though the user could be a tenant and not have any rooms. Hence, before performing the task for deleting the document for each room, we should check the role of the current user and only if the role equals to "Landlord", execute the task.

The next step was to start fulfilling the last non-functional requirements. The first one was about the GDPR consent of the user when crating an account. To fulfill that, we put a link at the Sign Up screen, redirecting to the Privacu Policy of WeRoom. The policy was free and elaborated by website[123], because security is not a requirement nor a focus of the project.

The next requirement was to inform the user when out of matches. That was quickly fixed, as it had to add a TextView when out of Cards in InteractionActivity. Those two requirements were left on purpose for the last sprint as we wanted to be sure we had all the necessary and essential implementation for the application to be working.

We make sure the app is runnning 24/7 because we are using Firebase and Firestore cloud database. Therefore, if there is a shutdown of those, there must be provided information from the providers.

Setting the succeeding requirements as fulfilled, we should test our application. We tested the response time for log-in a user and the loading time for the swipeable cards. Primarily, we have put as part of the requirement that the Wi-Fi connection should be at least 80%, but when starting to make the testing, we found out that we have no power over the strength of the connection. Therefore, the tests have been made without fulfilling this prerequisite.

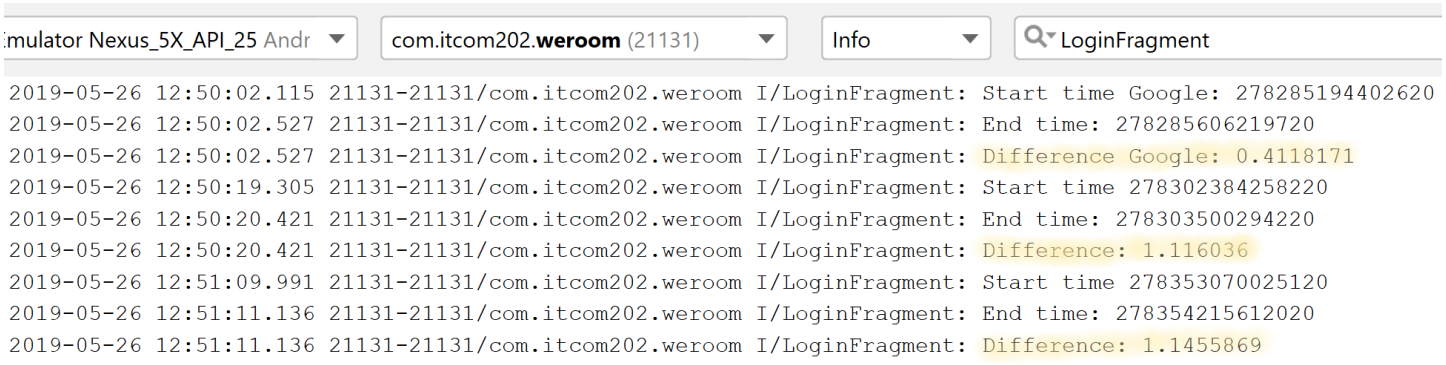


Figure 6.62: Response time for login

In fig.6.62 can be seen results after tests. The actual response time for log-in is a bit more than a second, when log-in with credentials, and half a second when using Google Lock function, log-in with token. This is fulfilling the requirement of having at most 5s waiting time for the user when log-in.

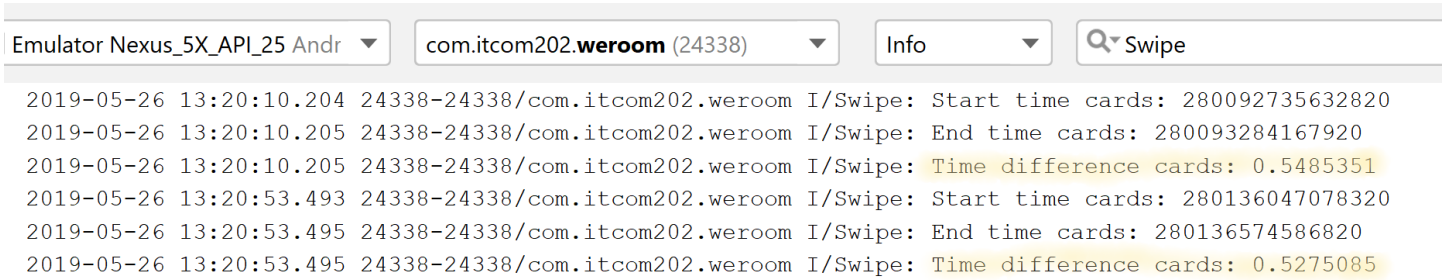


Figure 6.63: Loading time swipeable cards

In fig. 6.63 can be seen results from test for loading time for the swipeable cards. Test have been made for both tenant and landlord profiles. As it can be seen, it takes a bit more than 0.5 seconds for the user to see the loaded information on the cards. However, as we are querying all the users and rooms from the database(depending on the role of the current user), this loading time can exceed if the overall number of

users in the database increases. For the current state of the application, the results from the test fulfill the set requirement of at most 3 seconds loading time when starting SwipeFragment.

Figure 6.64: Size of WeRoom

The next requirement was about the total size of the application. This was checked at the end of the implementation. As seen in figure 6.64 the size is MB. The initial requirement was to be up to 40MB in total.

```
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read;
      allow write: if request.resource.size < 8 * 1024 * 1024;  }
  }
}
```

Figure 6.65: Restriction of image upload at Firebase Storage

The last non-functional requirement to check was concerning the pictures. We wanted to ensure the user cannot download any picture from the database bigger than 8MB. To make sure this is happening, we put a restriction in the rules of Firebase Storage, where we are storing the picture. The restriction is about the max possible upload size. This way, a picture bigger than 8MB will not be uploaded and stored. We put restriction for the input because this way we are making sure that no existing picture in the Storage will exceed this size and, hence, no picture bigger than that will be downloaded.

Last thing we did in the sprint is to prepare Unit tests for some of the methods in WeRoom. Not everything has been tested as not every method permitted this. We have made tests for the initialization of a profile, for the calculation of the distance between two addresses and for the creation of the unique ChatID.

Review

As this was the last sprint for the project we fulfilled the last non-functional requirements left and fixed bugs found out from Black box testing. We performed White box testing, in particular Unit testing. The group agreed that this technique should have been used from an earlier state of the project following the implementation flow.

6.6 Conclusion

The process of implementation of a software is very complex, as can be seen in this chapter. By explaining in detail how everything was implemented, there can be observed the process of learning and exploring Android, as well as bettering aspects regarding the organization of the project and the splitting of work, according to time and resources.

Chapter 7

Testing

This chapter presents a summary about testing the system with Acceptance Testing, White Box, Black Box and Monkey Run, as they have already been presented in the sprints. Moreover, there is an explanation on why we chose them and what the results were.

7.1 Acceptance testing

The Acceptance testing has been performed several times during the implementation. With the User Acceptance Testing we tried to get as much feedback as possible.
Ever since WeRoom was only a prototype, this testing was done with the aim to see if the flow and information displayed was good enough to start implementing. They thought the overall flow was good, but we were missing to put in some preferences the user could add to his/her account. After the testing we could start implementing this in our application.

After the implementation of the account creation, we did another acceptance testing with the same group, to see if we should correct or add information layout-wise. This round of feedback brought suggestions on confusing buttons and wrong input. They also encountered some bugs along the way, which we had to deal with later on.

Lastly, when the implementation part was done, we did a third acceptance testing with the same group. This was combined with the Black Box testing.

7.1.1 Black Box

As previously mentioned in the sprints, the Black Box is used for testing the system, regardless of the code and implementation, focusing on input and output only. The reason for choosing this method is because our application is meant to be used by people all over the world, so having the opportunity to observe a group of people verifying WeRoom has helped us find all the mistakes that otherwise would have remained undiscovered.

After we had our colleagues test it, they found a series of bugs, from which the majority crashed the application, like the error when deleting the account. Notes were taken of every problem they encountered, so we can later improve this. Afterwards, we asked them to suggest changes for WeRoom, which mainly were about the layout of the application, rather than implementing features.

Other results from the testing, brought various bugs to solve and an organized list of fulfilled and unfulfilled functional requirements, which can be found in the annex.

7.1.2 Monkey Testing

As mentioned in the Methodology chapter, we are using this tool for stress-testing our application. The reason for this is finding as many bugs and mistakes in the code as possible, so that they can be solved.

We used this tool mainly to check if all buttons work properly and there are no crashes caused by the Builders we had in all profile creation classes in order to avoid wrong input. After 20000 events performed on each screen, we only got one exception. As a result, we made sure we did not forget to check all information gathered from the users, such as name, age, pictures etc, so that they are realistic enough.

7.2 White Box

White box testing techniques analyze the internal structures the used data structures, internal design, code structure and the working of the software rather than just the functionality as in black box testing[124]. White box testing helped us identify possible bugs and improve the durability of the software.

During the implementation of WeRoom, we have been using a modified version of Kanban board for delegating tasks. The modification was an added section called "Verify". After each implementation of a functionality by a member of the group, another member was taking the task to verify the code. This approach improved the quality of the code in general, as we were checking each others work, making sure that everything was working properly and meeting all the requirements from the system modelling. Additionally, we were striving to write readable and clean code, so others can understand the functionalities faster.

Further more, one of the non-functional requirements for the development was that the app has no warnings by the IDE. In other words this would mean that all the methods/variables are used or access modifiers are changed to private if there is no need for them to be public, for example. Leaving warnings could result in instability of the software. We dealt only with the obnoxious warnings. Fulfilling this requirement ensures the quality of the code.

7.2.1 Unit testing

The unit test at WeRoom were done during the last sprint. However, the main idea of having unit test is to test different components behavior, and ensure that those behaviors doesn't change during new iterations. Because of this, unit testing should be made at early stages of the implementation.
On each sprint we should have created unit test for method that where critical to the application. Then, in future implementation after running the unit test before merging the code into the master branch we would make sure that the units functionality were unchanged.

Android can perform 2 different unit testing:

- **Local test:** Single test that runs on JVM (Java Virtual Machine), for faster performance than executing them on an Android emulator. If the test has any dependencies the test should be mocked [125].

- **Instrumented test:** If the test needs instrumentation information, such as the application Context. This test are performed on a Android emulator and are usually more complex tests that can't be tested on JVM.

The unit test that we created for WeRoom, are all local test. We covered only three units in our unit test: ProfileSingleTon isFinished method, Haversine distance calculation and CreateChat_ID method. The application should have more unit test coverage, and the units that we choose should had more test, covering more cases.

Identifying what unit should be tested can be difficult, and can end up creating an overloaded amount of unit testing. A good practice is to create a call graph[126] and cover the most used classes. As microsoft CEO states[127], 80% of the bugs are generated in the 20% of the code. This relation is also known as Pareto Principle [128].

Implementation of Unit Test

The main responsibility of a unit test is to call a method (this is the unit) with a specific value and test that the result is the one expected.

In some cases, those units that need to be tested belong to private or package-private classes, or the units itself are private. To be able to test them without changing the access modifiers of the implementation we can use Java Reflection feature[129]. This allow us to have access to any of the method or classes in the code, even if access modifiers define them as private.

Previously we mentioned that we did some unit test for createChatID method. This method creates a chat ID for any pair of profile and room given. The trick of this method is that the argument accept two strings, and return an ID string. To generate the same ID every time the methods put the lowest string and concatenates the biggest string afterward creating a unique chat ID since the Room and the Profile ID'S are also unique.

We would like to ensure that this always happens, independently on what we have implemented in the code this functionality is immutable. We perform a check with two strings "123" and "abc" and in both unit test check if the generated ID is "123_abc". By any chance if this changes the Unit test will notice the developer that something went wrong.

This method that we want to unit test is also private, and can not be access outside of the class. We can use Java reflection and get access to the private method as seen in the Figure 7.1

```
@Test
public void right_CreateChatID_One(){
    try {
        Class<?> c = SelectChatFragment.class;
        Object o = c.newInstance();

        Method method = c.getDeclaredMethod( name: "createChatID",
                String.class,String.class);
        method.setAccessible(true);
        assertEquals(method.invoke(o, ...args: "123","abc"), actual: "123_abc");

    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    }
}
```

Figure 7.1: Unit Test of createChatID using Java Reflection

Chapter 8

Requirement Verification

Following the testing, we could see which functional and non-functional requirements were done. The two tables below give a good overview of what has and has not been implemented. To verify the requirements, we went through each one and either checked or unchecked them, depending if they had been fulfilled or not.

Functional requirements

1	The user must be able to create an account/log-in using a Facebook/Google account or an Email address and a password.	✓
2	The user must be able to log out of his/her account.	✓
3	The user must be able to choose if he/she is creating a tenant/landlord profile.	✓
4	The landlord must be able to create profiles for at least one and up to three rooms.	✓
5	The landlord/tenant must be able to choose the filters that describe best the ideal tenant or the place to rent.	✓
6	The system must check if the information provided by the user is realistic.	✓
7	The landlord/tenant must be able to see only the tenants/rental properties that meet his/her requirements.	✓
8	The landlord/tenant must be able to swipe either left or right, depending on if he/she dislikes or respectively likes the tenant/property.	✓
9	The users must be able to see more information about a profile, when clicking on the swipe card.	✓
10	The tenant and landlord who swipe right to each other must be able to chat.	✓
11	The landlord must be able to switch between chat/swipe pages of the room profiles he/she owns.	✓
12	The user should be able to delete his/her account.	✓
13	The user should able to reset his/her password.	✓
14	The user should be able to describe himself/herself using tags.	✓
15	The system should show the address of a room on a map.	✓
16	The system should require users to post pictures both for his/her account and for rooms (in case of landlords).	✓
17	The system should require the landlord to post at least three and up to ten pictures of the room.	✓
18	The landlord should be able to choose to set his/her profile to no swipes, so that he/she would match with any tenant who swipes right.	✓
19	The system should allow the user to choose either to take a picture with the camera or to pick images from the gallery.	✓
20	The system should give suggestions to the addresses typed in the search bar above the map view.	✓
21	The user should be able to edit the information on his/her profile.	✓
22	The user could able to activate/deactivate his/her account.	×
23	The system could notify the user when he/she matches with another user.	✓
24	The landlord/tenant could be able reverse the swipe only once, immediately after he/she has swiped.	×
25	The user could be able to activate/deactivate his/her tenant profile or room profiles.	×

26	The system could notify the user when the user receives a message.	×
27	The system could notify the user if more swipes are available.	×
28	The user could enable/disable notifications.	✓
29	The user must be able to filter all tenant and room up to 50km from the desired city	✓
30	The user, when once created a room or tenant profile, must be able to modify those in the future.	✓
31	The user must be informed if there are no more tenants or room to be matched with	✓
32	Most of the personal information will not be required to be fill in the profiles, to have the application working	×
33	The user must be able to share a picture in the chat.	×

Non-functional requirements

1	The system must have a database to store information.	✓
2	The system must use Facebook and Gmail API for account creation.	✓
3	The device must have at least Android 4.4 OS version.	✓
4	Log-in connection between app and database should be encrypted.	✓
5	The application will show a GDPR consent to the users before creation the account, and theapplication will follow the GDPR legislation.	✓
6	The user should not wait more 5 seconds server response time to login, having at leats 80% of the Wi-Fi .	✓
7	The system should able to access both the camera and the gallery for adding pictures.	✓
8	The application should auto-fill and verify addresses typed into the search bar.	✓
9	The system should run asynchronous tasks related to queries in parallel.	✓
10	The system could be able to disable notifications if the battery is less than 10%.	✓
11	The system could disable the map view if the user does not have a Wi-Fi connection.	✓
12	The application could resize pictures to avoid using too much memory loading big images (e.g.: 2560*1920 resolution).	✓
13	The system could store the user's settings on the phone, using shared preferences.	✓
14	The system could follow the Fragment Navigation Pattern.	✓
15	In 95% of the cases the user should not take more than 5minutes to make a working profile	✓
16	The phone should not compute any filtering between tenant and rooms.	×
17	If the user has some tenant or rooms to be displayed in the match process, the responsetime to load the first tenant or room cannot exceed 3 seconds, while having at least 80% of the Wifi connection.	✓
18	The functions to filter the rooms and tenant will have complexity O(NlogN) or lower.	✓
19	The size of the application should not be bigger than 40MB.	✓
20	The user must not download from the database any picture bigger than 8MB	✓
21	The application must be running 24/7, and in case of a system shutdown for maintenance,need to be informed to the users beforehand	✓
22	The application is going to be developed in Android from minimum SDK: 19 to max SDK:28	✓
23	The application will be developed in Java	✓
24	The application's code will have Java doc for the core methods.	✓
25	The application's code will have no intetional warning from the IDE.	✓

26	The application will use the camera and gallery if it gets the permission from the user.	✓
27	The application will use 'Builder Design Pattern' to create the models	✓
28	The system will use SharedPreferences to store any application configuration	✓
29	The databases should only allow registered users to make queries.	✓
30	No passwords will be available to read for any database administrator.	✓

Chapter 9

Future Perspective

At present, the application does not fulfill all the requirements we have, as results from the White Box and Black Box testings. The reason for this is that we focused on implementing as many features as possible, by following the order given by the MoSCoW method. If we were to have more time and resources, we would have implemented the missing requirements and also other features we thought of in the process of coding.

Our application is missing a tutorial that would show the users what they can do with our application. As clear as it may seem to us who implemented it, there is still need for a guide to inform users of the features our app has and how to use them. For example, there could be users who never heard of Tinder or any other application alike and therefore, the swiping and matching system would be completely new to them.

WeRoom needs some improvement also on the log-in and sign-up pages. A first thing would be to verify if the email used for account creation is valid, because for now, the system only checks if the email address has the proper format. This feature is very important, because we would not want users with fake emails to use our application, both because they would not be able to recover/change the password and because it is a security measure against scams. As previously mentioned, changing the password is not possible at the moment, so that would be another change that should be made.

Regarding the profiles of the users, for the ones who sign up using Facebook or Google, the system should automatically complete their profiles with information gathered from the external profiles, like their name, date of birth, profile picture and others as such. Moreover, a feature that would be useful is showing the rent and deposit of each room in the currency of the country where the room is located. WeRoom should have this feature, because the users should be able to rent/search rooms from all over the world. Being also related to the profiles, the users should be able to have both a landlord and a tenant profile. In addition to this, the users should have also the possibility to deactivate tenant/room profiles or accounts and to activate them back by simply logging in to our application.

Another idea we had that is not implemented at this time is that we should show users after choosing all the filters what percent out of all the tenants/rooms available match the filters. This way, users would know if they are being too picky with their wishes.

To follow completely the concept of Tinder swiping, we should implement a button for undoing a swipe. With this feature, the users are able to bring back the last swiped profile, so that they can see more information and re-decide which way to swipe.

The chat needs a series of changes as well. The layout should look more like the one from Tinder, as that is typical look of a chat application. When sending messages, the name of the user should appear both in the top of the page and above each message and below the messages should be the time when it was sent. Furthermore, the chat should have settings that include the un-match feature, which deletes all connections between the two users and also the reviewing option, which allows users to rate and describe each other.

As a last expansion according to our ideas, the system should notify the user also when more swipes are available and when they get a message. For now, the system only notifies users when they match with other users.

Chapter 10

Discussion

The project gave us the opportunity to understand how an idea is being transformed into a software. During this transformation, we used different software engineering methods to model and design our idea into formal diagrams and requirements. Those diagrams and requirements helped the development process, giving us tools and illustrations that represent our desired system.

Project Management

The project took 15 weeks from transforming our idea until we had a product. We acknowledged from the beginning, that we had an ambitious project and we took it into consideration, the importance, that our project management would have to accomplish all the requirements for our application

We decided to use Scrum as a process model, as it brings small weekly deadlines and daily stand-up meetings that help sharing information among the group members. Using Scrum was a key choice for the development. We decided to start using Scrum on the 6th week of the project, and we had a clear overview that we had to split the rest of the design, research and implementation into 10 sprints. Sprint review was

done after each of the sprints and it gave us the opportunity to discuss how we should handle the future sprints, and discuss how we could improve the work-flow on the Scrum. We saw the importance from the first sprint of the scrum meetings and how critical is to share information about design, technology choices and implementation between the developers. Without this, we would have ended up having miss-communication issues during the implementation.

Using Scrum created some challenges as-well. The group members had to learn Android development while developing the project, making it hard to estimate the time needed for some task during the sprints. We ended up miss calculating the time required to implement different task, extending at least one task from one sprint to the next sprint, in most of the sprints.

We had the Scrum roles overlapped between the project members. We had to cover the product owner, scrum master and developer team with our group. During the implementation, we missed a product owner role who could validate the software that was developed. This created some confusion between the members, as we had different opinions, if a task was implemented correctly or not.

Three weeks before the end of the project, the members had to take an important task and cut out some of the requirement that we had for the application. The quality code was being compromised by trying to implement too many tasks in the same sprint, and we decided to undercut some of the requirements instead.

Using a MoSCoW categorization, lead us to the requirement that had to be postponed for a future perspective of WeRoom. We didn't compromise any of the core functionalities of the application during the process and we are satisfied that we managed to deliver a functional product.

Design

Having a clear design on how WeRoom should work and interact before starting the implementation was fundamental, to be able to deliver our product. Different software engineering tools were used before the implementation phase began. We had different models describing how components of our system should interact, what were the use cases of the system, what models did our system have and how they where related etc.

This generated documentation were used during the implementation, giving the group members clear task on what they had to implement, how the implementation should communicate with the rest of the application, having an overview on how the navigation between the screens should be etc.

In some cases, we delivered some difficulties on the implementation from our design phase. Once we got to the implementation phase where we had to create the chat and the chat selection room, we had to re-structure the class diagram on how the matches are stored and how the chat rooms are being created. After changing the structure of our models, we generated extra work re-factoring other classes and views. On top of that we had to delete the values of our databases.

We learn the importance of designing the system correctly. Any changes in future phases creates big cost on work and resources. Even worse, it might be too late to accept any new improvements in the design if the cost is deleting all the data from our databases.

Implementation

The implementation gave us the opportunity to learn how to develop Android applications, how to work with asynchronous tasks, how to work with databases for the first time, how to standardize code with design patterns and standardize the layouts with the help of material design.

Material design brings standards on the appearance and navigation for the application. Integrating this into our application, we improved the user experiences bringing a consistent visual language[130].

The biggest milestone during the implementation was the usage of the database. WeRoom follows a MVC design pattern. The main idea of our application is to show some data stored in our database on a Phone application and handle the relationship with the data of our database.

How the data has been structured in the database and the choice of the database has a big importance on the performance of the application. Having a good structure of our data will prevent having any redundant information on our models, and will allow the application to get all the information needed in a single query.

The main problem that we detected on our application was the choice of the databases. Our main intention was to filter all the requested data from the queries, so the phone didn't had to perform any operation filtering all the users of our database.

After the 5th sprint, when we had to start reading data from our database, we saw a challenge on filtering the data from the query. We learned that non-SQL databases are not the best choice when we wanted to range filter on different field on a single query.

Therefore, We researched different databases from FireBase, and found out that both of them are based on Non-SQL databases.

We had two options to solve this issue: Perform the filters on the phone, or change the database to a SQL database.

We diagnosed this issue on the 5th sprint and it was too late to change the database, so we took the decision of filtering all the profiles on the phone. This is a non-functional requirement that had to be compromised to be able to deliver the product on time.

Summary

Overall, we managed to deliver the product that we had in mind on a 15 weeks period. Using an agile process model, gave us the right tool to cut and adapt the requirement of the application during the implementation. Creating all the modeling of our system and having a clear MoSCoW requirement

categorization, assisted us to have a clear overview on which functionalities could be removed without compromising the main functionality of the design system.

Chapter 11

Conclusion

Our problem formulation was:

How can we make the room renting procedure easier by developing a system for Android users?

With the sub-questions:

- How can we use a process model to optimize the implementation of the application?
- How can we use software engineering and design tools to structure the system?
- How can we make the user interface experience better, by following the material design?

To answer the problem formulation, we decided to make an application called WeRoom. Since it can be hard to target a room/tenant that meets one's requirements, we wanted the application to pair them based on these requirements. By using a "swipe, match and chat" system, we set up a connection between the landlord and tenant, where the rest of the process will then be up to them.

The process model used during our project was the Scrum, which we combined with the KanBan. Having a sprint each week, helped us dividing the project into smaller pieces and therefore giving us a better overview of what tasks had to be done. The Scrum meetings in the beginning of every group work also helped knowing what each member were currently working on and planning to finish during the day.

All the tasks we had to do during a sprint, were written on post-it notes and put on the KanBan wall. By doing this we could always keep track of what had to be done, what is currently being worked on and what has been done.

By combining the process model with KanBan, it was easier to manage the work-flow. It optimized the efficiency of the group and helped us communicate constantly. By having this overview of all the tasks, we always knew what had to be implemented before we could start doing new objectives.

Finding the functional and non-functional requirements in the beginning of the project, we could make then the UML diagrams. The diagrams helped us finding, not only the use-cases for our project, but also to identify the external and internal actors along with making design choices, development priorities, finding the interactions between the objects, the relationship between classes and lastly, the behavior of our system.

By now knowing this, we could make the Wireframing and sketch out ideas for the application. A prototype was made to give us a better overview of how the screens should be connected and where to implement what. The prototype was then tested with user acceptance testing to see if there was more to implement in the application and if we should make changes in the layout.

Material design also played and essential role in the project, since we used it as a guideline to design the user-interface. Following the material design, we go more in depth with the design of the application.

We wanted our application to be simple and easy to navigate for our users, so a bottom navigation bar was implemented to easily switch between the fragments, and by displaying the icons, there is no doubt as to what button leads to where.

The different buttons encountered, has a simple design, following a common theme all through the application, so the user is not in doubt where to click. Lastly, there is the cards and dialogues. The cards can

be swiped left or right and is displaying simple information about either a room or a tenant. The user is then notified with a notification, if there is a match. The dialogues appears in the middle of the screen when the user performs certain actions and displays information of what is currently happening.

Combining all of this, it is clear for the user to navigate and see the information displayed throughout the application, since the user should never be in doubt of what to do.

In conclusion, WeRoom is a unique and innovative application that tries to help solve the problem of finding a room/tenant by making it easy to find something within the users preferences and connect them with each-other.

References

[1] https://www.creditdonkey.com/why-rent.html

[2] https://www.cnbc.com/2018/09/05/its-better-to-rent-than-to-buy-in-todays-housing-market.html

[3]https://www.copenhageneconomics.com/publications/publication/housing-market-analysis-of-greater-copenhagen-housing-shortage-urban-development-potentials-and-strategies

[4]http://cphnews.mediajungle.dk/archives/4916

[5]https://nethouseprices.com/news/show/2370/renting-out-a-room-the-pros-and-cons

[6]https://material.io/design/

[7]https://developer.android.com/guide/topics/ui/look-and-feel

[8]FireStore queries documentation: https://firebase.google.com/docs/firestore/query-data/queries

[9]Petter L. H. Eide (2005). "Quantification and Traceability of Requirements"

[10]http://www.coleyconsulting.co.uk/moscow.htm

[11]Ian Sommerville, 2011, Software engineering

[12] Jacobson Ivar, Christerson Magnus, Jonsson Patrik, Övergaard Gunnar, Object-Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley, 1992.

[13]Ralph, Paul (2015). "The Sensemaking-coevolution-implementation theory of software design". Science of Computer Programming.

[14]Alexander and Beus-Dukic, "Discovering Requirements", 2009. Page 120

[15]Context diagram: https://www.cs.uct.ac.za/mit_notes/software/htmls/ch06s06.html

[16]Use case diagram: https://cacoo.com/blog/how-a-use-case-diagram-can-benefit-any-process-and-how-to-create-one/

[17]Class diagrams: https://courses.cs.washington.edu/courses/cse403/15sp/lectures/L9.pdf

[18]Sequence diagram: https://courses.cs.washington.edu/courses/cse403/16wi/lectures/L10.pdf

[19]https://en.wikipedia.org/wiki/Software_architecture 29/03/19

[20]Ian Sommersvile, 2011, Software Engineering

[21]Ian Sommersville, 2011, Software Engineering

[22]Software Engineering Slides "Process models - how to organize your software development, part 2, February 18 2019

[23]https://www.justinmind.com/

[24]Ian Sommersvile, 2011, Software Engineering 9Th edition

[25]The "Gang of Four", 1994, Design Patterns: Elements of Reusable Object-Oriented Software

[26]https://material.io/design/introduction/

[27]Ian Sommerville, 2011, Software engineering

[28]http://softwaretestingfundamentals.com/acceptance-testing/

[29]http://softwaretestingfundamentals.com/black-box-testing/

[30]https://www.guru99.com/black-box-testing.html

[31]https://www.guru99.com/black-box-testing.html

[32]https://www.guru99.com/white-box-testing.html

[33]https://www.guru99.com/white-box-testing.html

[34]http://softwaretestingfundamentals.com/white-box-testing/

[35]https://developer.android.com/studio/test/monkey

[36]Android unit testing: https://developer.android.com/training/testing/unit-testing

[37]https://en.wikipedia.org/wiki/General_Data_Protection_Regulation 29/03/19

[38]http://boligportaler.dk/

[39]https://www.boligportal.dk/info/om-boligportal/

[40]https://www.facebook.com/pg/Findroommate.dk/about

[41]https://en.wikipedia.org/wiki/Airbnb

[42]https://material.io/design/

[43]https://play.google.com/store/apps/details?id=com.tinder

[44] Chamorro-Premuzic, Tomas (2014-01-17). "The Tinder effect: psychology of dating in the technosexual era". the Guardian. Retrieved 2018-03-27.

[45]https://jobfisk.dk/jobsogning/nyheder/2268-danske-meew-forenkler
-jobsogningen-og-skaber-problemer-for-rekrutteringsbranchen

[46]https://www.copenhageneconomics.com/publications/publication/demography-housing-needs-and-price-development-in-copenhagen

[47]https://www.copenhageneconomics.com/publications/publication/demography-housing-needs-and-price-development-in-copenhagen

[48]https://www.rd.dk

[49]https://charliesroof.com/about-denmark/the-danish-rental-market-and-how-we-nail-it/

[50]http://lexicon.ft.com/Term?term=two_sided-markets

[51]https://reasonstreet.co/business-model-two-sided-marketplace/

[52]https://press.airbnb.com/about-us/

[53]Scenarios purpose and usage: https://www.cs.cornell.edu/courses/cs5150/2014fa/slides/D2-use-cases.pdf

[54]Bourque, P.; Fairley, R.E. (2014). "Guide to the Software Engineering Body of Knowledge (SWEBOK)"

[55]Chen, Lianping; Ali Babar, Muhammad; Nuseibeh, Bashar (2013). "Characterizing Architecturally Significant Requirements"

[56]Ian Sommerville, Software Engineering, 9th

[57]Ian Sommerville, 2011, 9th

[58]https://courses.cs.washington.edu/courses/cse331/12au/sections/Section8Slides.pdf

[59]Ian Sommerville, Software Engineering, 9th

[60]https://gist.github.com/Mariovc/f06e70ebe8ca52fbbbe2

[61]Smart Lock function https://developers.google.com/identity/smartlock-passwords/android/get-started

[62]Facebook authentication: https://firebase.google.com/docs/auth/android/facebook-login

[63]Google authentication: https://firebase.google.com/docs/auth/android/google-signin

[64]Tag library: https://android-arsenal.com/details/1/7582

[65]Swipe action library: https://github.com/nihad92/SwipeableCards

[66]SeekBar library: https://github.com/woxingxiao/BubbleSeekBar

[67]Ian Sommerville, Software Engineering, 9th

[68]Sommerville, Software Engineering, 9th

[69]https://en.wikipedia.org/wiki/Builder_pattern

[70]Sommerville, Software Engineering, 9th

[71]Ian Sommerville, Software Engineering, 9th

[72]Clegg, Dai; Barker, Richard (1994). Case Method Fast-Track: A RAD Approach. Addison-Wesley. ISBN 978-0-201-62432-8

[73]"MoSCoW Analysis (6.1.5.2)". A Guide to the Business Analysis Body of Knowledge (2 ed.). International Institute of Business Analysis. 2009. ISBN 978-0-9811292-1-1

[74]https://sweetpricing.com/blog/2017/02/average-app-file-size/

[75]https://academy.realm.io/posts/michael-yotive-state-of-fragments-2017/

[76]https://firebase.google.com/docs/auth/

[77]Google places introduction: https://developers.google.com/places/web-service/intro

[78]Google Maps V2 Android Play Store dependency: https://developers.google.com/android/guides/setup

[79]Vaskaran Sarcar, Java Design Patterns (2015)

[80]https://www.toptal.com/android/android-fragment-navigation-pattern

[81]"The Builder design pattern - Problem, Solution, and Applicability". w3sDesign.com

[82] The DCI Architecture: A New Vision of Object-Oriented Programming – Trygve Reenskaug and James Coplien – March 20, 2009

[83]https://www.baeldung.com/java-adapter-pattern

[84]https://www.javatpoint.com/singleton-design-pattern-in-java

[85]https://material.io/design/

[86]Eva Heller, Psychologie de la couleur - effets et symboliques, p. 179-185

[87]Navigation Bar material design: https://material.io/design/components/bottom-navigation.html

[88]https://material.io/design/components/cards.html

[89]https://material.io/design/components/dialogs.html

[90]https://www.jesseshowalter.com/articles/how-to-wireframe-a-website-or-app/

[91]https://firebase.google.com/docs/auth/android/google-signin

[92]https://www.androidauthority.com/how-to-integrate-smart-lock-in-android-apps-702638/

[93]Android QuickStart GitHub: https://github.com/firebase/quickstart-android

[94]FireBaseAuth credential saving: https://firebase.google.com/docs/auth/web/auth-state-persistence
[95]JSON: JavaScript Object Notation - https://www.mongodb.com/json-and-bson
[96]Real-Time database vs Cloud FireStore comparision: https://firebase.google.com/docs/database/rtdb-vs-firestore
[97]Cloud FireStore price list: https://firebase.google.com/docs/firestore/pricing#europe
[98]Firebase Storage: https://firebase.google.com/docs/storage
[99]Design pattern list: https://www.javatpoint.com/design-patterns-in-java
[100]Adapter Design Pattern: https://www.javatpoint.com/adapter-pattern
[101]Google free 12 month trial: https://cloud.google.com/free/
[102]More information about thread and UI thread on Android: https://developer.android.com/guide/components/processes-and-threads
[103]Documentation on Gesture Detection https://developer.android.com/training/gestures/detector
[104]Recycler View Documentation https://developer.android.com/reference/android/support/v7/widget/RecyclerView
[105]Guide on using a RecyclerView https://guides.codepath.com/android/using-the-recyclerview
[106]Guide on using Card View https://guides.codepath.com/android/using-the-cardview
[107]Material design Card https://material.io/design/components/cards.html#implementation
[108]External API Swipeable Cards https://github.com/nihad92/SwipeableCards
[109]Realtime vs Firebase https://firebase.google.com/docs/database/rtdb-vs-firestore
[110]Serializable vs Parcelable https://android.jlelse.eu/parcelable-vs-serializable-6a2556d51538
[111]Open implementation how a chat is displayed on Android: https://github.com/DevExchanges/Android-Bubble-Chat
[112]Nine patch image Android documentation: https://developer.android.com/studio/write/draw9patch
[113]https://material.io/design/components/sheets-bottom.html
[114]https://developer.android.com/reference/android/provider/MediaStore
[115]Best practice Tablayout: https://material.io/design/components/tabs.html
[116]Haversine formula give the distance between two coordinate points on a sphere: https://en.wikipedia.org/wiki/Haversine_formula
[117]Android services documentation: https://developer.android.com/guide/components/services
[118]Notification in Android: https://developer.android.com/training/notify-user/build-notification
[119]https://www.geeksforgeeks.org/software-engineering-white-box-testing
[120] "UI/Application Exerciser Monkey — Android Developers". developer.android.com. Retrieved 2016-04-25
[121]https://en.wikipedia.org/wiki/Acceptance$_t esting$
[122]https://material.io/design
[123]https://www.freeprivacypolicy.com/blog/gdpr-privacy-policy/
[124]https://www.geeksforgeeks.org/software-engineering-white-box-testing
[125]Mocking a unit test: https://www.vogella.com/tutorials/Mockito/article.html
[126]Call graph explanation: https://en.wikipedia.org/wiki/Call_graph
[127]The CEO of Microsoft: 80-20 Rule Applies To Bugs, Not Just Features: https://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm
[128]Pareto Principle: https://en.wikipedia.org/wiki/Pareto_principle#In_computing
[129]Oracle documentation, Java Reflection: https://www.oracle.com/technetwork/articles/java/javareflection-1536171.html
[130]Why use material design: https://medium.com/pilcro/should-you-use-material-design-bfb596a04bae?fbclid=IwAR0vMEuHUAi d2oyUS73L0C-fbVtrbXdDiCH5XiDZP7e660OneDqM4L0sZoA