
Gamification of Mobile OpenStreetMap Contributions

Developing MapTogether

Rasmus Nørgaard Fjeldsø
Thomas Møller Grosen
Rasmus Hartvig
Sebastian Hjorth Hyberts
Phillip Bengtson Jørgensen
Simon Svendsgaard Nielsen

Aalborg University



AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science

Selma Lagerløfs Vej 300

Phone: +45 99 40 99 40

Fax: +45 99 40 97 98

<http://www.cs.aau.dk>

Title:

Gamification of Mobile OpenStreetMap Contributions

Topic:

Mobility

Project period:

01-02-2021 - 27-05-2021

Project group:

SW812F21

Authors:

Rasmus Nørgaard Fjeldsø
Thomas Møller Grosen
Rasmus Hartvig
Sebastian Hjorth Hyberts
Phillip Bengtson Jørgensen
Simon Svendsgaard Nielsen

Supervisor:

Daniele Dell'Aglio

Page count: 68

Abstract:

In this project, we analyse the map data of OpenStreetMap and the issues that arise when the quality of the data relies solely on volunteers and charity. Furthermore, we analyse gamification theory and how it can motivate different types of players to perform a task they otherwise would not. We use this analysis to design MapTogether, a mobile application that utilises gamification and social features to motivate users to contribute with useful map data to OpenStreetMap. From the design, we implement a prototype of MapTogether. To support the necessary features, we design and implement a MapTogether server with an API to persist data between clients and an OpenStreetMap API wrapper to download map data and upload contributions. In both of these APIs, we keep security in mind by verifying and authenticating the clients' OpenStreetMap credentials. The prototype has in its current state features such as user profiles, followers, leaderboards and maintenance quests to contribute data to OpenStreetMap.

Contents

Contents	ii
1 Introduction	1
2 Problem Analysis	3
2.1 Types of Contributions	3
2.1.1 Notes	3
2.1.2 Tagging	3
2.1.3 Adding Points of Interest	4
2.1.4 Mapping	4
2.2 OpenStreetMap Contributors	4
2.2.1 Amateurs	4
2.2.2 Professionals	4
2.2.3 Corporations	4
2.2.4 Governments	5
2.2.5 Bad Actors	5
2.3 Accuracy and Precision	5
2.4 Error Types in OpenStreetMap	7
2.5 Existing Solutions	7
2.5.1 StreetComplete	8
2.5.2 Urbanopoly	9
2.6 Gamification	9
2.6.1 Game Elements	9
2.6.2 Reason for doing gamification	9
2.6.3 User Types	10
2.7 Problem Statement	11
3 Design	13
3.1 Audience	13
3.2 Requirements	16
3.3 Game Design	18
3.3.1 Achievements and Player Profiles	20
3.3.2 Time-based Leaderboards and Brackets	21
3.3.3 Socialising and Gamified Mapping Parties	21
3.3.4 Gamification of Surveying	22
3.4 Features	23
3.5 Object-Oriented Design	25
3.5.1 Problem Domain	26
3.5.2 Application Domain	28
3.6 User Interface Design	30
3.6.1 Main Screen	31
3.6.2 Social Screen	33
3.6.3 Login Screen	36

3.6.4	UI Flow	36
4	Implementation	39
4.1	Architecture	40
4.2	Subsystems and Components	41
4.2.1	Graphical User Interface	41
4.2.2	Map-Data	42
4.2.3	Quest-Finder	42
4.2.4	Tile-Renderer	42
4.2.5	Social-Information Handler	43
4.2.6	Login-Manager	43
4.2.7	Leaderboard-Calculator	43
4.2.8	Quest-Solver	44
4.3	Connecting Devices	44
4.4	Log-In Handler	46
4.5	Database and Server	48
4.5.1	Database and Server Design	49
4.5.2	Database and Server Implementation	51
4.6	OpenStreetMap API Wrapper	54
4.6.1	Endpoints	54
4.7	Client	57
4.7.1	Futures	57
4.7.2	Provider	57
4.7.3	Quest System	58
5	Evaluation	61
5.1	Performance Testing	61
5.2	Discussion	62
5.2.1	Fulfilment of Requirements	62
5.2.2	Work Process	63
5.3	Conclusion	64
5.4	Future Works	65
5.4.1	Verification Quests	65
5.4.2	Tile Renderer	65
5.4.3	Activity Motivation Experiment	65
	Bibliography	67

Chapter 1

Introduction

Location-based services often require reliable and up to date map data, whether it is simple information such as road networks or the opening hours of shops. OpenStreetMap¹ provides completely free community-driven map data. The idea behind OpenStreetMap is that anyone can freely use the data. Companies such as Google also provides map data through Google Maps API. However, the uses are limited as specified by their terms of service and are also not free for commercial use. OpenStreetMap is thus quite attractive for developers and is used by services such as Snapchats SnapMap².

The data provided by OpenStreetMap is driven by a community of volunteers, and organised by the (non-profit) OpenStreetMap Foundation³. Much like Wikipedia, anyone can contribute with new data. This has both advantages and implications. Because it is easy to change, the users can quickly update the data to include the most recent information. The data, however, is also subject to the unpredictability of the laymen population. Because of the origin and the incentives (or lack thereof) of contributing with new data, we can imagine at least two issues; missing data, because nobody in the vicinity has an incentive to add it and incorrect data, either because of malicious intent or by human error.

In the following chapter, we analyse the quality of OpenStreetMap and problems related to the incentives of voluntarily contributing.

Who contributes to OpenStreetMap? Are there any specific challenges preventing people from contributing? Are there any existing solutions to increase OpenStreetMap contributions and contributorship?

After the the analysis we specify requirements and design the core game loops and interface of MapTogether. We implement different components that combined work together to form a slice of the complete featureset.

¹OpenStreetMap www.openstreetmap.org

²<https://map.snapchat.com/about>

³OSM Foundation https://wiki.osmfoundation.org/wiki/Main_Page

Chapter 2

Problem Analysis

In this chapter we start from the initial problem in Chapter 1 and analyse OpenStreetMap, as well as the incentives of contributing to such a project. We also identify sources of errors and how gamification can motivate people. The analysis ends with a problem statement that will carry us into the design phase.

2.1 Types of Contributions

In this section, we will define a few types of types of contributions to OpenStreetMap. The data in OpenStreetMap consists of nodes, ways and relations. A node is used to represent an object that is described by a singular point on the map. Examples of nodes are single trees, benches or a bus stop. A way is a list of nodes and typically represents things such as buildings and roads. Relations are collections of nodes and ways. A relation can represent things such as bus routes or complex buildings that can only be drawn as multipolygons.

There are different ways one can contribute to OpenStreetMap. This section categorises the different types of edits any contributor can make. The categories of changes have different levels of user involvement.

2.1.1 Notes

Contributors can add notes to OpenStreetMap. Notes are text comments specified on a specific coordinate, pointing out errors or other information. Notes can be read by other contributors and help others to refine the map data. Placing notes is the simplest way of adding information.

2.1.2 Tagging

Much information in OpenStreetMap data is in the form of *points of interests* (POI) represented by a node. A point of interest can, for example, be a tree, shop or a bus stop. Tags refer to information about points of interest. Points of interest can have several tags with information, such as the number of floors of a building or the material of a bench. We say that tagging happens when a contributor adds or changes tags of points of interest. Adding/changing tags to an already specified point of interest is an easy way to contribute to OpenStreetMap; it can be seen as answering simple questions like “what are the opening hours of this shop?”. Mapping or adding new points of interest often involves tagging.

2.1.3 Adding Points of Interest

Adding new points of interest is more complicated than tagging since contributors have to add new data represented by a single node with an accurate coordinate. In addition, adding points of interest requires contributors to find features that are not, but should be, represented in OpenStreetMap. They then have to select the specific location of the node.

2.1.4 Mapping

Mapping refers to adding roads, paths and buildings through a series of nodes. The difference between “mapping” and “adding points of interest” is that mapping adds new data features represented by multiple nodes. Common for paths and buildings is that they are represented as a list of connected nodes. Contributors who want to add new buildings or roads must accurately specify all the nodes on the map. When a contributor adds a new road or building, it is easy to add a few tags; thus mapping often involves tagging. Mapping is the most complicated type of editing because it requires contributors to be accurate about the nodes. Luckily, the placement of paths and building does not change as often as all information specified in tags.

2.2 OpenStreetMap Contributors

OpenStreetMap is, like Wikipedia, open for anyone to edit and change for better or worse. This allows for anyone to improve the data for the benefit of everyone, but it also means that there is the possibility of vandalism. Giving everyone the ability to edit the data does not mean that all users do. OpenStreetMap and Wikipedia both roughly follow the 90/9/1 rule of thumb, that is, 90% of users never edit, 9% of users edit sporadically and 1% of users regularly edit [1]. For OpenStreetMap the top 1.4% of editors are responsible for 90% of contributions and 1 to 13% of users will have contributed data in any given month [2].

2.2.1 Amateurs

A large portion of people who edit OpenStreetMap only edits a few times and are usually beginners when it comes to mapping. The average person probably does not have any experience mapping, nor any specialised equipment for it.

2.2.2 Professionals

Professionals, as defined by Yang et al. [3], are able to use the most complete and advanced mapping software like JOSM, as well as contribute a large quantity of high-quality data and regularly edit. For countries like France, UK and Germany, most OpenStreetMap data comes from professionals, which is just a fulfilment of the 90/9/1 rule. Professionals can reasonably be expected to be able to perform all contribution types described in Section 2.1.

2.2.3 Corporations

There are also multiple corporations that contribute to OpenStreetMap, including some of the largest IT companies such as Amazon and Apple [2]. The corporations have different interests in the map-data. However, what all of them have in common is that they have a large percentage of the total contributions in their location. However, corporations also contribute outside their immediate vicinity. They also increase the total contributions, an example of this is a five-fold increase in edits regarding already existing road networks [2].

⁰JOSM <https://josm.openstreetmap.de/>

2.2.4 Governments

The Danish government (among others) also contributes to OpenStreetMap by supplying aerial imagery. The Agency for Data Supply and Efficiency provide image coverage of the Danish acreage [4]. These images are captured every year in the spring, and later the same year released for usage by OpenStreetMap, or any other service, and they have a resolution of 10 cm² which means that each pixel corresponds to 10 cm x 10 cm area [5]. In order to get a perspective on this resolution, the finest resolution commercial satellites can capture, as of April 2019, is a resolution of 30cm [6]. This resolution means that the imagery of Denmark has a high level of details, which improves the overall quality of OpenStreetMap.

2.2.5 Bad Actors

Some people abuse the fact that anyone can contribute to OpenStreetMap. They modify or create information that somehow benefits them. This vandalism peaked at the launch of the game Pokemon Go, which uses information from OpenStreetMap in order to determine where different Pokemon should be placed [7]. Most of the changes happen within a 5 km radius of the vandal, but most do not sustain these vandalism activities. The non-vandalising part of the community fixes 65% of the errors introduced within a day [7].

2.3 Accuracy and Precision

Given that OpenStreetMap gathers their data from user reports, errors and inaccuracies in the map are bound to occur, whether, through a user error or malicious intent. When talking about errors in geospatial data, there are typically two factors that are taken into consideration, accuracy and precision.

We define accuracy as the degree of closeness to which the information on the map matches the values in the real world. Therefore, accuracy is a measure of the quality of the data and the number of errors contained in a certain data set.

We define precision as how the description of data is. Precise data may be inaccurate, as it could be exactly described but inaccurately gathered. This can happen if, for example, the mapper made a mistake or the data was recorded wrongly in the database.

Fig. 2.1 shows examples of data that has been gathered with varying degrees of accuracy and precision, as well as how a lack of either affects the quality of the data set.

In a real-world example, a precise but inaccurate measurement might show a road on the map as it looks in the real world, however, shifted 20 meters away from its actual position. Conversely, data gathered accurately but without precision would have the position of the road be closer to where it is in real life, but not guarantee the road's form or shape.

The degree of accuracy required to make a correction is highly dependant on existing data for a given area. For example, in areas where there are no previous contributions, even an inaccurate trace of a road that a contributor adds without the use of GPS or other sources is an improvement [9]. Likewise, in dense areas such as a city, many users refine the data over time, and users should be cautious when adding new data.

Ideally, a mapping system gathers data both precisely and accurately. However, according to the OpenStreetMap wiki[9], there can be multiple reasons for data to be inaccurate or imprecise, with some being rather obvious and others more challenging to notice.

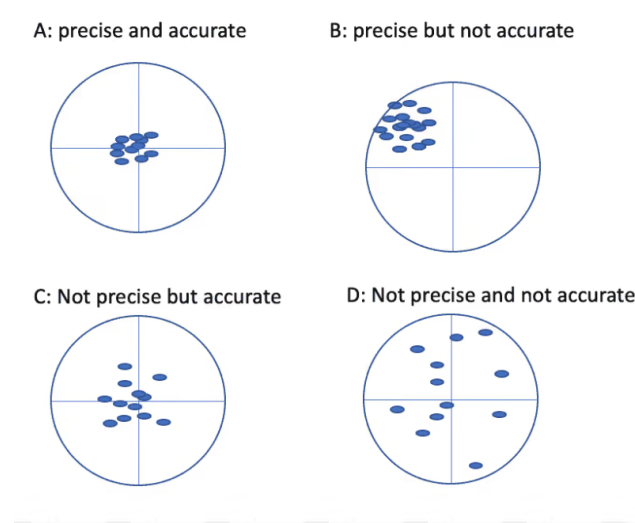


Figure 2.1: Examples of accuracy and precision, the crosshair represents the true values of each entity with the dots representing measured/recorded values [8]

GPS data is one of the significant sources for OpenStreetMap. If the user allows it, data can be collected about a geographical area as a person travels through it with a GPS. Generally, this provides relatively accurate data as to the location of the point of interest; however, it also has several errors. GPS is optimal when the receiver has a clear view of the sky in all directions, which is not always the case, and repeatable errors can occur. If a contributor goes out to record map data of a specific area, they cannot be sure that their data is accurate. Even if the contributor records a path of them starting and stopping in the exact same location and then repeat the route the next week with the same equipment, they may be several meters off the original path simply due to the nature of GPS. Because of this, it is also generally a bad idea to correct 'errors' in other people's GPS traces that do not disagree significantly with that of another user. There may also be specific areas where a satellite consistently is obscured by a building or a similar object. This can then cause consistent inaccuracies on specific points in the map data [10].

Aerial imagery and satellite pictures can be used in a process called georectification, which combines the image set into a single image that is consistent with the ground. Often this is both an accurate and precise method for gathering data, especially in built-up areas where edges can easily be matched, and there are likely to be already surveyed reference points. In many regions, aerial imagery is considered as being nearly completely accurate, and using aerial imagery is often both more accessible and more accurate than relying on repeated GPS measurements. One exception is if the mapper has access to special hardware, which allows for a better accuracy using GPS, but this is rarely a viable solution for the typical mapper. The problem with aerial imagery is that the mapping done based on the imagery, can only be as recent and valid as the imagery provided.

Most of these errors are mainly prevalent in areas with little to no prior data. In such cases, it is still preferable to have map data with inaccuracies as opposed to having no data at all. Meanwhile, in dense areas such as large cities, many users are already present and errors have a good chance of being corrected by other mappers, once they discover them.

2.4 Error Types in OpenStreetMap

Errors in OpenStreetMap are classified as situations where the map differs from the real world. The differences are caused either by faulty reporting or a lack of information regarding a specific area. This section presents some common errors in OpenStreetMap.

One apparent error when using OpenStreetMap is the lack of data pertaining to shops and points of interest, especially in small towns. Unless a shop or location is reported manually by a user, it does not appear on OpenStreetMap. This means that many points are not recorded on OpenStreetMap as they are in the real world. An example of this can be seen below in Figures 2.2 and 2.3 where the shops located at Adelgade in the city Skanderborg are shown both in OpenStreetMap and the proprietary competitor Google Maps.



Figure 2.2: Adelgade in Google Maps

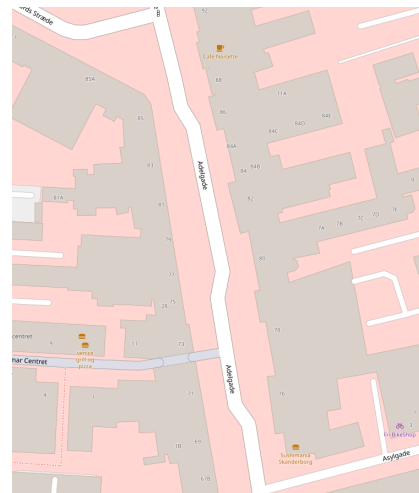


Figure 2.3: Adelgade in OpenStreetMap

Looking at Figures 2.2 and 2.3, it becomes apparent that Google Maps has vastly more information readily available pertaining to shops and their locations when compared to OpenStreetMap.

While the disparity in the figures seems extreme, this appears to primarily be a problem in small cities and towns, as mapping is generally more complete and descriptive in more populated areas.

The age of the data may be another source of error. When the data sources become outdated, some, or a large part, of the information may become stale as to how a location now looks in the real world. For example, if a road is permanently closed or redirected or a shop in town goes bankrupt, the data pertaining to the road or shop becomes invalid and incorrect. These types of errors are usually detected by others in dense areas with lots of contributors. However, in smaller, rural areas where user updates are less frequent, these errors can potentially exist for a long time.

2.5 Existing Solutions

In Section 2.1, we investigated different types of contribution. We discovered that only a few of the users contribute to improving the OpenStreetMap data, for which we can find two reasons. Firstly, it could be that the users are not aware that OpenStreetMap is open and

that they can contribute. Secondly, even when the users are aware that they can contribute, it might be too difficult or uninteresting to contribute. There exist multiple systems which aim at making it easier and more enjoyable for different users to contribute. Most of these systems use gamification to varying degrees.

Gamification is the use of video game elements in non-gaming systems. Video game elements can improve the participation and motivation of people in carrying out tasks that they do not otherwise find interesting to perform. These could, for example, be people who usually do not contribute to OpenStreetMap because they find it boring or unrewarding [11].

Some of the game mechanics that can be used to gamify mundane tasks are point systems, levels, leaderboards, and positive feedback [11].

2.5.1 StreetComplete

One example of a system using gamification is StreetComplete [12]. StreetComplete is an app that finds and shows incomplete data near the user and displays it as tasks on a map on the screen. The tasks can be questions about road names or the opening hours of local shops. Fig. 2.4 shows the map with tasks in the StreetComplete application and Fig. 2.5 shows an open task after tapping on it. One of the game mechanics used in StreetComplete is points. Thus, a counter constantly shows how many contributions the user has made, even though it is somewhat unrelated to the mapping itself. Another game mechanic is the use of positive feedback in terms of badges. As an example, a user receives a badge after completing 30 tasks.

StreetComplete has more than ten thousand downloads from the Google Play store and a rating of 4.8/5 which indicates the application is used and liked by the users.

On the 4th of March 2021, version 31.0 of StreetComplete was released, with a social feature called StreetComplete teams, which aims at allowing people to map better while spending time with other people. Previously when two friends map together, they would both see the same tasks. With StreetComplete teams, the tasks are split evenly between all team members to reduce duplicate changes and increase the total number of changes.

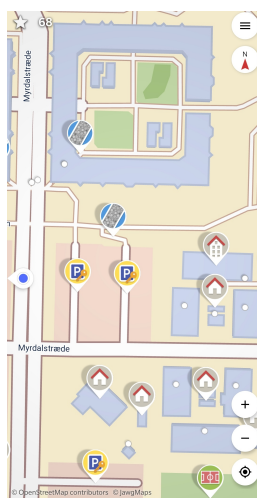


Figure 2.4: The StreetComplete map screen

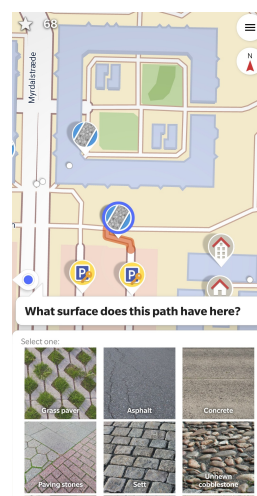


Figure 2.5: Selected quest for determining the surface of a path

2.5.2 Urbanopoly

Another example is Urbanopoly [13]. Urbanopoly is more like an actual game, as it is a “game with a purpose”. The players are playing the game because they want to, due to the game being entertaining and not because they wanted an easy way to contribute to OpenStreetMap. Urbanopoly was developed as part of a research project within Human Computation, which is when humans perform necessary tasks for computers, tasks which the computers are unable to perform themselves.

There are two metrics to verify the effectiveness of games with a purpose. These are throughput, defined as the average number of problem instances solved per human hour, and average life play (ALP), defined as the average time each player dedicates to the game. Focusing on this ALP is essential as this is the player base, and without any players, the throughput is irrelevant.

2.6 Gamification

As mentioned in Section 2.5, gamification is the process of using video game elements outside the context of games. In this section, we delve further into some of the design considerations undertaken when developing tools through the means of gamification.

2.6.1 Game Elements

We present the Bohyun Kims *MDA* framework[14], in order to understand how gamification works. This framework encompasses the three design processes, *mechanics*, *dynamics* and *aesthetics*, where each of these processes produces different game elements. The game mechanics are references to different behaviour and mechanisms that a player can be directly impacted by [14]. Leaderboards, quests and other specific elements belong to this category. The aesthetics fall under the category of creating goals and making the games entertaining when playing them, such as fellowship – making the game social and encourage communication – and discovery – making the game such that it takes the player into uncharted territory [14]. The game dynamics support the game aesthetics, for instance having win conditions that are easier to reach through team play or increasing the points given depending on how rural the area is [14].

2.6.2 Reason for doing gamification

The reason why gamification works is that it brings motivation to the people using the application. Bohyun Kim splits motivation into two categories, extrinsic and intrinsic, where extrinsic motivation refers to external rewards such as points, and intrinsic motivation refers to doing an activity because it is enjoyable [14]. People using a gamified application are often extrinsic motivated since there are rewards associated with the task. However, there are exceptions where the users are intrinsically motivated[14] For instance, people enjoy contributing to OpenStreetMap through StreetComplete. Since there are multiple ways of motivating people to keep playing the game, it is essential to tailor the application according to the target audience because this can determine how successful the application is.

However, there are also some problems when trying to motivate people to keep playing or encouraging them to do specific tasks. One of the problems when controlling them through positive feedback or encouragement is that it leads to the feeling of being manipulated. As a result, the users will disengage from using the application, and they would most likely not reengage [14]. Another problem is the lack of a clear goal with the application, for instance when the goal is to increase the total number of contributions or to increase the map-data quality through team play. This could lead to an application that achieves neither of the

previously mentioned goals or achieves both goals in a mediocre way, which makes the users disengage from the application.

2.6.3 User Types

In order to better design a game experience that caters for a specific type of player, players can be divided into groups that have certain preferences in regards to what they wish to get out of a game. One way to do this is by classifying them using Bartle's taxonomy of player types [15].

Bartle's Player Types are a way to group players of an online game into four categories: Killers, Achievers, Socialisers and Explorers. This is done such that development decisions can be consciously made to cater to a specific group, and the potential consequences and impact of such a choice on the playerbase become better known. At the same time, Bartle's player types are less relevant for the topic of gamification, as they are mainly meant to describe online players. Andrzej Marczewski redesigned Bartle's model such that it better fits the players of a gamified application. To this avail, he proposes six player types to better describe the motivations of players engaged with a gamified system. These can be taken into account when designing the system, resulting in a better experience [16].

The six types of users of gamified systems that Marczewski proposes are:

- **Socialisers:** Motivated by *Relatedness*, socialisers want to interact with other users and create a social experience through the system. This is done by either meeting new people through the system or using it as an excuse to talk with other people.
- **Free Spirits:** Motivated by *Autonomy and Self-expression*, free spirits wish to create and explore using the system. This often means not following the exact set guidelines and seeking challenges outside of what the system offers at surface value while exploring the system. Free spirits are the type of users who will have the fanciest avatar and create the most personal content in an effort to build new things.
- **Achievers:** Motivated by *Mastery*, achievers look for new things to learn and challenges to overcome. They want to learn everything about the system and then apply it to "play" the game to their utmost ability. The person at the top of a leaderboard is usually an achiever.
- **Philanthropists:** Motivated by *Purpose* and *Meaning*, philanthropists are altruistic in their intentions, wanting to give to other people and enrich their lives. In the case of a gamified mapping system, this type of user is most likely already using some type of mapping tool prior to finding the gamified system, simply because they wish to contribute regardless of getting anything in return.
- **Players:** Motivated by *Rewards*, players will do whatever is required to collect rewards from a system. They are the group most willing to play the game and are in it for themselves.
- **Disruptors:** Motivated by *Change*, disruptors are unique to gamified systems in the sense that they are players who do not wish to "play" the game. Instead, they seek to disrupt the system, either directly or through other users in order to force a positive or negative change. In the case of a mapping system, disruptors would be people who purposefully submit wrong information about the mapped area.

While *Players* and *Disruptors* are both extrinsically motivated by rewards from the game or other players' frustrations, the four remaining user types are intrinsically motivated,

meaning that they are motivated by factors not necessarily provided directly by the game, but factors outside of it such as interacting with other players, helping other people, or achieving mastery of the system for their own sake.

It is nevertheless possible to design the game to cater differently to these four groups. For example, socialisers would enjoy a game that empathises teamwork and communication, despite these not being direct requirements to play or “rewards” from the game itself. Likewise, a leaderboard to keep track of the top-performing players in the area adds little to the game itself but allows achievers to keep track of how they fare compared to the rest of the playerbase.

2.7 Problem Statement

We define a problem statement based on the problem analysis. The goal of the rest of the report is to provide a solution to the problem statement. In the problem analysis, we explored what kinds of contributions exist and who already contributes to OpenStreetMap. We found that most contributors are not very active, and few are very active. We want to make contributing more appealing for those that only map a little, or people that may not even know about mapping. We also explored existing solutions and how gamification can motivate users. StreetComplete makes it easy for Android users to make a lot of bite-sized contributions. We would like to further expand on this principle and use gamification to motivate users to cooperate and compete to contribute. By cooperating, users could contribute through social activities and invite friends new to OpenStreetMap to contribute.

How can we develop an application that utilises gamification to increase contributions to OpenStreetMap, both by motivating mappers to map more and by onboarding new people to mapping?

Chapter 3

Design

In this chapter we describe the design of MapTogether, as well as the process we use for making the design decisions throughout the project. We also set forth the requirement for MapTogether for how we can realise the propositions for the final implementation.

3.1 Audience

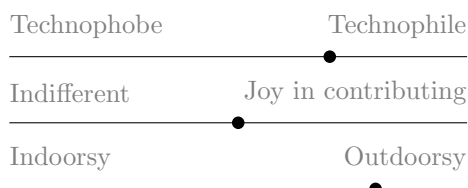
In this section we look at the potential audience, and how we can cater a solution to them. The potential users can provide requirements and prioritisation of features. This work is targeted towards groups of young teenagers as well as adults who likes outdoor activities and want to stay active both health-wise and socially. This could be scout groups or families that play outdoors together. The purpose of this application is to benefit the OpenStreetMap community and developers who use the OpenStreetMap data. It seems logical to include features that make it attractive to people that are already familiar with OpenStreetMap. Experienced mappers know more about what kind of features can be helpful for them to do more for OpenStreetMap, however, as mentioned in the problem statement, we also want to introduce OpenStreetMap to more people.

In order to gain a perspective from users, we develop three personas. These are fictional characters and not real people. Any resemblance to real people is coincidental. The personas should be as realistic as possible and resemble people with different perspectives and needs. Constructing these personas gives us insights into a number of requirements and their prioritisation. All personas have general information about themselves, a motivation, a story related to the topic, and some questions they might have about the application.



Daniel

- 16 years old
- He is a troop scout
- He loves to be outdoors
- Has never heard of OpenStreetMap before
- Daniel uses an Android phone
- From Denmark



Daniel's Situation

Goals, motivation:

- Planning activities for the local scout group
- Doing good for the community
- Getting better quality of local maps
- Getting new and better experiences with the scouts

Daniel's Story

I joined the scouts at age 7 after seeing the fun my friends had, and I have been with the local scout group since. Lately I have been getting more responsibility, and have to plan a few sessions with my friends, but have been struggling to find new and exiting things to do, instead of doing the same things.

Questions

- Will our contributions be used?
- Do all my friends have to create an account?
- How many/few can participate?
- Can I change the activities, so that it doesn't feel the same every time?
- Will the application work when we don't have internet?

As a scout, Daniel is doing activities with friends outdoors, but he is also inclined to help the local community. Daniel wants to be helpful, so the contributions should matter. He is concerned if everybody in his group is able to join the activity, since it might be a large group. Since Daniel is a scout, the activity should work in remote areas with bad connection, such that mapping in places where phone signal don't reach won't become an issue. As Daniel plans activities for his scout group, he might want the activities to vary to keep them interesting: It would increase the longevity if the activity could be customised or changed.



John

- 42 years old
- Father of three kids (12, 14 and 17 years old)
- Goes for walks and small trips with his family
- Has never heard of OpenStreetMap before
- From Canada



John's Situation

Goals, motivation:

- Having a family activity
- Spending time outdoors
- Staying healthy and active

John's Story

I like spending time with my family doing different activities, like playing board games or football in the garden. I try to get my children to do outdoor activities to stay healthy, but it is not always easy. I know one of my friends does something called geocaching and it sounds interesting, I would like to try it.

Questions

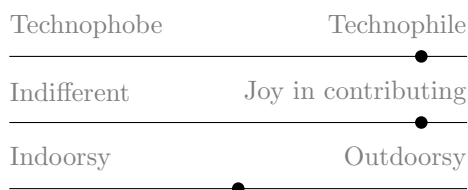
- Is it complicated to add data?
- How long time does an activity take?
- Will all my kids be able to use the application?
- Does it work on my kids different devices?

John cares about staying healthy, and wants to do outdoor activities with his family. He is not particular good with technology, so starting an activity should be very simple, with minimal setup and registering. The activity itself should be fun for his children, and maybe also possible for John to do alone.



Tsu

- 23 years old
- Uses StreetComplete from time to time
- Uses Vespucci to edit when StreetComplete cannot do the task
- Loves to explore her city and cities she travel to
- From Taiwan



Tsu's Situation

Goals, motivation:

- Wants to get friends into the OpenStreetMap community
- Wants to contribute to the OpenStreetMap project more

Tsu's Story

I really like the OpenStreetMap project and what it is doing. It is a good cause that more people should contribute to. I like using StreetComplete because it's more fun than using a normal mapping app, but it's annoying having to switch to other editors. I often hang out with friends, but most of them are not that interested in mapping.

Questions

- Does the application allow me to edit more than StreetComplete does?
- Can I play with people in different timezones?
- Does the edits always appear on my own account?

Tsu is already an adept mapper, and knows existing applications for contributing. She receives great joy in contributing and seeing her growing list of past contributions. Because of this, the contributions added through the activities should be linked to her own OpenStreetMap account. She might also appreciate a leaderboard (or similar) feature, to compare her contribution score with friends and others.

The provided personas give us ideas about what types of features can be useful and what requirements are essential. We will now look at what specific requirements we should satisfy and what features should be implemented.

3.2 Requirements

Looking at everything brought up in the problem analysis, we derive some requirements. The different requirements are not of equal importance. For this reason, we prioritise them using the Moscow prioritisation technique[17]. This involves labelling the requirements with either *must-have*, *should-have*, *could-have*, or *won't-have*. The must-have requirements are the most important ones. If even one of them is not fulfilled, the application will be considered a failure. The should-have requirements are important, but not necessary for the project to provide value. Could-have requirements are desirable and could improve the

user experience or customer satisfaction, however, these will only be implemented if time permits. The least important requirements are labelled as won't-have. These are the ones that simply are not important enough to even consider in this project. The requirements are given in the following list with the most important ones first.

Must Have

1. **The users must be able to maintain existing map data by tagging** like, for example how it is possible in StreetComplete, where users can update the opening hours of shops or the surface material of a road.
2. **The application must not incentivise the users to add incorrect data** since that is the opposite of the intended goal of the application.

Should Have

3. **It should be possible to add new POIs** because users like the persona Tsu, who already uses StreetComplete, misses this functionality and finds it annoying that she should switch to more complicated editors to add new POIs.
4. **The application should work on major phone operating systems** because all three personas want to do the task with other people. An example is the scout Daniel, who lives in Denmark, where 47% of the population¹ uses iOS, and the rest use Android. Daniel will not choose the mapping as an activity for the scouts if half of them are unable to participate due to the application being incompatible with their phone.
5. **The application should use gamification elements** to motivate users, since contributing to OpenStreetMap is generally a mundane task and only provides intrinsic motivation.

Could Have

6. **The application should use social features** to incentivise users to map together and invite new people to contribute.
7. **The user interface could be similar on different platforms**, meaning that people experienced on one platform can still teach people on other platforms. All the personas want to spend time with other people, and the lower the barrier to entry, the less effort people have to put in to get other people to map.
8. **The application could be usable even without permanent access to the internet.** Contributions could be saved locally and uploaded later, when an internet connection is available. This is an important requirement for users like Daniel and the other scouts, who sometimes go to places where no internet is available. Furthermore, it will make it possible for everybody else to map when being abroad where they may not have the same possibility to stay online as when at home.

¹According to MereMobil.dk who got access to data from Gfk <https://meremobil.dk/2020/01/android-vs-iphone-tallene-du-ikke-maa-se/>

Won't Have

9. **The application will not have support for multiple languages.** Multi language support is important to allow as many people to map as possible, but we deem that this is not as important or interesting in the earlier stages.

3.3 Game Design

With the requirements in mind, we define the core experience the user should have when using the app. Designing the experience can be helped a lot by working along with a framework for game design. One such option is the idea of thinking about the gameplay as nested loops of recurring action [18].

Extra Credits made a video on Youtube about game loops “The Real Core Loop - What Every Game Has In Common”². Figure 3.1 shows the game loop presented in their video as a graph between different phases. The game loop is an abstract model of what a player does when playing a game. First, they define some goal/objective, and then they gather information on how to achieve this goal, then they develop and test a hypothesis. Finally, depending on the success or failure of such a test, they either define a new objective or gather new information to test a new hypothesis.

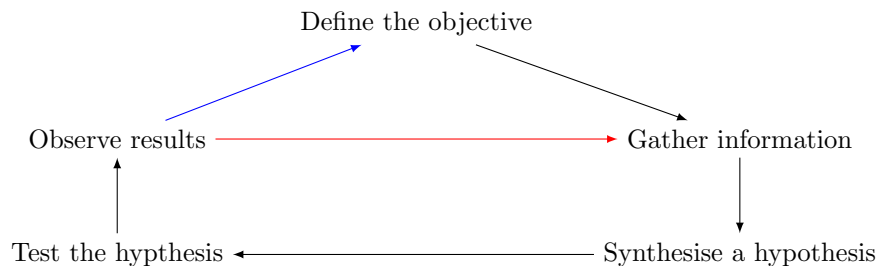


Figure 3.1: Visual interpretation of *Extra Credits*’ game loop from “The Real Core Loop - What Every Game Has In Common”.

If we look at StreetComplete, we can see that one of the smallest core loops in the gameplay is picking a quest, and then answering the question to solve it (as can be seen in Fig. 3.2). This base loop is quite simple and can be made a joy to use by applying various design techniques to make it feel good to do, like applying animations and sounds to the actions.

This core loop itself is not enough to keep users attention for a long time, and some other type of motivation is needed. This could be an extrinsic motivation like enjoying contributing to OpenStreetMap because it is a shared owned community resource. Far from all players will be motivated by this alone, and we see that StreetComplete does not rely on this either because it has an outer game loop in its achievement and leaderboard system.

²<https://www.youtube.com/watch?v=mGL5YGcAxEI>

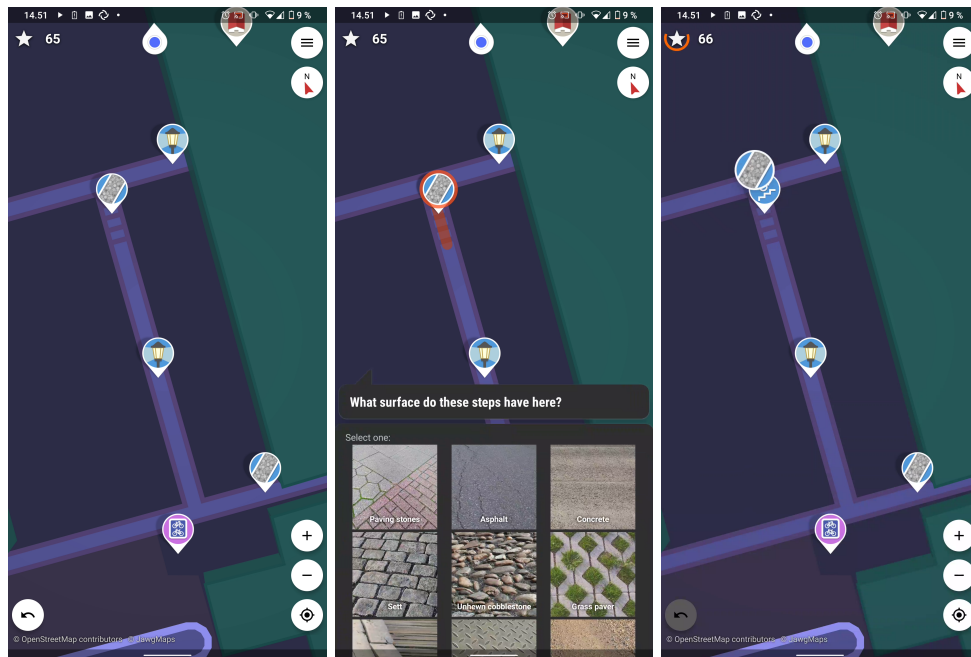
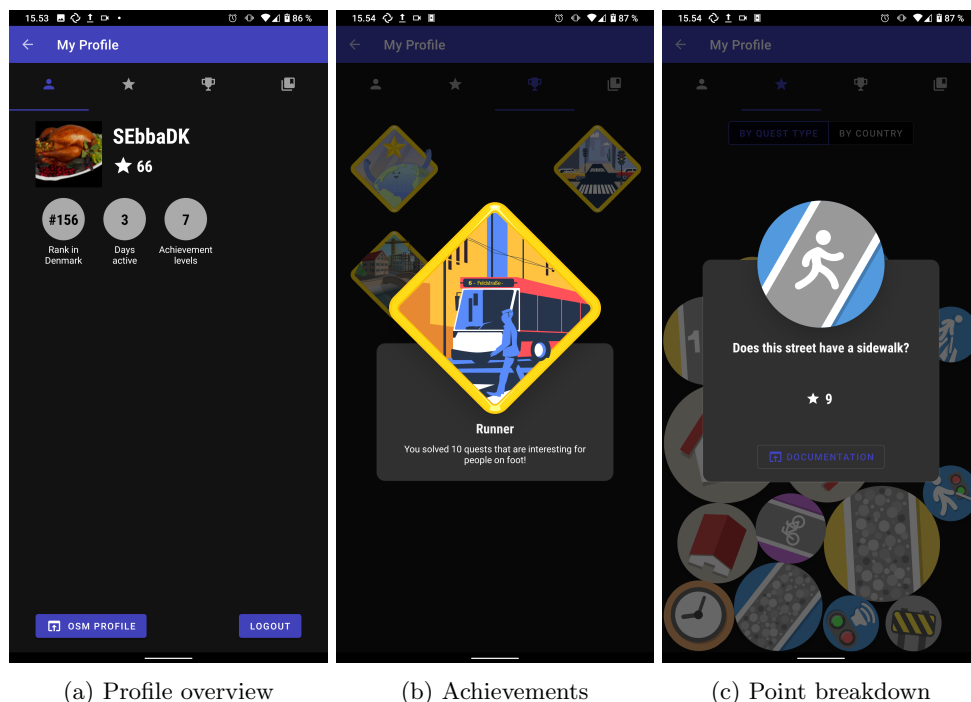


Figure 3.2: Solving a quest in StreetComplete



(a) Profile overview

(b) Achievements

(c) Point breakdown

Figure 3.3: Three of the screens used for the outer game loop of StreetComplete.

StreetComplete makes use of outer game loops to create long term goals for the player. The profile screen of StreetComplete (Fig. 3.3a) contains a view of the users' placement in their regions leaderboard. The leaderboard position is never shown graphically, and it is not possible to see the other users on the leaderboard. Furthermore, the user has to go into

a two-level submenu to see the leaderboard position, which is not optimal for making sure the user always has a set goal in mind. We can make the objective definition part of the game loop easier by displaying the users ranking more prominently. We can also allow for an intermediate loop by showing all the users on the leaderboard, so the player can have a subgoal of “passing player X” instead of relying on a more general “climb the ranks” goal. Especially the “climb above player X” and the “get to the top of a list of named players” motivations play into the Achiever archetype. An example of the loops can be seen in Fig. 3.4.

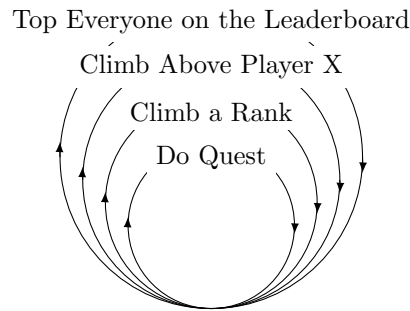


Figure 3.4: The competitive game loop of visible and named leaderboards. It can be even more motivating if X is someone the player has some strong reason for being competitive with (i.e. a friend).

3.3.1 Achievements and Player Profiles

Another thing StreetComplete does for player motivation is the achievements screen, as depicted on Fig. 3.3b, where the users will hit milestones along their mapping. An example is the “Runner” achievement, obtained for solving quests that help people on foot, such as questions about sidewalks or paving material. StreetComplete does not show the achievements in advance, which means they always come as a surprise for players, which can be beneficial in providing a reward, but for the players that associate with the Achievers and Players types mentioned in Section 2.6.3, it might not be the best choice. As seen in Fig. 3.1 the first step in a game loop is to define an objective, which is impossible for Achievers and Players that want specific achievements if they are hidden. Furthermore, it is impossible to see other users’ achievements, so there is no way to show off your own achievements other than in person or share them with a screenshot. A possible way to improve on this could be to make player profiles public and allow users to customise their public profile through banners and selecting achievements they want to display. This would lead to a game loop as seen on Fig. 3.5.

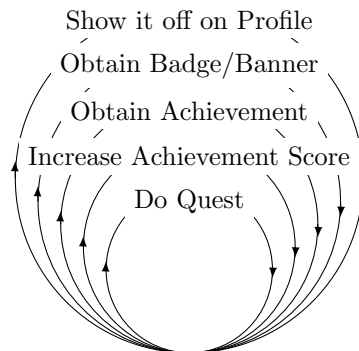


Figure 3.5: The achievement game loops in a system where achievements are not only surprises and profiles can be viewed and customised.

3.3.2 Time-based Leaderboards and Brackets

StreetComplete uses all-time regional leaderboards. Unfortunately, this means it might be tough to climb in the rankings because the person ranked one place before, might have played the game for a long time, and have such a massive lead that overtaking seems impossible. We can work around these massive leads by employing time-based leaderboards, as it has been done in other gamified processes such as the language-learning platform Duolingo³. Duolingo makes heavy use of a week-to-week leaderboard and encourages climbing the ranks even more by splitting up players into “leagues” or brackets. Hence, the learners only compete with other learners who normally get a similar amount of points each week. Furthermore, at the end of every week, the top learners in a league are promoted to the next league, and the bottom learners are demoted. Therefore, getting promoted to higher leagues would motivate especially Achievers. This weekly loop is shown on Fig. 3.6. Employing a similar system might net a big effect on creating many nested game loops and making it easy for players to define their goals, so there is also something to work towards when they are out mapping.

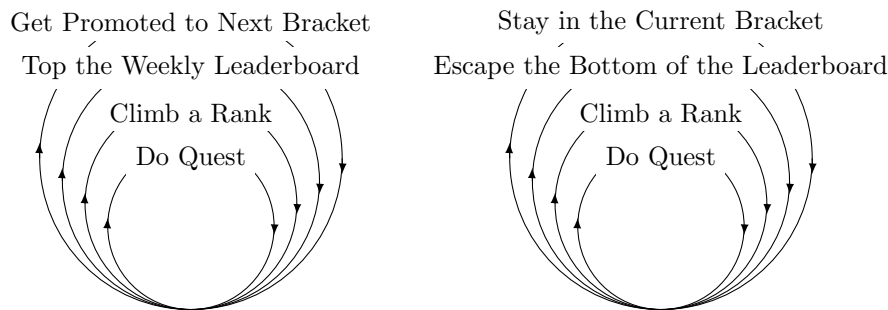


Figure 3.6: The game loops of a bracket based weekly leaderboard system for mapping.

3.3.3 Socialising and Gamified Mapping Parties

All of the mentioned features lean mostly into the things that attract Players and Achievers. If we use design elements that attract the other player types, it might help the app reach a wider audience and increasing the amount of mapping done. One of the player types not directly touched by StreetComplete (until recently) is the Socialisers. However, as mentioned in Section 2.5.1, StreetComplete introduced a feature to split quests up among mappers walking together physically. This is possibly to make StreetComplete nicer to use for mapping parties, which are events where people meet up and map together⁴. It seems likely that the people meeting up to map together would fit with the Socialiser archetype, and since mapping via StreetComplete often requires being (or have been) at the location, mapping parties seem like the best way for Socialisers to map. While StreetComplete now has better support for mapping together, it is not gamified in any special way compared to the solo mapping experience. One way to change this is to create motivating game loops around doing activities together.

One idea for gamifying the mapping parties can be derived from the scouting activity of walking to some different points and completing tasks at those locations. Basing the gamification off scout activities might make it natural to use MapTogether in the scouting context Daniel would use it in. The activity could be structured around going from one location to another and completing some specified amount of mapping in each place. This would create an overarching loop, finishing a location being an inner loop of that and finishing a single quest being an inner loop of that as shown in the leftmost game loop on Fig. 3.7.

³<https://www.duolingo.com/>

⁴https://wiki.openstreetmap.org/wiki/Mapping_parties

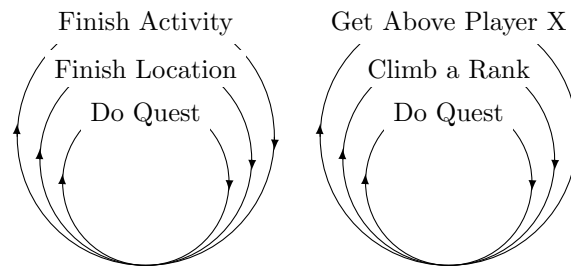


Figure 3.7: The game loops of the multiplayer mapping activities. The leftmost being the immediate goal of completion, and the rightmost being the possible competitive goals if a activity-leaderboard is introduced.

Another feature that might encourage the social player type is groups. Players could form groups where their contributions could be combined in a joint effort. Groups could possibly compete with other groups in some common score between the group members. The competition would then be more a cooperative experience which could be good for the social player type.

3.3.4 Gamification of Surveying

The previous design ideas we have been describing have primarily been quests that exist in StreetComplete. The quests are only map maintenance tasks through extensive and updated tagging and not the creation or removal of points and ways. Surveying is harder to define as a set of quests because there is no way to know for certain if a blank area on the map is unmapped or physically empty, other than having a person survey the area.

StreetComplete does not have the ability to create or remove details on the map beyond tagging, which means it is a task normally delegated to more powerful mapping tools. This breaks with the gamification elements since the user is not rewarded for mapping outside of the app, and it also adds a hurdle for the user to have two different tools needed to map. Therefore, adding the ability to map new things may improve the mapping. However, without any gamification elements, that type of mapping might only be attractive to the Free Spirit and Philanthropist player types.

Ensuring that the player can define a goal when surveying means that we need to help them break down the “survey the world” goal into smaller tasks. One way of doing this is by decreasing the region’s size that the player aims to survey. OpenStreetMap contains geographical subdivisions. Many of them are things like nations, regions, city limits or a neighbourhood in a city. This is a great reduction in size, but it can still be very overwhelming for the player to survey such large areas and having a quest asking “Is Aalborg mapped completely?”, which is impossible to answer due to the scale. Therefore, we need to do further subdivision.

We can see in Fig. 3.8a that dividing the map into grid-cells of a size which a single person could survey in around a minute could allow for queries easier than surveying an entire neighbourhood at once. The problem with grids can be seen in a cell, like the top left square of Fig. 3.8a, where a building splits the two halves, and to get an overview, the player is required to walk around the building. The problem can be overcome by using an alternative way to subdivide the map like shown in Fig. 3.8b.

To further gamify surveying, the community-survey status of the tiles could be visible to the player with colouring or the likes. This could also help the player decide what to do next,

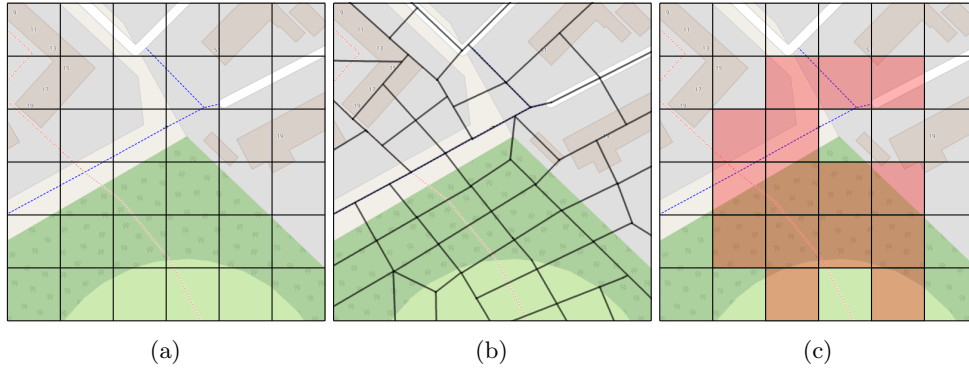


Figure 3.8: Two examples of subdivisions of the map for surveying, and an example of a partially surveyed area. The left-hand side figure shows a evenly split subdivisions, and the middle figure shows better subdivisions

as they could pick an area with more unmapped tiles, which could also come with a higher reward, to encourage surveyors to map areas that are not well mapped instead of focusing on the frequently used and mapped areas.

Having a map with changeable tiles like in Fig. 3.8c might also offer the motivation of completely filling in an area. However, painting in the area is not all: it is easy to imagine the map as a canvas, which opens up motivations for carto-vandalism, as explained by Ballatore [19]. For our case, we assume that free-spirits using the app might engage in the form of artistic carto-vandalism of painting things with these tiles. Contrary to changing the map directly, this form of “painting on the canvas” is not only not detrimental, but a positive force if the users actually do surveying to paint. Carto-vandalism only becomes problematic if the users fakes the survey-questions in order to paint on the canvas resulting in tiles being marked falsely as surveyed. We can mitigate this carto-vandalism in multiple ways. We can show the the player only tiles they have surveyed themselves, thereby removing the problem of having falsely-marked tiles. Alternatively, we can avoid letting a single user mark a tile as surveyed, thereby relying on different users not having the same aligned vandalistic goals.

A player type that is incredibly difficult to accommodate is the Free Spirit. This is due to their nature of exploring the system and finding their own use instead of following the usual loops of the game. Due to their way of playing, any feature or change made to accommodate the free spirits is unlikely to find great reception as free spirit players find their own meaning with the game instead of accepting the one put in front of them.

The last player type we need to consider is the Disruptors. We account for these players differently than the other player types, as they can be counterproductive to the purpose of MapTogether. The issue with Disruptors is that they might contribute bad data to OpenStreetMap though MapTogether, which is troubling as this is directly counter to the purpose of MapTogether. Because of Disruptors, it is a good idea to validate the contributions and perhaps players through MapTogether, and not trust the info provided without having multiple players validate it. ,

3.4 Features

Now that we have the design mostly specified, we make a plan for the features that are most important. To start with, we define which features depend on other features to see how coupled they are. When we plot it as a directed graph in Fig. 3.9, we see that some features

are necessary for nearly all other features.

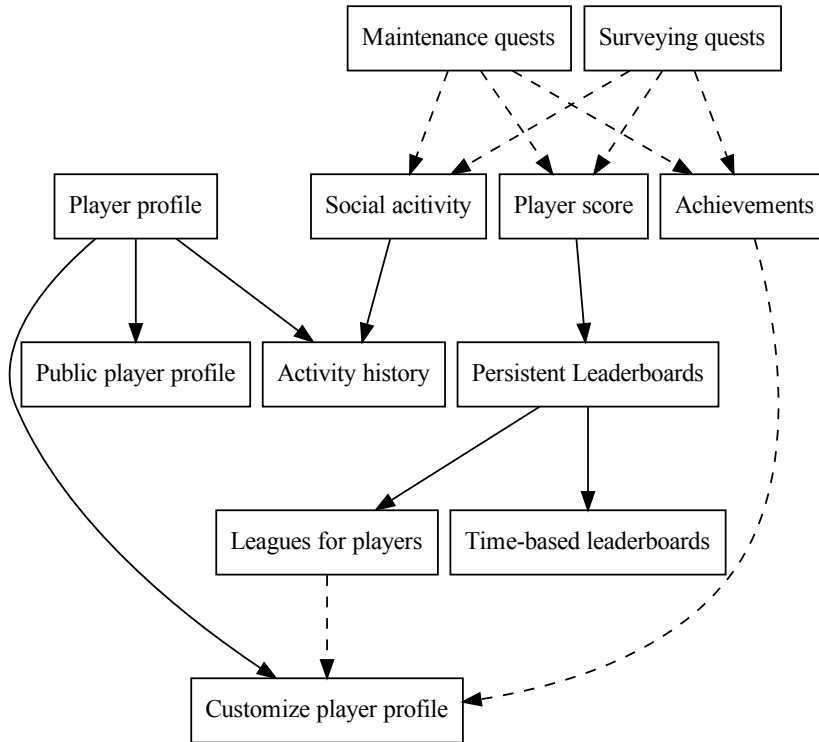


Figure 3.9: The features and how they depend on each other. Arrows denote that features cannot be completed until all incoming arrows are implemented. Dashed arrows mean at least one dashed dependencies must be fulfilled.

Maintenance quests are simple tasks like those in StreetComplete that can easily be completed.

Surveying quests are tasks where players must “complete” a small zone on the map as described in Section 3.3.4

Player profile shows information about a player and their friends. It can also contain information about achievements and more if those features are implemented.

Social activity refers to players completing quests together in the same area and competing for the most points.

Player score is a players total amount of points attained from completing either type of quest.

Achievements are awarded for milestone accomplishments such as completing a certain amount of a quest type.

Public player profile is the player profile information that the player decided to share publicly, such as specific achievements, leagues or leaderboard positions.

Activity history is a log of activities that the player has taken part in.

Persistent leaderboards are leaderboards that list players' total points within a parameter, such as a global leaderboard or national leaderboards.

Leagues for players are time-based leagues, where at the end of the given time period, top players in a given league are promoted to a higher league, and the bottom players are relegated to a lower league.

Time-based leaderboards are leaderboards where only points gained within a certain time frame are counted.

Customise player profile is the ability for players to customise what is displayed on their public profile, based on leagues, leaderboards and achievements.

There are two key features: *player score* and *social activity*. Scores open up for having leaderboards and motivates competitive users. Social activities will make MapTogether stand out and motivate people to invite their friends to use it. To implement these features, we first need to have some sort of quest/task implemented. We divided this into maintenance quests (much like StreetComplete) and surveying quests, as explained in Section 3.3.4.

We start by implementing some simple maintenance quests, so social activity and player score systems can then be implemented afterwards. More complex and diverse maintenance quests can be added later. Before we can implement any quest types, we need to implement a way to talk to the OpenStreetMap API⁵, to download raw map data, upload contributions and log in to their OpenStreetMap account through MapTogether. In order to implement the social activity mentioned in Section 3.3.3, MapTogether needs to be able to communicate between devices.

Player scores should be trivial to implement, but before we can implement persistent leaderboards, we need to set up a way to store and synchronise player data – such as the player's score – between different users. The leaderboard would then be calculated based on the synchronised data. While implementing persistent leaderboards, we also work on the player profile, which should be updated when more features are implemented. After that, the rest of the features can be developed in parallel and almost in any order.

3.5 Object-Oriented Design

In this section, the solution is designed according to methods from *Object-Oriented Analysis & Design* (OOAD) [20]. This is a method for design a complete application, from the initial idea to the complete design. We will, however, start from a later point since we already have an idea and the target audience. The parts after the initial idea phase that we will use are the problem domain and the application domain. First, the problem domain is analysed to understand the scenario that the system will imitate. After this, the application domain is analysed to understand how the target audience interacts with the system.

Before we start the analysis, we must perform a FACTOR analysis. Then, the FACTOR analysis model is used to determine the necessary criteria for a system definition to form a concrete system definition.

⁵https://wiki.openstreetmap.org/wiki/API_v0.6

- [F]unctionality: Support for adding and editing OpenStreetMap map data.
- [A]pplication Domain: The players who contribute data to OpenStreetMap.
- [C]onditions: Developed through requirements from the target audience.
- [T]echnology: Android and iOS phones. A database running on a server.
- [O]bjects: Users, OpenStreetMap map data, contributions.
- [R]esponsibility: Supports the addition of nodes and edits of current map data. Automatically generates quests and distributes them among the participants.

System definition A mobile application for interacting with OpenStreetMap, where tasks are distributed over the map and these quests lead to OpenStreetMap contributions. Opportunity for social interaction and competitive gameplay with friends through the multiplayer nature of the application and its leaderboards. Creating and configuring the game mode is also done through the mobile game, which is based on either Android and iOS with internet and GPS capabilities. The development process should be conducted based on the requirements from Section 3.1.

3.5.1 Problem Domain

In the following section, the classes, events and structure of these classes are identified. Events can occur several times for each class. We consider if an event can occur once or multiple times for each class. This information is gathered and represented in an event table.

Classes are a fundamental part of the problem domain, and identifying classes is a way to begin forming an overall understanding of the problem domain. A class is an abstract representation of similar objects, which share behaviour and attributes. The first step is to find the class candidates. To maximise the different perspective of the problem domain, it is beneficial to have a long and varied list of class candidates. To have an abbreviation of the candidate list, some of the classes in the class candidates list on Table 3.1 are already collections of objects.

Quest	User	Activity
Leaderboard	POI	Group
Waypoint	Message	

Table 3.1: Class candidates

These candidates have been classified in Table 3.3. Some of these classes encompass multiple different objects. For instance, there are multiple different quests, but there is no difference in these quests from the systems point of view. The *Activity* class contains all the data related to a specific activity, such as the users partaking, the quests in MapTogether, the game mode and the waypoints related to the activity. These waypoints are indicators on the map, and the *Waypoint* contains its location as well as the quests associated with it. The waypoint itself is also a part of a game. The *POI* class revolves around a location and the information about that specific location, for instance, a bench and the materials it is made of.

After identifying the class candidates, the event candidates must be identified. An event is an instantaneously occurring action, which is experienced or performed by objects in the problem domain. Finding event candidates constitutes the next part of getting an

Activity created	Quest added	Group invite received
Activity started	Quest completed	Group invite accepted
Activity cancelled	User signed up	Message sent
Activity completed	User completed waypoint	Message received
Activity invite sent	Group invite declined	Message deleted
Activity invite received	Group invite sent	POI added
Activity invite accepted	Group created	POI deleted
Activity invite declined	Group joined	POI updated
Followed	Group left	
Waypoint visited	Group deleted	
Waypoint completed	Group changed	

Table 3.2: Event candidates

understanding of the problem domain. To find the event candidates, one could examine similar computer systems or analyse activities from the beginning to the end.

The events in Table 3.2 have been found by looking at the prototypes and system definition. Event candidates are found by using verbs that are used to describe what happens to an object. As with the class candidates, the found events will be evaluated in the event table.

The event table describes the relationship between the evaluated classes and events. The classes and events in the event table have been chosen from the class candidates in Table 3.1, and the event candidate in Table 3.2. The general evaluation criteria used for the classes and events are whether the class or event is within the system definition and whether the class or event is within the problem domain.

After evaluating the classes and the events based on the general criteria, some more specific evaluation criteria can be used. For the classes, the information within them has to be unique while also encompassing multiple objects, they must however not encapsulate too many events. For the event, one has to answer whether they are instantaneous, atomic and if they can be identified when the event happens. These specific criteria then lead to the event table in Table 3.3.

To determine whether an event happens once or multiple times, behavioural patterns are used. Behavioural patterns are used for better understanding the dynamics of the system, by showing event traces. An example of this relation is *Quest completed*. The event *Quest Completed* can only happen once per quest, whereas a user can complete multiple different quests. Completing quests influences both the global and group leaderboards, and it also helps with completing the current waypoint. After completing the event table, a structure of the relations can be made, which is represented in a class diagram.

Structure

In the following paragraph, the relations between the classes from the previous section is inspected. This relation is structured as a class diagram. This class diagram gives a model of the system's problem domain and gives an overview as to how various classes interact with one another. The class diagram shown in Fig. 3.10 represents an early structure of the

Events\Classes	Quest	User	Activity	POI	Leaderboard	Group	Waypoint	Message
Activity created		*	+					
Activity started		*	+					
Activity cancelled		*	+					
Activity completed		*	+					
Activity invite sent		*	*					
Activity invite received		*	*					
Activity invite accepted		*	*					
Activity invite declined		*	*					
Followed		*						
User signed up		+						
User completed waypoint		*	*				*	
Waypoint visited		*	*				*	
Waypoint completed		*	*				+	
Group created		*			*	+		
Group joined		*				*		
Group left		*				*		
Group deleted		*				+		
Group changed		*				*		
Group invite sent		*				*		
Group invite received		*				*		
Group invite accepted		*				*		
Group invite declined		*				*		
Quest added	+							
Quest completed	+	*			*	*	*	
POI added		*		+	*	*	*	
POI deleted		*		+				
POI updated		*		*	*	*	*	
Message sent		*	*					+
Message received		*	*					*
Message deleted		*	*					+

Table 3.3: Event table. The events marked with an asterisks (*) can occur zero or multiple times. The plus (+) marks the events which happen zero or a single time

application and gives us an idea of how to start the development of the application. The classes *Element* and *Changeset* comes from OpenStreetMap. An element represents a node, way or relation. A changeset represents a set of contributions.

3.5.2 Application Domain

In this section, we analyse the application domain, with a focus on identifying the necessary system functions. The functions will be used to design the user interface in Section 3.6. Both the system definition and the problem domain model are used in this analysis.

Functions

These functions represent the necessary functions for the program to achieve the users' goals. These functions have been identified and analysed based on previous information. The complexity of a function is an estimation of how difficult it is to develop. The types of

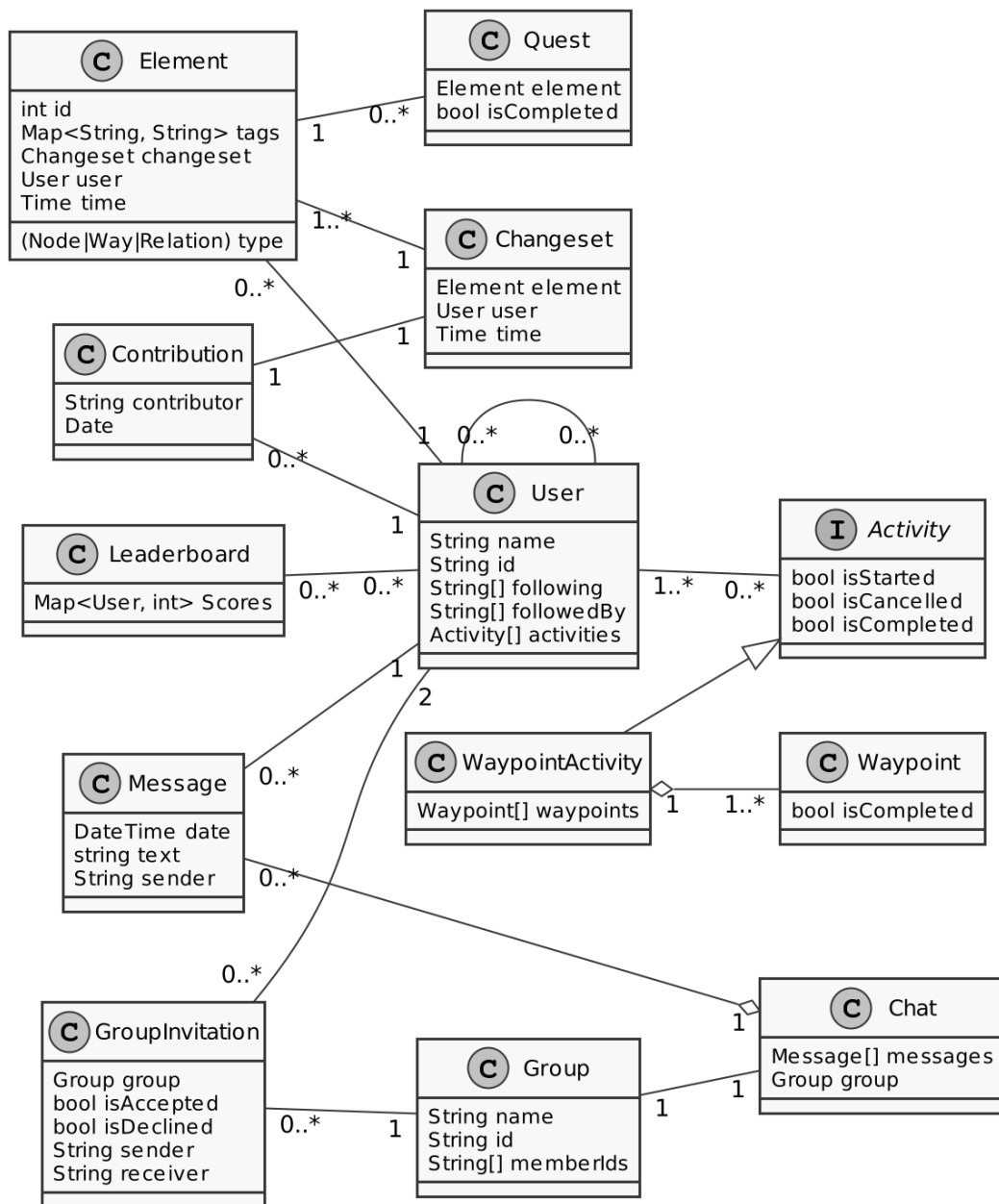


Figure 3.10: Diagram of the classes in the application. These have been found based on the event table

Functions	Complexity	Type
Leaderboard change	Simple	Compute
Find nearby quest	Medium	Compute
Create Activity	Complex	Compute
Answer quest	Simple	Update
Follow a user	Simple	Update
Join a group	Simple	Update
Send invite	Simple	Update
Add point of interest	Medium	Update
Send message	Medium	Update
Complete Waypoint	Complex	Update
Login	Complex	Update
Upload contribution	Complex	Update
Complete Achievement	Simple	Signal
User GPS location	Medium	Signal
View Achievements	Simple	Read
View Leaderboard	Simple	Read
User search	Medium	Read
Show History	Medium	Read

Table 3.4: Functions listed along with their complexity and type

functions are divided into four categories *compute*, *update*, *signal*, *read*. *Compute* is used for functions that compute data before it is either displayed or added to the database. *Update* is used for changes in the system without adding new information. *Signal* is used for functions that monitor a continuous data flow and notifies actors in certain conditions. The fourth and last category is *Read*, which retrieves one or more values, usually to display them in the user interface.

With this categorisation and evaluation of the necessary functions for MapTogether the primary object-oriented design has been laid out, both for the problem domain as well as the application domain. The next step in designing the application is making a user interface in accordance with this design before we implement the application.

3.6 User Interface Design

We have laid out the features of MapTogether in Section 3.4. Now we design the interface to support user interaction with the different available systems and features. We draw simple wireframes in Balsamiq⁶ to visualise and discuss the appearance of initial design ideas.

Balsamiq is a UI wireframe tool designed for quickly creating prototypes for a GUI such that potential problems and shortcomings of the interface may be identified at an early stage of

⁶<https://balsamiq.com/wireframes/>

development.

3.6.1 Main Screen

When opening the application, the first thing the user sees is the main “map” screen shown in Fig. 3.12a. Opening the app initially takes the user directly to a slippy map – which means the map has the ability to zoom and pan is integrated – of their surrounding area, with the functionality to add data or complete quests instantly. Having tasks given to the user from the moment they open the app is done in an effort to hook the user and immediately give them a starting point rather than potentially repelling them from the app, at the prospect of an extensive setup to get started. The user must sign up with an OpenStreetMap account to successfully upload contributions, delaying the process until necessary. MapTogether incentivizes users to try out the application, and make them more likely to then sign up later due to the sunk cost-fallacy⁷. If the user has already completed some quests or added a few locations, it would seem like a waste to not sign up and get points and actually contribute to OpenStreetMap.

Given that the main menu is the first thing the users see, it is also responsible for navigating to the remaining pages containing information that can not be displayed here. For this reason, we put five buttons in the bottom right corner of the screen. The first button (from left to right) opens the social screen described in Section 3.6.2. We choose a profile icon for this button because the social screen is associated with the user profile and interaction with other users. The second button is for panning the map to the location of the user. The third button is used to indicate north, and pressing it, rotates the map so that north is up. The fourth button is for synchronising data with OpenStreetMap so that the user can download the most recent map data. The last button is “Menu”: pressing it brings up additional buttons that allow the users to start an activity or access settings.

Aside from the quests present on the map, the user should also have the possibility of adding Points of Interest (POIs). It is unknown by nature where these should be added, so the user is instead allowed to create them by interacting with the map. Seeing as a single, quick press is used for interacting with quests and dragging is used for panning the map around. It is decided to use a long press on the map, in order to add a new POI at the clicked location. This is not entirely optimal, as the user would require previous knowledge of how to add a POI or find the feature through experimentation in order to successfully interact with it. A solution could be to add a small tutorial, however that is out of scope for the implementation during this project. Given that this mode of interaction provides the least disruption to other features, it is nevertheless used as the method for adding a POI.

Adding a POI

After a long press, a prompt is used to finalise the addition of a POI. This process is shown in Fig. 3.11. In order for a user to add a new POI, they must specify information about the new point. The prompt opens a dialogue, where the user describes the required information in accordance with the real-world information available to the user. For this purpose, we use a slide-up menu with fields to give various info about the POI, such as the name, type of venue and opening/closing hours. Once these fields have been filled out, pressing the button at the bottom of the slide-up menu adds the POI. This should be visualised to the user by instantly adding it to their local map data, such that they are aware the new POI has been properly reported. We can then use the new node as a button to allow the user to edit the data once more, in order for them to correct errors, or delete it in case of misplacement.

⁷A fallacy where someone convinces themselves to continue doing something because they have already put time and/or effort into it



(a) Pop up of POI adder after long pressing on map.

(b) Description pop up upon clicking to Add POI.

(c) POI has been added to the map.

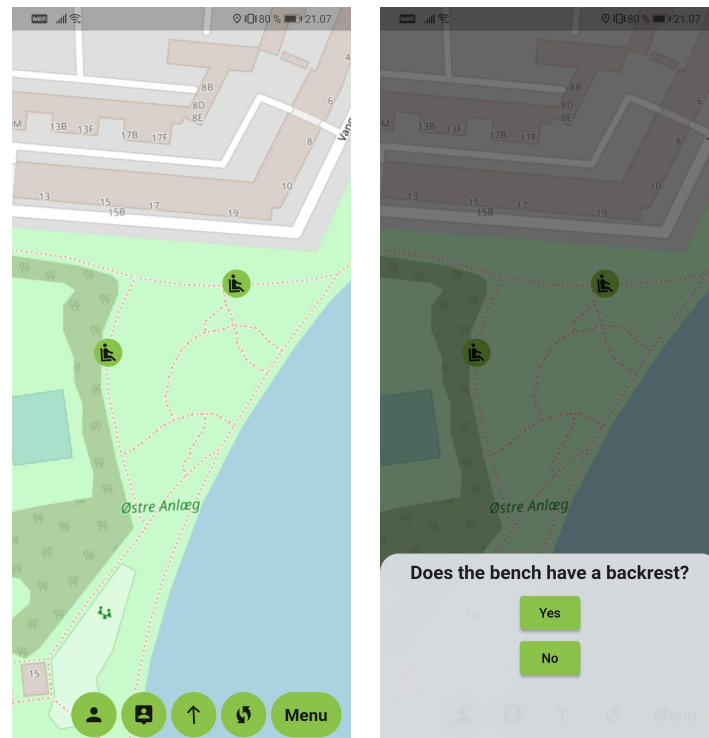
Figure 3.11: General flow for adding a POI on main map.

Doing a Quest

Quests appear on the map as they are fetched from the OpenStreetMap server, much like StreetComplete. Quests have a distinct type of visual identifier, to help the user identify the specific type of presented quest. An example of an identifier that shows a quest to confirm whether a bench has a backrest or not can be seen in Fig. 3.12a. Quests like these appear randomly on the map in the proximity of the player and vary depending on the data fetched from the server. Activating the quest brings up a prompt, as seen in Fig. 3.12b. This prompt allows the user to fill out information related to the specific quest. In this case, the user is asked if the bench has a backrest or not. Upon having filled the prompt, it pops back down and the quest is considered “completed”. This means the quest also disappears from the users’ map to indicate that it can no longer be completed.

Activity

During an activity, users have areas where they need to complete a number of quests to progress to the next waypoint. Starting an activity includes specifying certain conditions such as distance, number of quests and inviting other players. Starting an activity initially has two options: search or create new. Searching shows a list of nearby activities. Pressing a nearby activity automatically joins it. Creating a new activity should have a tab for inviting followers or nearby players, and another tab to specify rules for the activity, and a button to start the activity. Once the activity has started, the current waypoint should be clearly visible with a special marker on the map, so that users know where to go. An arrow points the user in the direction of the waypoint to make it easy. A pie diagram for each location at the bottom of the screen can be used to indicate how many remaining quests the user has at each location before they are ready to move on to the next waypoint.



(a) Bench backrest quests as they appear on the map.

(b) Prompt to fill in information regarding bench backrest.

Figure 3.12: Example of quest displayed on the map.

3.6.2 Social Screen

The majority of the features in Section 3.4 requires some socialising and interaction between the users. Because of this, we have a social screen with the purpose of containing everything in regards to profile and user-to-user interaction. We set up the social screen like a navigation menu that houses a list of MapTogether's social features in a menu bar at the bottom of the screen. The body of the screen is occupied by whichever item is chosen in the menu. It is necessary to be logged in to access the social screen so that data such as followers and groups can be fetched from the server. Through the social screen, a user should be able to access leaderboards, followed/followers, groups and log history. We put these four things in their own tab in the bottom menu.

Leaderboard

As described in Section 3.4, we need to provide a view of leaderboards. The leaderboard should provide users with an overview of their own placements and close competition or the top scores. We would like different leaderboards based on location or time periods, the user should have some menu that provides all the leaderboards they currently participate in.

As mentioned in Section 3.3.2, time-based leaderboards are a way to give players a reachable goal to strive towards, as they will not be overwhelmed by having to climb above a person who has been using the application to accumulate points for years, but rather on a week-by-week or month-by-month basis where anyone can be competitive.

We separate time periods (weekly, monthly, all-time) in their own tabs as shown in Fig. 3.13a. Each tab shows a scrollable list of all the leaderboards that the user currently participates

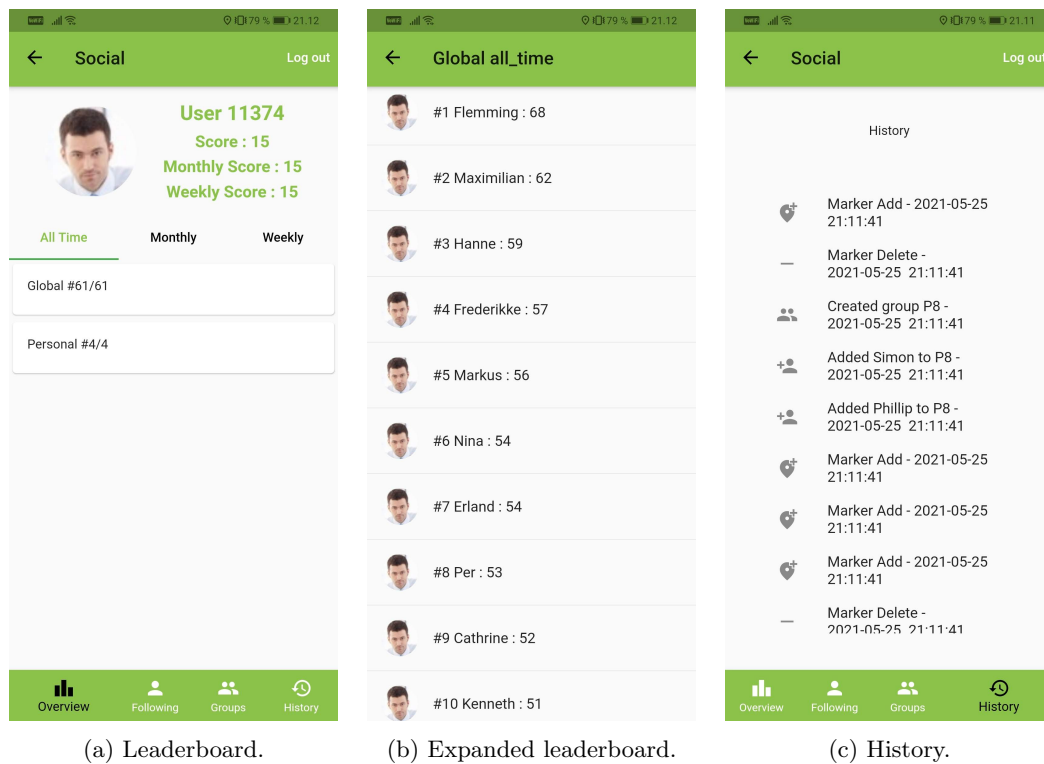


Figure 3.13: Social Menus for Leaderboard and History.

in. Each displayed leaderboard is identified by a title, as well as the current user's ranking and the total number of participants. This allows the user to immediately get an overview of their position in each leaderboard.

Each leaderboard can be tapped, to gain an expanded view of the rankings, score and current positioning on the selected leaderboard. An example of this can be seen in Fig. 3.13b.

At the top of the leaderboard screen in Fig. 3.13a, a small view of the currently logged in player's profile with their scores for the current day, week and all-time can be seen.

Follow List

Per the functions in Table 3.4, users should have a way to follow others and view their profiles. For this, we have a separate menu in the bottom bar: The Follow Screen. The purpose of the following screen is to let users see people they follow and the people that follow them. This is mainly used to keep track of other people such that they can easily be added to a group when creating it, as well as keep track of the scores of the various people they follow. The follow list is unique to each user and displays a list of the people they follow whilst simultaneously allowing them to follow new users or unfollow some of their current following with little effort. The standard view of the follow list can be seen in Fig. 3.14a. Any followee in this list can be selected to get an expanded view of said player's profile, displaying their score for the current week, month and all-time. The player profile also displays said person's ranking on their own leaderboards, such that the current user can see their comparative performance in groups they participate in when inspecting another's profile. Viewing another user's expanded profile is very similar to seeing one's own leaderboard as seen in Fig. 3.13a. However instead containing information about the specified user, as opposed to the one currently logged in. Followers and followees can be

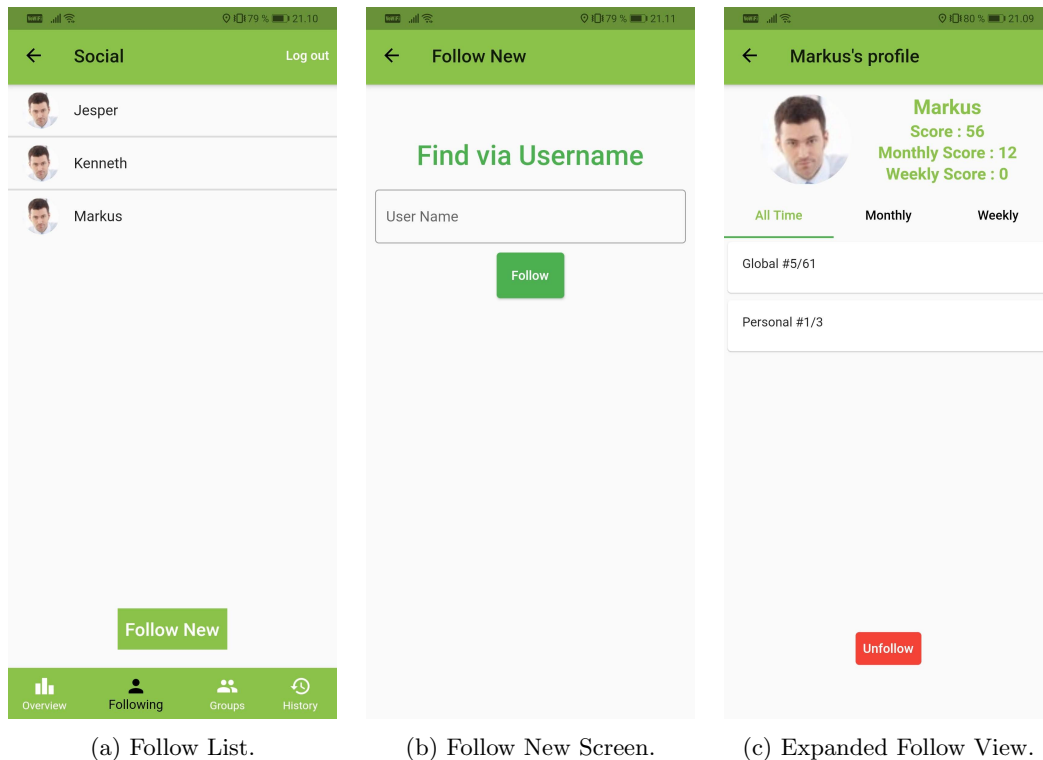


Figure 3.14

separated into two tabs, in order to easily see a distinction. The expanded user profile can be seen in Fig. 3.14

In order to manage the follow list, users need functionality to both follow and unfollow people. The option to unfollow is not readily apparent on the follow list, as unfollowing someone is likely to be a rare occurrence. For this reason, we keep the unfollow button away from the main view of followers, so as to avoid the user mistakenly pressing it. Instead, we put it at the bottom of the expanded follow view as seen on Fig. 3.14. This allows the current user to manage their followings through visiting the expanded user pages on their follow list. Alternatively, the option to unfollow is available by pressing and holding a user for a few seconds. Doing so could give a small pop-up at the bottom of the screen, prompting the user as to whether they want to unfollow the chosen user. There are no indicators to this feature, which means it can be unintuitive for users. However, this is generally not a feature expected to be used often by the majority of users. Some users may wish to frequently clean out their follow list. In this case, unfollowing each followee separately becomes a rather tedious task. This alternative is there to potentially make this task easier for some users.

Adding another user to the follow list is also important. We design two ways to do this. The first is similar to unfollowing from their player profile page. At the bottom of the player profile of any given player not already on the users follow list, the option to add the player is added where the option to unfollow them would otherwise be as seen in Fig. 3.14. This requires the user to find the player profile on their followers list or a leaderboard. The second option is to add players by their username. a “follow new” button takes the user to a screen shown in Fig. 3.14b. The functionality of the screen allows the user to enter the username of a player that they wish to follow.

Groups

Leaderboards are closely tied to the groups in which the user is in. Groups are displayed separately on the leaderboards and could have specific leaderboards where different groups could compete. The groups could be viewed much like the follow list. It should facilitate the creation and deletion of groups. We design it in the same way as the follow list screen described in Section 3.6.2, having a scrollable list of current groups each of which can be interacted with.

Unlike the follow screen, however, users should be able to edit the groups separately as opposed to only having the option to remove them entirely. This means the user should be able to add or remove individual people from a group instead of having to create an entirely new group. To this end, groups have an expanded view that can be accessed by opening them. Within the expanded view, members of the selected group should be clearly visible by any other member of the group. Moderators of the group are also able to invite and remove participants from the group here, should they so desire. The groups created here also appear separately in the leaderboard tab described in Section 3.6.2.

History

We think it would be useful for users to be able to get an overview of their recent activity (follow updates, quests, activities). For this, we create a history menu. The history menu grants the user a view over their recent actions, by presenting them with a scrollable list of their previous actions taken with the app as shown in Fig. 3.13c.

3.6.3 Login Screen

A user is required to log in before they can report their completed quests, as well as access MapTogether's social features. As described in Section 3.6.1, the user should have the ability to complete quests without logging in. This is to let the user start using the app quickly, as well as retain their interest before they commit to signing up for an account. Instead of prompting the user on opening the app, the user is asked to log in when trying to access the social menu without being previously logged in. The user can then decide to either proceed with the login return to the main map without doing so. The user must also receive a login prompt after answering a quest. The users may not be interested in the social features of MapTogether. Therefore they might not realise that their contributions are not being reported, until after they receive the prompt. The login screen itself is a web link that leads to the official *OpenStreetMap* login page.

3.6.4 UI Flow

To illustrate the flow of the app in regards to the design description, we draw a UI flow chart shown in Fig. 3.15. It shows how the different pages direct to one another and provides a general overview of the flow of MapTogether.

The majority of the arrows in Fig. 3.15 are bidirectional because most screens have a back button. This returns the user to the location from which they got to the current screen in the first place. The exception to this is the login screen which automatically directs users back after logging in. There is no reason for users to return to the login screen without first logging out again. The edges around the login require special conditions, which are not directly apparent on the graph. To go from the map to the leaderboard, it is required that the user first logs in. If the user is not previously logged in, the user is lead to the login page, which will then redirect them to the leaderboard, on successful login, or return us to the Map if the login fails or is cancelled.

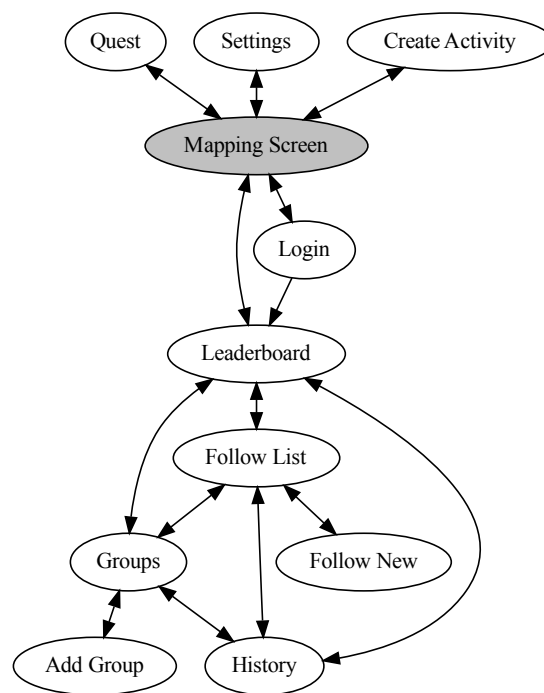


Figure 3.15: The various UI screens and how they direct to one another. The grey circle is the starting point for the application.

Chapter 4

Implementation

In this chapter we describe the implementation details of MapTogether, based on the design from Chapter 3. This includes descriptions of how both the server- and client-side specific details of the design, were implemented in the final prototype. As well as the technologies used therein.

We lay out the foundation for a slice of the full implementation, in order to create a functional prototype within the allotted project time. The slice has the purpose of show-casing the most important features of MapTogether and will include the features shown in Fig. 4.1. We limit this chapter to describe the implementation of the slice.

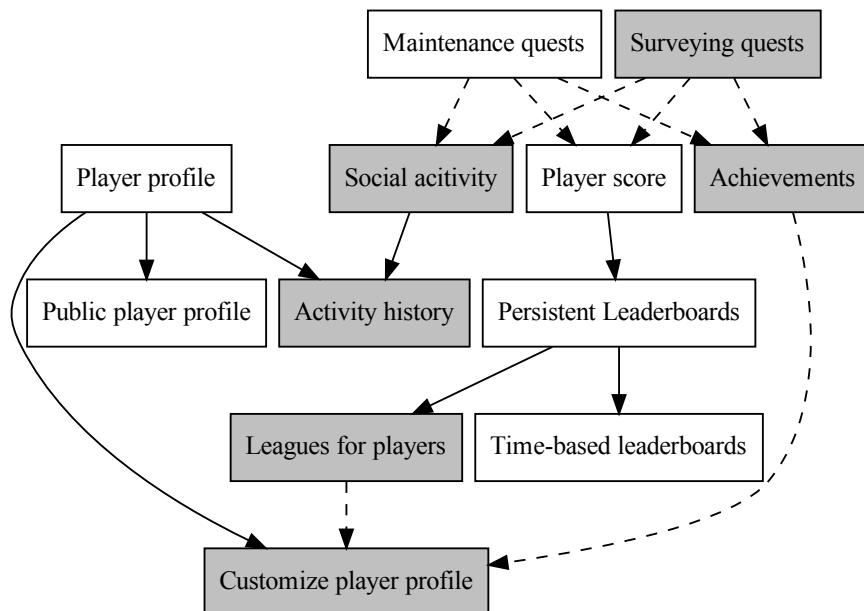


Figure 4.1: The feature dependency-graph showing features that will be implemented in white and features not targeted for implementation in grey.

4.1 Architecture

To make the system fulfil the design outlined in Chapter 3, we have to plan which subsystems to create, and which parts of the system handles which functions. Since we know that map data will need to be fetched from the centralised OpenStreetMap server, we know that the system will consist of at least a client on the user's phone, and the OpenStreetMap server. Part of the functionality and requirements are that we need to have communication between devices, to support functionality like leaderboards and activities. A centralised approach is simpler and better synchronisation but a peer-to-peer approach would be more in line with Requirement 8 about making MapTogether work offline without a permanent internet connection. Since we want the user's phone to do as much of the work as possible, we start out imagining a situation where the phone only talks to each other and the OpenStreetMap servers as illustrated in Fig. 4.2.

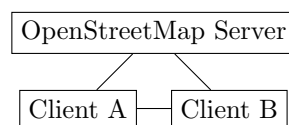


Figure 4.2: An example of the peer to peer architecture where data is fetched from OpenStreetMap servers.

If we chose this architecture, we will need to have some way of synchronising and connecting all the clients to each other.

A solution to this is to introduce another subsystem responsible for the synchronisation of data between the clients. This could be a server with an available WebRTC or REST HTTP API, which takes the information from the clients and builds a database that can be queried much more efficiently than each client building that database locally. This communication design is illustrated in Fig. 4.3.

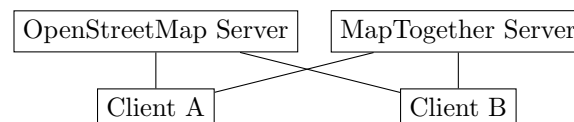


Figure 4.3: An example of a client-server architecture where map data is fetched from OpenStreetMap servers and social data is fetched from MapTogether servers.

This method provides much more simple communication. However, it introduces a single point of failure, and with no revenue, the bigger the server needed for MapTogether, the more losses the project will incur. Therefore we can try to use a hybrid approach, using the online server for as few subsystems as possible, and thereby relying more on the user's phone for the majority of the computational power.

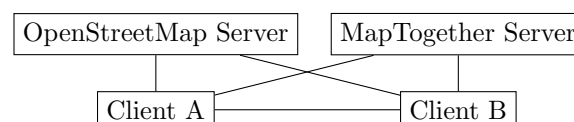


Figure 4.4: An example of a hybrid architecture where map data is fetched from OpenStreetMap servers and social data is fetched from MapTogether servers, with other tasks being done with a direct connection between clients.

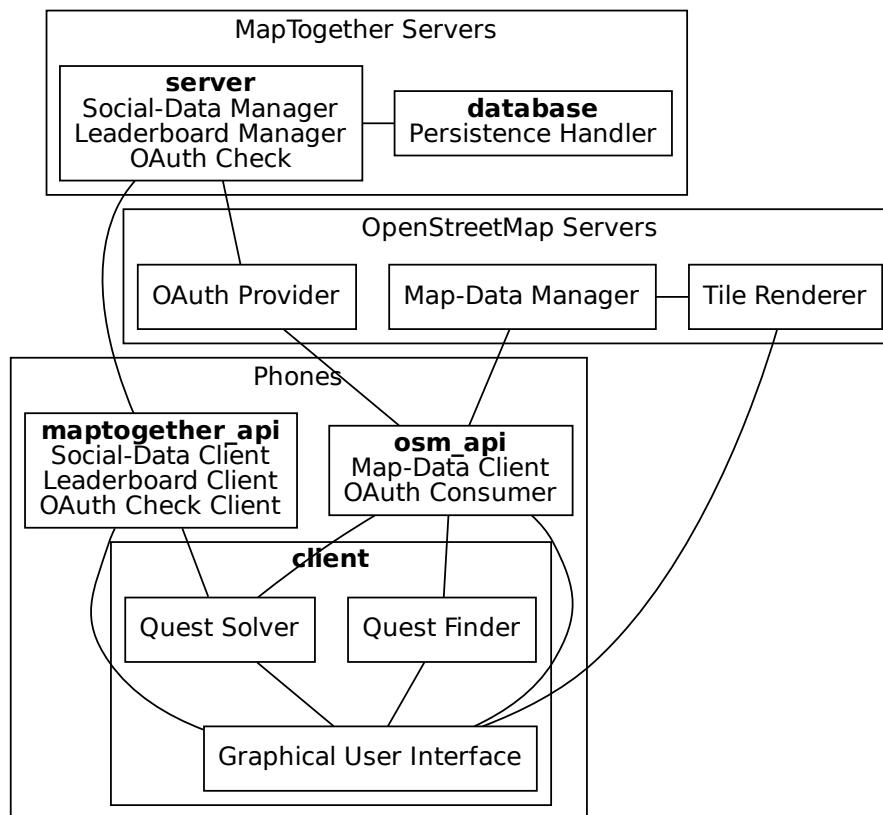


Figure 4.5: The component and subsystem diagram showing all hardware in the outer boxes, components as the inner boxes with names in bold, and subsystems listed inside the components.

The hybrid architecture in Fig. 4.4 would be the best to achieve the goals, but to keep development time down, we will not consider peer-to-peer transfer of map and social data, only the more limited activity data.

4.2 Subsystems and Components

Before we can specify exactly how the features are implemented, we will first define clearly which subsystems make up the whole of MapTogether. The subsystems are logical units that implement a feature, however, a single system component might be responsible for multiple subsystems and a single subsystem might need to reside in multiple components. Components are the real specific pieces of software that comprise the MapTogether code base.

4.2.1 Graphical User Interface

This is the subsystem whose functionality was outlined in Section 3.6. The design is done, but we need to make it available for the user. We can handle this in one of two ways: we

create a UI that the client renders like most mobile apps, or it can be rendered remotely as some websites do. One of the requirements for MapTogether is to have as much functionality available offline as possible, therefore it seems better to have the client be responsible for the GUI. We will place the Graphical User Interface subsystem in the `client` component on the phone.

4.2.2 Map-Data

The Map-Data Manager is a simple subsystem since it is already handled by the OpenStreetMap servers available with a public HTTP REST API. No further effort will be put into considering alternatives to this service, however, for availability reasons, it might be valuable to do caching of this data on the client. Since we do not write any of the code or decide the architecture of OpenStreetMap subsystems, we will not create a component on the OpenStreetMap servers, but simply place the Map-Data Manager subsystem there. On the phone we will need a way to talk to the Map-Data Manager, so we will define a Map-Data Client subsystem and place it in the `osm_api` component.

4.2.3 Quest-Finder

The Quest-Finder is responsible for figuring out what parts of the map data is incomplete or possibly out of date. This data needs to be turned into quests that can be displayed to the user, solved and the fixed data uploaded to OpenStreetMap. This could be handled by either the client or the MapTogether server. If it is done on the server it makes it possible to easily refer to the same quest with a single id for easy reference and to avoid any attempts at the same quest being solved by multiple clients. Doing this processing of the map data on the server would increase the amount of bandwidth and processing the MapTogether server would need to do. It also means the application becomes unusable if the MapTogether server goes down, which means less resistance to failure.

Because of this, we will place the Quest-Finder on the phone inside the `client` component.

4.2.4 Tile-Renderer

This subsystem could be handled by all three parts of the architecture. The OpenStreetMap websites slippy map is rendered by some freely available rendering-servers, which are available for public use, and that service or other similar paid tile-rendering services could be used. This does require more bandwidth, as the map data has to be fetched as rendered images or vector data and the user interface quality will deteriorate with the internet connection.

We can also make the client render the tiles on the user's device. This can result in less bandwidth and quicker response times on the user's device if the device is fast enough. It does require more processing on the user's device and might have an impact on the battery life. Most of the premade rendering software is made for either iOS or Android, requiring work to set up on each platform in order to fulfil the cross-platform requirement.

We can also choose to make the MapTogether servers host a tile-rendering service. This has many of the same benefits and drawbacks as the third-party rendering services, only offering more control over the look of tiles compared to the free OpenStreetMap service.

To support a cross-platform system more easily and reduce initial project complexity, we choose to use the OpenStreetMap tile rendering service, as we also need to fetch map data from the OpenStreetMap servers in order to find the quests. This means we place the Tile-

Renderer component on the OpenStreetMap servers, and have our Graphical User Interface talk directly to the Tile-Renderer.

4.2.5 Social-Information Handler

We need a subsystem that keeps track of which users follow who and how many contributions have been made by a user. We can handle this in three ways:

It can be stored on the client only, which allows for offline usage and less load on the servers, however, an approach to back up and move around the user's stored data will be necessary to support the user switching phones. It also means that data is lost if the phone is lost.

OpenStreetMap already has some social information stored. There is a friend/follow list that works in the same way intended for MapTogether, which could be used to store data about user relationships. If we do this, it will couple our friend-list to the OpenStreetMap one, which means the user cannot use MapTogether without changing their OpenStreetMap profile. It provides benefits in synchronisation between MapTogether and other services using the OpenStreetMap profile, and it lets the MapTogether servers do less work.

The last way we can manage the social information is by making a service storing the information separately from the client and the OpenStreetMap server. This provides decoupling between OpenStreetMap and MapTogether, allowing the user to opt-out of the social features, while also getting the benefits of information stored centrally. It does mean more work has to be done on the MapTogether servers, but with the benefits it provides, it is the path we will take. Therefore we place the Social-Data Manager in the **server** component and the Persistence Handler in the **database** component on the MapTogether servers. Furthermore, to interface with the Social-Data Manager, we create the Social-Data Client and place it in the **maptogether_api** component on the phone.

4.2.6 Login-Manager

We need a subsystem to manage both the credentials for doing changes to the OpenStreetMap data, but also for changing the social information on the MapTogether servers. OpenStreetMap provides the ability to use the OAuth protocol for authenticating external applications via the OpenStreetMap login. Because of this ability, the decoupling between OpenStreetMap and MapTogether will not have to require a separate login for each service. It is a possibility to have a separate login for each service, but it seems to have no drawback to unify them while making a breach of the MapTogether social manager less important.

To support the Login-Manager, we create three subsubsystems, the *OAuth Provider* already handled by the OpenStreetMap servers, the *OAuth Consumer* in the **osm_api** component on the client, responsible for doing the OAuth workflow, and a OAuth Check subsystem in the **server** component on the MapTogether servers for verifying that the OpenStreetMap credentials the client supplies are valid. Lastly, we add the OAuth Check Client in the **maptogether_api** component on the phone. This four-way split can be seen in Fig. 4.5.

4.2.7 Leaderboard-Calculator

This is a subsystem that also could be implemented in one of three ways. We can calculate the scores by looking at all changesets on the OpenStreetMap servers that have been created since the start of MapTogether. These changesets can contain metadata tags that can be used to figure out which changes are MapTogether related.

```

1 <osm>
2   <changeset uid="11321" user="maptogether-test">
3     <tag k="comment" v="Add missing backrest info about ↗
↗bench"/>
4     <tag k="created_by" v="MapTogether v0.2.2"/>
5     <tag k="maptogether_score_awarded" v=5/>
6     <tag k="maptogether_task_type" v=" ↗
↗backrest_bench_quest"/>
7   </changeset>
8 </osm>

```

Figure 4.6: Changeset metadata we could use for storing MapTogether specific information.

The problem with the approach shown in Fig. 4.6 is the possibly very large amount of irrelevant data queried from non-MapTogether changesets resulting in wasting bandwidth and processing time on the OpenStreetMap servers and the client. One solution to this could be to have processed leaderboards stored on a MapTogether server, only being updated with the newest data once in a while, reducing the bandwidth while also not requiring much storage, as only the compiled information needs to be stored.

The OpenStreetMap project might be weary of adding a bunch of metadata tags that do not directly improve the OpenStreetMap map data but only serve as data storage for MapTogether. It is however what is done for the StreetComplete project, where the type of quest is recorded in the changeset tags, and a service called `sc-statistics-service`¹ parses the changesets and keeps track of user scores and achievements. MapTogether makes even more use of social features and has the option to opt-out of social features entirely, which means a design where the client informs the server about new changes manually, makes for more client-controlled handling of the social service.

To facilitate this, we create a Leaderboard Manager in the `server` component on the MapTogether servers and a Leaderboard Client in the `maptogether_api` component on the phones.

4.2.8 Quest-Solver

The Quest-Solver is the subsystem that takes care of when the user answers a quest. Since we made the client inform the MapTogether server of the solved quests, it might seem logical that the MapTogether server is responsible for telling the OpenStreetMap servers of the changed map-data. However, as the user can opt-out of the social features, it needs to be able to tell the OpenStreetMap servers of changes without talking to the MapTogether servers. To avoid the need for both the client and MapTogether servers to communicate with OpenStreetMap about map-data, we can give the complete responsibility for communicating about map-data to the client.

As such we create a Quest Solver subsystem in the `client` component on the phones, which talks with the Map-Data Client subsystem.

4.3 Connecting Devices

We need to connect user's devices in order to implement several features. For example, when creating an activity, users need to invite other nearby players. The connection can be

¹<https://github.com/streetcomplete/sc-statistics-service>

made locally between nearby devices by utilising the Bluetooth antennas built into almost all modern mobile phones or via an internet connection, either peer to peer or client-server. As mentioned in Section 4.1, we want to minimise the workload of the server, and therefore, we explore the options for making the devices connect through Bluetooth.

Most modern phones have a dual-mode module that enables them to communicate with both devices using *Bluetooth Classic* and devices using *Bluetooth Low Energy (BLE)* [21]. Since we are implementing the MapTogether application using the Dart framework Flutter, we look at some ready to use Flutter libraries for working with Bluetooth. Examples of these are *flutter_serial_bluetooth*² and *flutter_blue*³. These are different Flutter libraries that are wrappers around the native iOS and Android Bluetooth libraries. Of course, more Flutter Bluetooth libraries exist, but because Flutter is a new framework, many libraries support only iOS or Android and not both.

One fundamental difference between the mentioned libraries is that *flutter_serial_bluetooth* uses *Bluetooth Classic*, and *flutter_blue* uses *Bluetooth Low Energy (BLE)*. However, Bluetooth Classic devices communicate, only when they are paired beforehand, requiring the users of MapTogether to close the app and pair their devices. This becomes tedious. As searching for a username would be faster. This means that Bluetooth Classic and thereby *flutter_serial_bluetooth* are not applicable for use in MapTogether because it becomes more tedious to add players, than by searching for usernames. We will instead restrict us to explore BLE and *flutter_blue* in further details.

The BLE protocol differentiates between two types of devices, namely peripheral devices and central devices. The peripheral devices are typically constrained devices that need to conserve energy, and the central devices typically have more processing power and memory. An example of this could be a smartwatch as a peripheral device and a mobile phone as a central device. However, most Android and iOS phones support switching between peripheral and central devices, and this is possible programming-wise through each platform's respective native Bluetooth libraries.

For the MapTogether, we need the phone, which creates an activity, to be a peripheral device and the phones searching for an activity to be central devices. However, the *flutter_blue* library only supports central mode, meaning that we can not rely solely on *flutter_blue*, since we need to switch between the modes to connect the devices. It turns out that only a single BLE flutter library that supports peripheral mode exists⁴, and it has only limited support for iOS, maybe due to Flutter being a relatively new framework. Therefore, we have to use something else or develop a BLE flutter library that supports both peripheral and central device modes.

After investigating different possibilities, we discovered *Google Nearby Messages API*⁵, which we found suitable as an alternative to Bluetooth even though it requires internet to function. *Google Nearby Messages* is an *API* to publish and subscribe to small messages between internet-connected devices. It is well-suited because it does not require any pairing of the devices, it is available for both iOS and Android, and the devices are not required to be on the same network. The *Google Nearby Messages API* discovers nearby devices by utilising *Bluetooth Low Energy*, *WI-FI*, and near-ultrasonic audio to communicate a unique-in-time pairing code and then communicate this to the Google servers, which informs the client if there are any published messages from the device which transmitted the pairing code. Since the *Google Nearby Messages API* uses sound, which cannot go through walls,

²*flutter_bluetooth_serial*: https://pub.dev/packages/flutter_bluetooth_serial

³*flutter_blue*: https://pub.dev/packages/flutter_blue

⁴*flutter_ble_peripheral*: https://pub.dev/packages/flutter_ble_peripheral

⁵<https://developers.google.com/nearby/messages/overview>

it is more like when humans are looking for nearby players to discover other players. Thus, only other players whom the player can see in real life are found by the player's phone.

Google Nearby Messages sounds like a promising technology to use. As we did for Bluetooth, we start by exploring Flutter libraries compatible with *Google Nearby Messages*. However, it becomes apparent that *Google Nearby Messages* is not very popular. We are only able to find a few, and according to their documentation, most of these only work with Android. Therefore we explore the possibility of developing a Flutter library for *Google Nearby Messages API* ourselves.

Flutter libraries that call native code, like functionality from *Google Nearby Messages API*, require method channels, making it possible for cross-platform Flutter code to invoke platform-specific functions written in other programming languages. We start by creating the Android part of the library. Working with *Google Nearby Messages*, we discover problems getting the API to transmit any messages. We searched for other projects using *Google Nearby Message API* in order to get recent successful examples. We discover a project developed by the Google Team. However, unfortunately, this project does not work either. Since we cannot find recent usages or examples that work, we give up on using *Google Nearby Message API*. It seems that the API is outdated.

We decide to develop the feature of finding nearby players by using internet communication to a central server instead. The users publish their location to a central server and then receive a list of nearby players from the server.

4.4 Log-In Handler

In the following section we describe how the authentication of both the OpenStreetMap servers and the MapTogether server. OpenStreetMap supports both OAuth 1.0 and OAuth 1.0a for authentication, although 1.0 is only supported for legacy applications. OAuth allows the client to access resources on the server, and authorise the third-party application. This allows for authentication of the user in our application⁶. The OAuth terminology varies dependent on who uses the authentication method. In OpenStreetMap the terminology differs from RFC5849, and uses the terminology from the original community specification, and we also use the original community specification when talking about OAuth.

In the original specification a consumer is simply just a HTTP client, and a service provider is an HTTP server which in our case is the OpenStreetMap server. The consumer token and consumer secret is assigned by OpenStreetMap to uniquely identify the application. The temporary token is used for the initial authentication and is only available for a limited time frame. This does not give access to the resources on the server and cannot be used for anything but authentication. The access token is a long lived token, and the temporary token is destroyed when the access token is created. The access token have to be included in every request for resources that are authenticated.

When the client request temporary access, it sends the consumer token to identify which application needs to get access to the OpenStreetMap server. The server checks if the consumer token is valid, and sends the temporary access token back to the client. The client then shows the login page in an embedded webbrowser, and the user have to login and grant OpenStreetMap access to the MapTogether application. The server then checks the request to see if the consumer key, temp token and verifier are all valid, and makes a valid access token if they are. This token is then returned to the client. The received access token

⁶RFC for OAuth 1.0: <https://datatracker.ietf.org/doc/html/rfc5849>

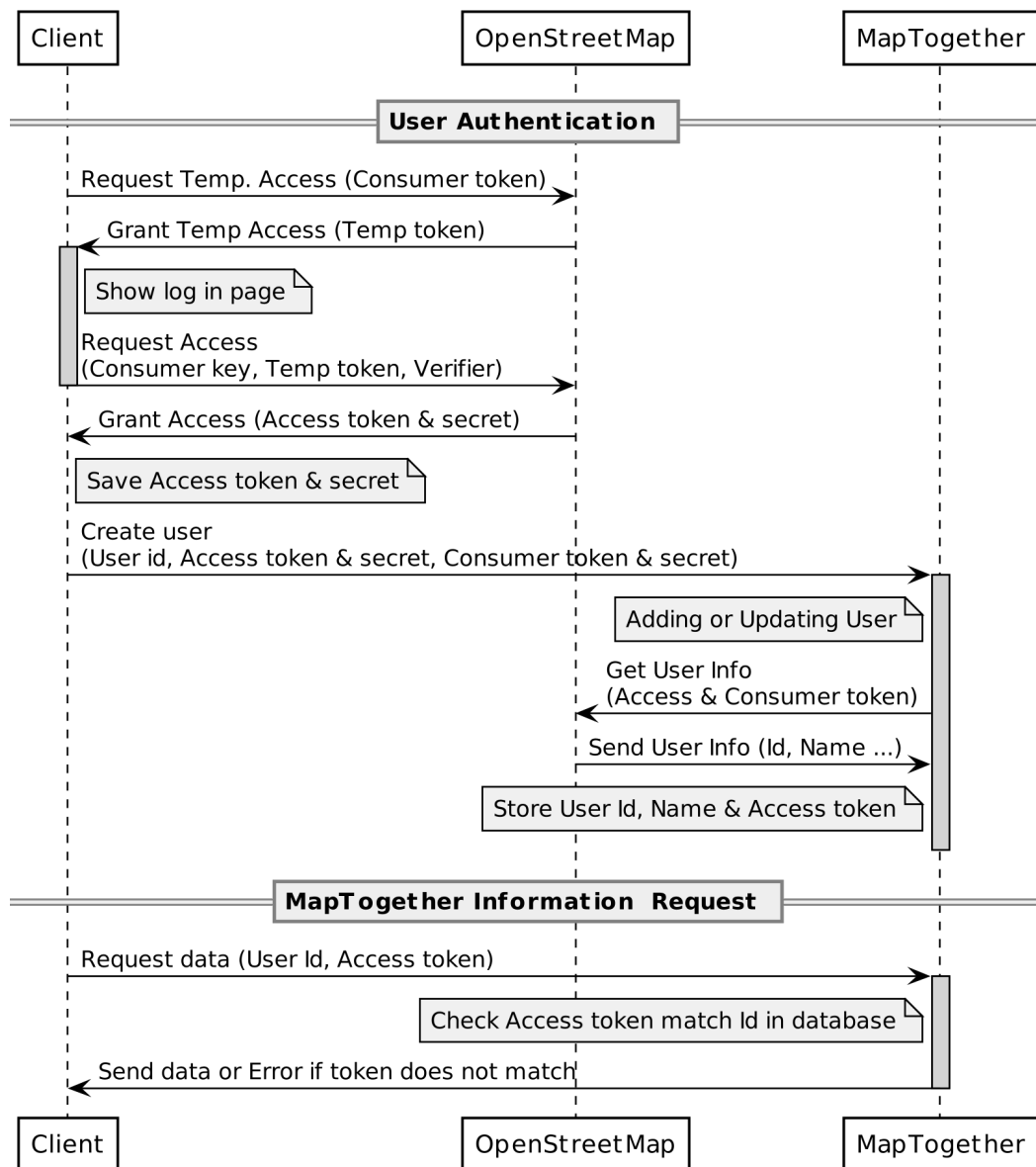


Figure 4.7: Sequence diagram of a client logging in to OpenStreetMap and being verified, and requesting data from the MapTogether server.

and secret is saved in the devices persistent memory, such that it can be used whenever a OpenStreetMap change is made.

When the client has received the access token and secret from OpenStreetMap, then it sends a request to the MapTogether server to create a new user (or replace an existing one). The client provides the server with their user Id, access token and secret and their consumer token and secret. The server then use the access and consumer tokens and secrets to sign a request for the user data from OpenStreetMap. If the user information is received, then the server will store (or overwrite) the user Id, name and access token in the database.

When the client has retrieved an access key from OpenStreetMap and the MapTogether database has stored the user information with the access key, then the client can request personal information from the MapTogether server. As shown in Fig. 4.7, the client sends a request to an endpoint in the server. The server need to know the Id of the user that makes the request, and an access token to verify that the client is requesting data for the user that they are logged in with.

```

1      %aid = {{db}}.query_one? "SELECT userid FROM users
2      ↳WHERE access = $1", %key, as: Int64
3      http_raise 401, "User #{%id} does not have the given
4      ↳access token" if %aid.nil?
5      http_raise 401, "Authenticated user does not have
6      ↳permission for this (#{%id} != #{%aid})" if %id != %aid
7      end
8
9      put "/user/:id" do |env|
10         id = env.params.url["id"].to_i64
11
12         vals = env.request.headers["Authorization"].split("
13         ↳")
14         http_raise 400, "Both access and client keypairs are
15         ↳required" if vals.size < 5
16         atype, key, secret, ckey, csecret = vals
17         http_raise 400, "Authentication type needs to be '
18         ↳Basic'" if atype != "Basic"

```

Listing 4.1: Macro in Crystal to verify that the id matches the access token in the header of a request.

As shown in Listing 4.1, the server first checks for errors in the header of the request. The header should be on the form "Basic <Access Key>". On line 12 the server tries to retrieve a user Id (%aid) from the database using the access token. If %aid is null, then it means that no user in the database has the given access token. If %aid is not equal to the provided user Id, then it means that the client is requesting information that they do not have permission to.

4.5 Database and Server

In this section, we design and implement the server and database, in accordance with the classes, events and functions previously outlined in Section 3.5 as well as the user interface designed in Section 3.6.

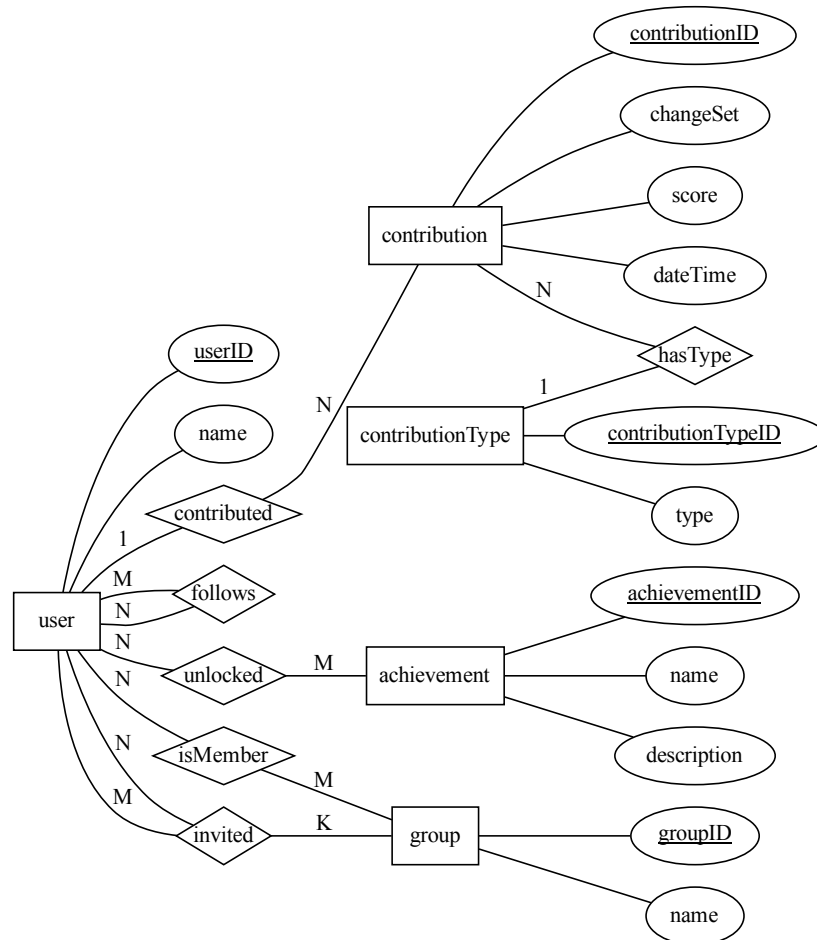


Figure 4.8: Entity relationship diagram of entities in the MapTogether

4.5.1 Database and Server Design

To design the database, we must decide what the responsibility of the database is. The database should contain the social data necessary for the application to function. We want the database to contain information about the users' followers, groups and achievements. The database should also contain contributions' points and type, to keep track of what kind of different contributions each user has made. The type of contributions could be used to unlock specific achievements.

In order to visualise and describe the relations within the database, we draw the entity relationship (ER) diagram in Fig. 4.8. This diagram describes the entities in the database, their attributes as well as the relations between them.

We have the following entities: user, contribution, contributionType, achievement and group. A user has an ID and a name. We want the user to be linked to a profile in OpenStreetMap, so the ID should be the same as their OpenStreetMap profile. The user name should also be a copy of their OpenStreetMap user name. A user contributes a

number of contributions. A contribution has an ID, a changeset that corresponds with an OpenStreetMap changeset ID, a score and a timestamp. The score is the number of points rewarded to the user. A contribution also has a type. A contributionType has an identification and a name of the type. The idea behind types is to be able to award achievements depending on user activity.

A user unlocks a number of achievements. An achievement has an ID, name and description. A user is also member of any number of groups. Groups just have an ID and a name. A user can also receive an invitation from another user to a certain group.

Based on the entity relationship diagram, we form the following collection of relation schemas.

```

users : {userID, name}
contributionType : {contributionTypeID, type}
contributions : {contributionID, userID → users(userID),
                  type → contributionTypes(contributionTypeID), changeSet, score, dateTime}
achievements : {achievementID, name, description}
groups : {groupID, name}

unlocked : {userID → users(userID), achievement → achievements(achievementID)}
follows : {follower → users(userID), followee → users(userID)}
hasMember : {groupID → groups(groupID), userID → users(userID)}
invitations : {invitationID, host → users(userID), invitee → users(userID),
                group → groups(groupID)}
```

The database runs on a server, accessible through endpoints via HTTP requests as a REST API. We choose this setup because the information is always requested by the client and response time on the endpoints is not critical for the app. These endpoints are used for accessing the database by performing SQL queries.

Endpoints

We developed the wrapper around the API in Dart, using the language constructs for asynchronous programming, such that we can have multiple different operation running while waiting for some operations. This also makes it easier in development since the Client does not need to handle JSON, but simply other Dart objects.

We specify some endpoints in order to access the data from the database. To access data about a user, we set the endpoint: `/user/<id>`. This endpoint is used for accessing a user by their id, with a GET request. The id is a parameter given in the URL. The response should include information about achievements, followers and scores in different time frames. The same endpoint allows for adding a user with a PUT request. The data sent with the PUT request have the same information structure as one would receive from the GET request.

We also need an endpoint to add or remove follow relations between users. We use the endpoint: `/user/<follower>/following/<followee>`. A HTTP PUT request adds the relation that the user with id `follower` now follows the user with id `followee`. A corresponding DELETE request will remove said relation if it exists.

We also need to be able to add new contributions. To do this, the client can send a POST request with the needed data to the endpoint: `/contribution`.

Since we have multiple types of leaderboards, we create endpoints for each of them by using URL parameters. Leaderboards are categorised into three time frames: all time, monthly and weekly. For each category we have a global and a personal leaderboard. The global leaderboard includes all users globally. The personal leaderboard is user specific, and includes a user and all the users they follow. The leaderboards can be accessed by a HTTP GET request to the endpoint: `/leaderboard/<time_frame>/<global|personal>/<id>`.

4.5.2 Database and Server Implementation

We implement the designed server and database from Section 4.5. First, we set up the database management system. For this we use PostgreSQL⁷. We initialise the DBMS with UTF-8 encoding in order to allow special local characters. In Section 4.5 we specified the tables and their attributes, we create them as shown in Listing 4.2.

```

1 CREATE TABLE IF NOT EXISTS users (
2     userID    bigint PRIMARY KEY,
3     name      varchar(255),
4     access    varchar
5 );
6
7 CREATE TABLE IF NOT EXISTS contributionTypes (
8     contributionTypeID SERIAL PRIMARY KEY,
9     type        varchar
10 );
11
12 CREATE TABLE IF NOT EXISTS contributions (
13     contributionID BIGSERIAL PRIMARY KEY,
14     userID          bigint REFERENCES users(userID),
15     type            integer REFERENCES contributionTypes(
16         ↪contributionTypeID),
17     changeSet       bigint,
18     score            integer,
19     dateTime         timestampz
20 );
21
22 CREATE TABLE IF NOT EXISTS achievements (
23     achievementID BIGSERIAL PRIMARY KEY,
24     name          varchar,
25     description    varchar
26 );
27
28 CREATE TABLE IF NOT EXISTS unlocked (
29     userID          bigint REFERENCES users(userID),
30     achievement      bigint REFERENCES achievements(achievementID
31         ↪),
32     PRIMARY KEY (userID, achievement)
33 );
34
35 CREATE TABLE IF NOT EXISTS follows (

```

⁷<https://www.postgresql.org>

```

35     follower      bigint REFERENCES users(userID),
36     followee      bigint REFERENCES users(userID),
37     PRIMARY KEY   (follower, followee)
38 );
39
40 CREATE TABLE IF NOT EXISTS groups (
41     groupID BIGSERIAL PRIMARY KEY
42 );
43
44 CREATE TABLE IF NOT EXISTS hasMember (
45     groupID bigint REFERENCES groups(groupID),
46     userID  bigint REFERENCES users(userID),
47     PRIMARY KEY (groupID, userID)
48 );

```

Listing 4.2: SQL code for creating the tables specified in Section 4.5

We use **bigint** (64 bit integers) for identification attributes, and **integer** (32 bit integer) for smaller numbers that are going to be set by the application. Attributes of type **varchar** are of unspecified length, except for the users' names, which are defined by the users themselves and has a maximum length of 20 characters.

Leaderboards are not saved as relations in the database, so in order to get a leaderboard we need to combine *users* and *contributions* and sum the scores. We know that clients could request data on the leaderboards quite often. Because of this we use *materialised views* to save the global leaderboards. One of the materialised views can be seen in Listing 4.3. The materialised views provide better performance, because the server does not have to compute the leaderboard at each new request. The server updates these materialised views with a interval of one minute, such that the leaderboards updates.

```

1 CREATE MATERIALIZED VIEW leaderboardWeekly AS
2     SELECT u.userID, u.name, COALESCE(s.score, 0) AS score
3     FROM (
4         SELECT userID, SUM (score) AS score
5         FROM contributions
6         WHERE dateTime BETWEEN date_trunc('week',
7         ↪ CURRENT_DATE) AND CURRENT_DATE
8         GROUP BY userID
9         ) AS s
10    RIGHT OUTER JOIN
11    users AS u
12    ON u.userID = s.userID
    ORDER BY score DESC;

```

Listing 4.3: A materialised view of the global weekly leaderboard.

To implement the server endpoints we use Kemal⁸, a framework for developing REST API's and more. Kemal is written in Crystal⁹, a primarily functional and object-oriented language. We use JSON for HTTP POST parameters and all responses from the server.

We implement the server as a REST API, which means that the server is stateless, and requests are always processed individually with the same procedure. This also makes the

⁸<https://kemalcr.com>

⁹<https://crystal-lang.org>

server highly scalable. The clients send HTTP requests to the server, which the server then processes and queries the database accordingly.

We setup the endpoint `/leaderboard/<time_frame>/personal/<id>` to allow a client to request personal leaderboard data for all time, monthly or weekly time frames. The implementation of handling a HTTP GET request on this endpoint is shown in Listing 4.4. This endpoint has two URL parameters, and responds with a JSON object that is an ordered list of users with a name and id, and a score.

```

1  # Retrieve all users' id, name and score
2  get "/leaderboard/:time/global" do |env|
3      time = LeaderboardType.from_s env.params.url["time"]
4
5      string = JSON.build do |json|
6          POOL.using_connection do |db|
7              db.query Queries.leaderboard(RankType:: ↗
↪Global, time) do |rows|
8                  Leaderboard.new(rows).to_json json
9              end
10             end
11         end
12
13         env.response.content_type = "application/json"
14         string
15     end

```

Listing 4.4: Crystal implementation of HTTP GET request on the personal leaderboard endpoint.

The query we use to get the leaderboard data (line 7 of Listing 4.4) is returned from a function shown in Listing 4.5. The function takes two parameters, `RankType` (global or personal) and `LeaderboardType` (all time, monthly or weekly). The function queries the correct materialised view depending on the leaderboard type. If the the rank type is global, the query is for all users, if the rank type is personal, then the query is for followed users. The SQL query selects all attributes from a materialised view. We declare most queries as functions with parameters in a Crystal module called `Queries`.

```

1  def leaderboard(r : RankType, l : LeaderboardType)
2      if r == RankType::Global
3          "
4          SELECT *
5          FROM #{l.to_leaderboard}
6          WHERE score != 0
7          "
8      else
9          "
10         SELECT *
11         FROM #{l.to_leaderboard}
12         WHERE EXISTS (
13             SELECT 1
14             FROM follows
15             WHERE userID = $1 OR (follower = $1 AND ↗
↪followee = userID)
16         )
17         "

```

```

18         end
19     end

```

Listing 4.5: SQL query for getting a leaderboard. The first parameter is either global or personal, the second parameter is the time frame converted to the name of a materialised view (all time - “leaderboardAllTime”, monthly - “leaderboardMonthly” or weekly - “leaderboardWeekly”)

On line 8 of Listing 4.4 a new instance of the **Leaderboard** class is created and the method **to_json** is called on the new instance. The constructor of **Leaderboard** takes the result of a query with three columns: user id, name and score. The **to_json** method takes a JSON Builder with which it builds a JSON object like the example shown in Listing 4.6.

```

1  [
2      {
3          user: {
4              id: <user id>,
5              name: <user name>
6          },
7          score: <1. place score>
8      },
9      ...
10 ]

```

Listing 4.6: Example of the response a client will receive from a HTTP GET request to a leaderboard endpoint. A sorted list of pairs of a score and a user with an id and name.

The other endpoints specified in Section 4.5.1 are implemented the same manner. We use up to two URL parameters for some endpoints and a JSON object as a body parameter for uploading advanced elements.

4.6 OpenStreetMap API Wrapper

In the following section, the design of the communication between the OpenStreetMap server and the client is outlined. This design is made according to functions in Section 3.5, and the target audience, from Section 3.1 whom wishes to contribute to OpenStreetMap¹⁰. We build a wrapper around the communication such that the calls to the API are handled in a separate part of the project. This allows the Client and User Interface to abstract away the direct information regarding the endpoint, and thus we only need to handle Dart classes on the Client.

4.6.1 Endpoints

To access data from the OpenStreetMap servers, we develop a wrapper around the OpenStreetMap API. The endpoints we need to support for our application are map, user and changeset. However, the wrapper also supports other OpenStreetMap endpoints such that other developers can use it in the future. To gather the data about the currently logged in user, the endpoint: `/api/0.6/user/details` is used. The response returns the information shown from Listing 4.7.

¹⁰https://wiki.openstreetmap.org/wiki/API_v0.6

```

1  "user": {
2    "id": 11321,
3    "display_name": "maptogether-test",
4    "account_created": "2021-04-26T11:59:10Z",
5    "description": "",
6    "contributor_terms": {
7      "agreed": true,
8      "pd": false
9    },
10   "roles": [],
11   "changesets": {
12     "count": 0
13   },
14   "traces": {
15     "count": 0
16   },
17   "blocks": {
18     "received": {
19       ...
20     }
21   },
22   "languages": [
23     ...
24   ],
25   "messages": {
26     "received": {
27       ...
28     },
29     "sent": {
30       "count": 0
31     }
32   }
33 }

```

Listing 4.7: The response from a GET call to the user endpoint

An abbreviation of the user information is also useful in our server, to show the different information regarding the logged-in user as seen on the Fig. 3.13a. An example of this is users shown on the leaderboards and the users seen in the follow-list. For these users, we do not require every piece of information about the user, only a few data-points such as their score, name, id and profile picture. For this reason we make a call to retrieve an abbreviated user only containing these items, to minimise the amount of data sent.

To gather data regarding nodes, ways, and relations, the endpoint `/api/0.6/map?bbox=left,bottom,right,top` is used. The endpoint returns every node, way and relation within the square constructed by the four corners, given as a parameter to the endpoint, where an abbreviation of this can be seen on Listing 4.8. This allows for the construction of multiple different quests, with a single call to the endpoint.

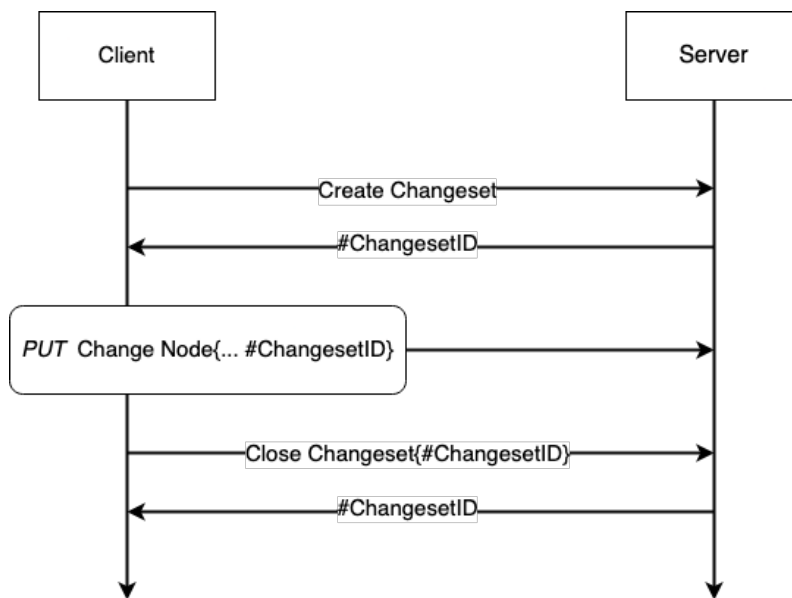
```

1 {
2   "elements": [
3     {
4       "type": "node"
5     },
6     {
7       "type": "way",
8     },
9     {
10      "type": "relation",
11    },
12  ],
13 }

```

Listing 4.8: Abbreviated representation of response to the map endpoint

Writing data to the OpenStreetMap servers requires user authentication, and how the authentication work is mentioned in Section 4.4. Aside from the argument mentioned in Section 4.4, another reason for authentication is also to minimise the number of bad actors interacting with the server. To insert a new object in the OpenStreetMap server, when the user already is authenticated, the object has to be encapsulated in a changeset. Each changeset has a comment describing the encapsulated content. The changeset flow can be seen on Fig. 4.9.

Figure 4.9: An example of a changeset used for changing already existing node¹¹

We close the changeset immediately after changing an element, this means that the quest related to the element will be answered and removed from the quest shown in the client.

However other users will not be able to see the quest on their device when they reload the quests.

4.7 Client

We have multiple options for the client implementation, which is the mobile application used by the users. One option is to develop native applications, but because MapTogether must support both iOS and Android we have to implement and maintain two different native applications, one for Android and one for iOS. Many companies select this approach when developing applications because it is possible to achieve the best possible performance for the specific platform. Also, the challenge of maintaining multiple apps is minimal when the applications mainly consist of UI that fetches data from a server that runs all the business logic. In this case, the business logic is only implemented once since it is not dependent on the native platforms. However, as mentioned in the architecture section, we want the subsystem responsible for finding the quests to run locally on the client. Therefore, we would need a quest system for the Android platform and another one for the iOS platform, if we chose the native approach. Instead, we explore possibilities for cross-platform development so we only have to develop and maintain a single codebase. There are multiple cross-platform options like Xamarin, React Native and Flutter. Flutter provides some convenient features like hot-reload and hot-restart, and we, therefore, implement the client side of MapTogether, using the Dart framework Flutter.

UI elements in Flutter are built using *Widgets*. These are ways for us to declare and construct graphical elements for the UI, in a condensed yet mutable manner. Widgets are both responsible for displaying information but also setting up the design and layout of specific pages. For example, columns, rows and padding in Flutter, are all represented through widgets. To pass information between the different screens and widgets in Flutter, as well as fetch data from our server and *OpenStreetMap*. We use a mix of futures and providers.

4.7.1 Futures

When the client needs to read or write data from the server, said data is not readily available, without first making a call to the server. Likewise, it is not always certain that MapTogether will be able to retrieve the data, and therefore needs to be able to handle potential errors which might occur while fetching the data. For this reason, we use the future property of Flutter. A future is effectively an asynchronous method for telling a widget that the corresponding piece of data associated with the future, will be provided at a later point in time. We then send a fetch request for the data to our server and the future awaits a response. Through a widget called a *FutureBuilder*¹², we can then build specific widgets on the screen, with varying contents depending on the response from the server. For example, while the data is being fetched a loading screen appears. Depending on whether the data is then correctly fetched, a widget containing either the correct information or an error message is displayed.

4.7.2 Provider

In order to pass data between screens and widgets in MapTogether, we make use of providers¹³. Providers are a means of passing information between screens and widgets in a Flutter project. This ensures that all UI elements dependent of data with the associated provider are updated in both the back and front end of the program. Ordinarily, one way to do it

¹²<https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html>

¹³<https://pub.dev/packages/provider>

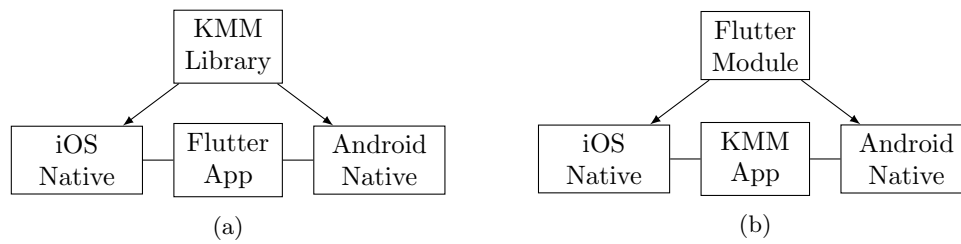


Figure 4.10: The two way to structure a combined Flutter/KMM project.

in Flutter is by creating so-called stateful¹⁴ widgets, whose state then have to be manually updated whenever a change in the data set occurs, for it to be reflected in the UI.

4.7.3 Quest System

From Section 3.2, we know that MapTogether must provide the functionality to maintain existing map data by adding and updating tags on nodes, ways and relations. This was further specified as the two subsystems Quest-Finding and Quest-Solving in Section 4.1. StreetComplete already has a system for finding and solving quests and since StreetComplete is an open-source project, their quest system can legally be used in MapTogether. These quests could even be solved during social waypoint activities, as we proposed in Section 3.3. The StreetComplete quest system is written in Java, which is not directly compatible with the Dart/Flutter system chosen in Section 4.7. To be able to use the already developed and maintained quest system, we will figure out a way to combine Java and Dart code.

The only available way is using Kotlin Multiplatform Mobile (KMM), which allows using Kotlin code for logic on both Android and iOS. Since Kotlin can interface with Java code this means the quest system could be interfaced with on both platforms. Combining Flutter and KMM is not a widely used approach and none of the available projects online that tries to use it, is not functional on both iOS and Android.

We can combine KMM and Flutter two ways. We can create a Flutter app and a KMM library (Fig. 4.10a), where the Flutter Dart code talks to the native code via Flutter Method-Channels and the native code talks to the KMM library as if it was a native library. This approach has the problem of the KMM Library setup that is importable in iOS not being importable in Android and vice versa.

The second way of combining KMM and Flutter is by creating a KMM App and a Flutter module as seen in Fig. 4.10b. The communication between native code and Kotlin is almost seamless like if it was all native code, and the communication with Flutter being done via the aforementioned MethodChannels. The big problem with this setup is that setting up an KMM-driven Android native app, seems to exclude proper import of the flutter module.

None of the two ways have any working examples available, and the authors spent considerable effort trying to get it set up, but the authors were unable set up a working project, that allows combination of KMM and Flutter.

Since we are unable to use KMM in MapTogether and therefore cannot use StreetCompletes quests, we develop a new quest system in Dart, as it should be part of the mobile app (Client) for the reasons mentioned in Section 4.1. We prioritise regular quests instead of the survey quests mentioned in Section 3.3.4, as most other features of the application depend

¹⁴<https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html>

on either StreetComplete style maintenance quests or survey quests being implemented, as seen in the dependency graph in Fig. 3.9. Maintenance quests are chosen over surveying since surveying quests will not cover the mentioned must-have requirement of maintaining existing data. The quest system needs to find quests in OpenStreetMap data as illustrated in Fig. 4.11.



Figure 4.11: The Quest system input and output

The quest system that we develop is a proof of concept system that only includes two types of quests: a **BackrestBench** quest, asking the user if a bench has a backrest, and a **BuildingType** quest, asking the user what type of building a building is, for example, a school or commercial building. We implement the quest system such that new quests added in the future can easily be integrated by inheriting and implementing the relevant functionality from an abstract class.

The quest system consists of **Quests** and **QuestFinders**. The class diagram for these is shown in Fig. 4.12. All quests have some common elements in the abstract class **Quest**. The common elements consist of an OpenStreetMap data element (a node, relation or way), a position, an icon, a changeset comment (describing the change to the OpenStreetMap data), and the question that the user should answer. All quest types should inherit the **Quest** class. The two quests that we implement are of the **SimpleTagQuest** type. What identifies quests of this type is that solving them means adding or updating a simple tag. For example, in the **BackrestBench Quest**, either “yes” or “no” should be added to the backrest tag for the given bench. Thus **possibilities** is a list of the possible values for the specific tag that is updated.

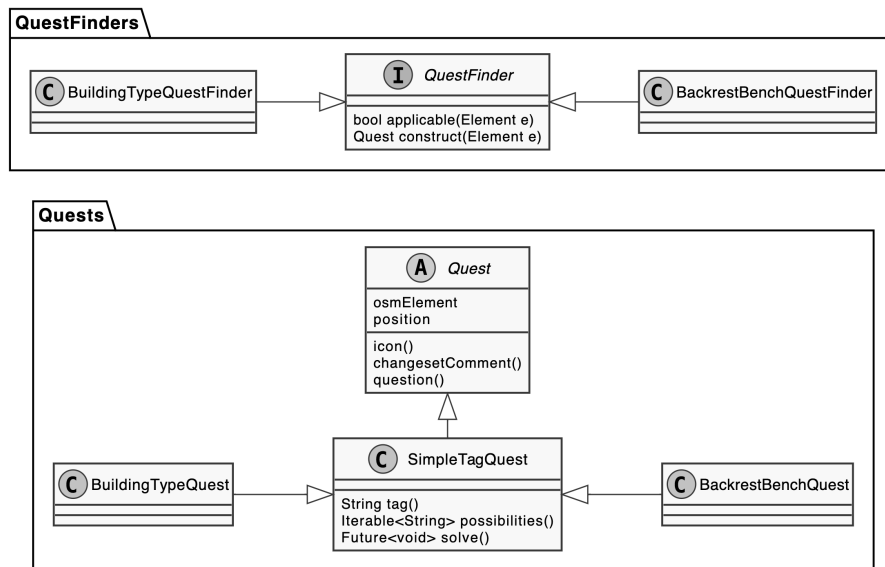



Figure 4.12: Quest system class structure

To display the **Quests** on the map and the pop-ups with the question and answers, we define a simple marker-widget showing the **Quest** icon on the map, and we implement a pop-up

for each type of `Quest`. For example, for the `SimpleTagQuest`, we create a pop-up showing the question with a scrollable list of buttons, one for each possible answer.

The interface `QuestFinder` provides functionality to find the different type of quests when given OpenStreetMap data. A `QuestFinder` has two methods, `applicable` and `construct` . The `applicable` method takes an OpenStreetMap data element and checks if the given `Quest` applies to that element. `construct` calls the constructor of the corresponding quest-type of the specific `QuestFinder` instance. With the `QuestFinder` and `Quest` defined, we can find all quests in an area by testing if each `Quest` applies to the area's elements.

Chapter 5

Evaluation

In this chapter we examine the performance of MapTogether. We also discuss the fulfilment of the requirements to conclude whether the implementation is sufficient and if the implementation fulfils the problem statement. Lastly, we propose different features which could enhance the implementation.

5.1 Performance Testing

To assess the quality and scalability of the MapTogether prototype, we conduct several stress tests on the MapTogether server. For the stress test, we make about 50 requests per second for 5 minutes, with a wind up and down of about 30 seconds. The requests are HTTP GET requests on the user endpoint to retrieve user “1”. We conduct a stress test where the requests come from Frankfurt in Germany and one where the requests come from Ashburn in the USA. The server is hosted in Helsinki in Finland, so the response time to Germany should be faster than to the US.

A summary of the results is shown in Table 5.1. The table shows the response times (average, minimum and maximum) and the standard deviation of the response times. The P99 value tells us what 99% of response times will be at most. Fig. 5.1 and Fig. 5.2 show the response times (minimum and maximum) and the request rates during the tests.

Client Location	Avg. RT.	Min. RT.	Max RT.	Std. Deviation	P99
Germany	90 ms	38 ms	357 ms	29 ms	186 ms
USA	158 ms	107 ms	337 ms	26 ms	256 ms

Table 5.1: Results of two stress tests, sending about 50 requests per second from Germany and USA. The test measured response time (average, minimum, maximum), and the standard deviation. 99% of response times are below the P99 value.

The results show us that the response time, as expected, depends on the location of the clients. We can see that the server is able to handle 50 requests per second with a maximum response time of 357 ms, with no errors or faults. It is difficult to assess how many requests realistically would happen every second, if MapTogether were released. In order to make a valid estimation, we would have to do an experiment to see how often a user would request the server while using the application. We would think that handling 50 requests per second is pretty good but might not handle it well if something would cause a large portion of the player base to use it simultaneously. Because of the relatively small number of contributors, we think that handling 50 requests per second is enough. Not everyone will be using the application simultaneously due to different time zones, and not everyone will make requests at the same time.

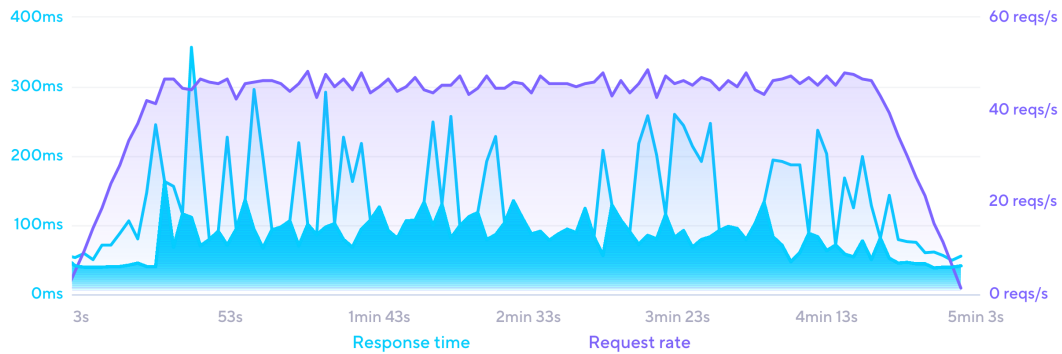


Figure 5.1: Graph showing the request rate and response time (minimum & maximum) in the course of the 5 minute long stress test. The request were sent from Frankfurt, Germany

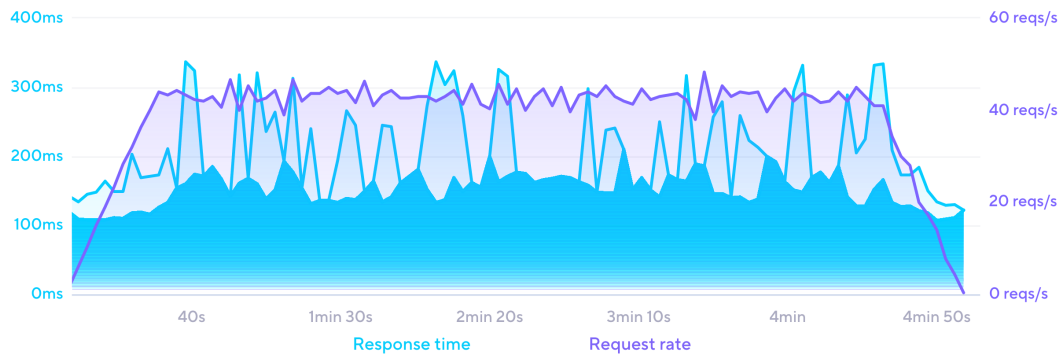


Figure 5.2: Graph showing the request rate and response time (minimum & maximum) in the course of the 5 minute long stress test where the request was sent from Ashburn, US.

To increase the responsiveness of the server, we could use replication to scale horizontally. If we replicated the server itself, the clients' requests should be evenly distributed between the servers. This is easy to do because the server implements a REST API. If we did this we could also replicate the database itself. This would require some protocol for synchronisation that ensure correctness across the database replications.

5.2 Discussion

In this section, we evaluate our work in this project and results on the problem statement from Section 2.7 and requirements from Section 3.2. We will also describe and evaluate our development process.

5.2.1 Fulfilment of Requirements

Requirement 1 aims at the core functionality of the application and is supported through the quest feature implemented in Section 4.7.3. We implemented two simple quest types, but more quests must be implemented for the application to be complete. Similar quest types are trivial to add because the implementation is almost the same.

We incentivise users to contribute to OpenStreetMap by rewarding points for completing quests. This could potentially incentivise users to complete quests quickly, and possibly answering dishonestly, going against Requirement 2. We have not implemented checks in

the application to verify if a contribution is legitimate or not. However, we designed the activities so that users should not be pressured on time, because it could incentivise users to recklessly answer quests they are not sure about.

The user interface has been designed for adding POIs as shown on Fig. 3.11. Our OpenStreetMap API wrapper also implements this feature. However, we have not implemented this functionality in the backend of the client yet. This means that the functionality to address Requirement 3 is missing.

We fulfil the Requirement 5 by having multiple different leaderboards that have time segments, such as weekly leaderboards. However, the designed achievement feature is not implemented.

We partially have solved Requirement 6 by implementing following, follower list and leaderboards. However, we do not incentivise the user to invite new people to contribute, beyond the incentivisation of being able to compete with them on the MapTogether leaderboards.

We fulfil Requirement 4 with a single client codebase thanks to Flutter. The application works on both Android and iOS. We also achieve Requirement 7 through the use of Flutter. With Flutter, we develop the Android and iOS versions simultaneously and with similar user interfaces. There are very slight differences such that the application meets the different design practices of iOS and Android. An example is the “go-back arrow” on iOS is just an angle as opposed to an arrow with a tail on Android.

Offline capabilities, mentioned in Requirement 8, is also not addressed. However we planned on how to achieve it in Section 4.1. The data is always retrieved from the web and never stored locally.

5.2.2 Work Process

In the first part of the project, we, as students, had many small time slots to work on the project interrupted by lectures. We planned the tasks in increments of a week to keep an overview between the lectures. When planning the workload for a week, we used a kanban board, which we also used in our daily stand-up meetings. This helped us get an overview at the beginning of each day. In the final weeks of the project, we started to plan more than a week ahead because the tasks were more easily defined, and we had a clearer idea of what we had to do. This worked well near the end because we had no lectures (less interruption) and thus it was easier to keep track of the plan.

In each sprint, everyone had a specific task to begin with, and when done they could pick any from the backlog to continue with. Once we got closer to the hand-in date, all tasks in the sprints were assigned to a person, so that everyone knew exactly what they had to do throughout a given sprint. Tasks would mainly consist of either researching a topic, writing a part for the report or creating a specific feature for the implementation. The latter two of which we will more closely examine the work process for. Whenever a part of the report had been written, it would be assigned as “ready for review”. Once reviewed/proof-read by another member of the group, it would be either marked as completed or changes would be requested to the initial author.

In regards to our implementation, we used git as a version control system. Through git, we practised continuous integration, in an effort to distribute the programming workload. While still ensuring that no single branch would remain without integration to the master for a prolonged period. Whenever a new feature was to be implemented, we made a feature branch for it in our git repository. The purpose of using feature branches was to give a highly-focused

purpose to each branch in the repository. This ensured that the purpose, and by extension lifetime, of a given branch, was readily apparent from its creation until integration to the master. Features were kept as small in scope as possible, to once again ensure continuous integration in the project. The development process we used for implementation was great, since it opened up for us developing independent features in parallel.

Once the feature of a given branch was completed, a pull request would be made to the master branch. This would then prompt the other members of the project to initiate a code review of the feature, before integration to the master. Upon review, changes would then either be requested for the initial owner of the branch, upon the completion of which another review would initiate, or the pull request would be accepted, integrating the feature into the master branch.

5.3 Conclusion

In this section, we summarise and conclude the work and results of the project. We analysed different aspects of contributing data to OpenStreetMap and what kind of problems become apparent when the data relies solely on volunteers. We analysed existing software for contributing to OpenStreetMap and gamification elements to motivate different types of players.

The analysis ended with a problem statement from which we described some people that could be a potential audience. From their perspective, we formulated requirements for a mobile application.

We designed the core game loops and features to motivate different users. We used object-oriented analysis and design to analyse and model the problem domain and to identify and plan functions of the application domain. We used those functions to design an interface to support them.

After the design, we mapped the architecture of different components which would be needed and implemented the base features. We focused on implementing the *minimum viable product* as described in Chapter 4, but also on implementing what would provide more learning value for us. This is instead of focusing on what would be most important if the application were to be deployed.

Firstly, we implemented the client, a cross-platform mobile application and a quest system with two quest types. Secondly, we implemented an API for OpenStreetMap to download and upload map data and contributions. Lastly, we implemented a database and a server to host and manipulate the database and an API that the client use to access the server.

The result is a prototype of MapTogether. The app implements the base features we wanted it to have, and all components work together in accomplishing the tasks that we implemented. Many features are still yet to be implemented before MapTogether is a complete product, for example, waypoint activities and surveying quests. Nevertheless, we are satisfied with what we accomplished and learned during the analysis and development of MapTogether and believe it has the potential to be improved and become a very usable alternative to StreetComplete and a complement to other contribution methods.

5.4 Future Works

The following section describes the features that need to be implemented in order to complete the application.

5.4.1 Verification Quests

As mentioned in the discussion in Section 5.2, the competitive aspect of MapTogether could make users reckless and add wrong data. We could implement some kind of verification system to compensate for this.

One way to implement this is through verification quests. A verification quest asks a user whether some contribution of a different user is correct. The result would be noted and using the MapTogether database we could keep track of which contributions are corrected/-confirmed and by how many. This data could be used to measure the correction rate (the chance that a users contribution is corrected), and possibly prioritise verification quests from users with a high correction rate (without excluding others). The verification data could also be used to withdraw points from users if they are consistently being corrected because of wrong data. It is possible to make honest mistakes, and some analysis is required to further identify how to properly implement the mechanism for withdrawing points. Another possibility is to notify a MapTogether user that their data was corrected or confirmed. This could be a way to let new contributors know if they made mistakes or misunderstood some convention, or make users happy by letting them know that others agree their contribution was good. Conversely, it could be a bad idea because a new user being corrected could make them less motivated to continue making contributions, not to mention the verifying user could also make a wrong correction.

The MapTogether client would analyse the OpenStreetMap changesets and choose certain changesets for which to open verification quests mostly for recent contributions. This would possibly requires assistance from the MapTogether database because the client would need to know about the users' correction rate. The user cannot verify their own contributions. Completing a verification quest would notify the MapTogether server which would in turn note the confirmation/correction. It might also be possible to annotate the corrected changeset with some label that tells us it was corrected.

5.4.2 Tile Renderer

One of the requirements is that MapTogether should work with minimal internet access. One thing that ties into this is visualising the map data. The prototype developed in this project always downloads the map tiles from an OpenStreetMap tile server. There are two ways to circumvent this. Cache the tiles locally, or rendering the tiles from the raw map data. The advantage of rendering the tiles locally is that we can visualise the tiles as we please, however it is much more complex to implement. We could more easily use existing renderers if any are implemented in dart. If it is too difficult to implement a renderer locally in the client, we could host our own tile server and then cache the tiles on the client.

5.4.3 Activity Motivation Experiment

In order to identify how well activities actually motivate users, we could conduct a field experiment. The null hypothesis of one such experiment could be: "Activities affect the motivation of the users". In the hypothesis, we can identify "Motivation" as the only dependent variable, and "Activities" as an independent variable (whether activities are available or not).

We can test the hypothesis with a between-group experiment, where participants are divided into two groups, “A” and “B”. Participants in group A receive a version of MapTogether with a working activity feature, and participants in group B receive a version without any activity feature. All participants must participate in the experiment as small groups of friends (2-10 people) so that they are able to map together if they want to.

The participants are tasked with using the app at least once a week for three months. Play time and quests completed should be measured throughout the experiment. We assume that the more time the users spend, and the more quests they complete, the more motivated they are to use the application. After the experiment, we could ask the participants “How likely are you to use this application again in the future on a scale of 1 to 5 where 1 is not likely and 5 is very likely?”.

It is important that all participants get an equal introduction to OpenStreetMap and MapTogether so that no participants receive a more motivating introduction. This also ensures that all participants have a minimum understanding of OpenStreetMap. We should also be aware of previous experience with OpenStreetMap because it might affect their motivation.

The result of the experiment should be an understanding of how activities affect the users’ motivation (positive, negative or non-significantly). This knowledge could be used to decide whether more development resources should be spent on developing activities, or if development should focus on other features.

Bibliography

- [1] J. Nielsen, “The 90-9-1 rule for participation inequality in social media and online communities.” <https://www.nngroup.com/articles/participation-inequality/>, 2006. Retrieved 24th of February.
- [2] J. Anderson, D. Sarkar, and L. Palen, “Corporate editors in the evolving landscape of openstreetmap,” *ISPRS International Journal of Geo-Information*, vol. 8, no. 5, 2019.
- [3] A. Yang, H. Fan, and N. Jing, “Amateur or professional: Assessing the expertise of major contributors in openstreetmap based on contributing behaviors,” *ISPRS International Journal of Geo-Information*, vol. 5, no. 2, p. 21, 2016.
- [4] A. for Data Supply and Efficiency, “Datadistribution.” <https://eng.sdfe.dk/datadistribution/>, 2018. Retrieved 25th February.
- [5] OpenStreetMap, “Contributors.” <https://wiki.openstreetmap.org/wiki/Contributors#Denmark>, February 2021. Retrieved 25th February.
- [6] E. O. System, “Satellite data: What spatial resolution is enough?.” <https://eos.com/blog/satellite-data-what-spatial-resolution-is-enough-for-you/>, April 2019. Retrieved 25th February.
- [7] L. Juhász, T. Novack, H. H. Hochmair, and S. Qiao, “Cartographic vandalism in the era of location-based games—the case of openstreetmap and pokémon go,” *ISPRS International Journal of Geo-Information*, vol. 9, no. 4, 2020.
- [8] Manuel S Pascual, “Gis data: A look at accuracy, precision, and types of errors.” <https://www.gislounge.com/gis-data-a-look-at-accuracy-precision-and-types-of-errors/>, November 2011. Retrieved 25th February.
- [9] OpenStreetMap, “Accuracy in openstreetmap.” <https://wiki.openstreetmap.org/wiki/Accuracy>, December 2020. Retrieved 25th February.
- [10] OpenStreetMap, “Accuracy of gnss data.” https://wiki.openstreetmap.org/wiki/Accuracy_of_GNSS_data, June 2020. Retrieved 25th February.
- [11] A. F. Aparicio, F. L. G. Vela, J. L. G. Sánchez, and J. L. I. Montes, “Analysis and application of gamification,” in *Proceedings of the 13th International Conference on Interacción Persona-Ordenador*, INTERACCION ’12, (New York, NY, USA), Association for Computing Machinery, 2012.
- [12] “Streetcomplete.” <https://wiki.openstreetmap.org/wiki/StreetComplete>. Retrieved 26th February.
- [13] I. Celino, D. Cerizza, S. Contessa, M. Corubolo, D. Dell’Aglio, E. Della Valle, and S. Fumeo, “Urbanopoly – a social and location-based game with a purpose to crowd-source your urban data,” in *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*, (Amsterdam, Netherlands), pp. 910–913, IEEE, 09 2012.

- [14] B. Kim, *Understanding gamification*. ALA TechSource Chicago, 2015.
- [15] Janaki Mythily Kumar, Mario Herger and Rikke Friis Dam, “Bartle’s player types for gamification.” <https://www.interaction-design.org/literature/article/bartle-s-player-types-for-gamification>, July 2020). Retrieved March 15th.
- [16] G. Tondello F, R. Wehbe R, L. Diamond, M. Busch, A. Marczewski, and L. Nacke E, “The gamification user types hexad scale,” in *CHI PLAY Companion ’16: Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts*, (New York, NY, USA), pp. 229–243, Association for Computing Machinery, October 2016.
- [17] S. Hatton, “Choosing the right prioritisation method,” in *19th Australian Conference on Software Engineering (aswec 2008)*, pp. 517–526, IEEE, 2008.
- [18] M. Sicart, “Loops and metagames: Understanding game design structures,” in *Foundations of Digital Games 2015*, 2015.
- [19] A. Ballatore, “Defacing the map: Cartographic vandalism in the digital commons,” *The Cartographic Journal*, vol. 51, no. 3, pp. 214–224, 2014.
- [20] L. Mathiassen, A. Munk-Madsen, P. A. Nielsen, and J. Stage, *Object-oriented analysis & design*. Metodica AoS, 2000.
- [21] G. C. Blog, “The difference between classic bluetooth and bluetooth low energy.” <https://blog.nordicsemi.com/getconnected/the-difference-between-classic-bluetooth-and-bluetooth-low-energy>, May 2021.