IN2000- Group project

Members:

Bavo Meyvis (bjmeyvis)

Marius Dorgli (mariusrd)

Tobias Fromgren (tobiafro)

Jørgen Henriksen (jorgenwh)

Bjørn Harbo Dahle (bjornhd)

Advisor: Isak

Contact: Gaute

Case: Air quality in cities (4)

University of Oslo

IN2000- Air pollution - Group 58

Table of Contents

Short Summary

In short, our group initially got together later than we should have allowed. It took longer than expected to get started on our project. Although we quickly agreed on a project case and began discussing different approaches to our project case, it took longer to actually execute what we had discussed.
This and future hinders would make it hard to achieve a polished product within the deadline.

We originally sat down to figure out when each group member was available to meet and decided to have weekly mandatory meetings. Thereafter, we speculated on which tools to use for keeping everyone in the loop as well as for planning. Slack would be used for chatting and discussions, Discord for meetups online and this very document for planning. Teamviewer and Trello would become additional helpful tools.
Not all of the tools we intended to use were being used properly by every group member, which in hindsight resulted in some tools being less useful than others.

We chose project case 4, "Air quality in big cities", which we named "Pollution".
The initial purpose of our project was to simply get and display air quality data for any point (especially the big cities) in Norway, and implement a functions like the ability to compare or favorite cities.
Google's Google Maps API was chosen to represent the map of Norway, and some other libraries like Retrofit and Moshi were discussed early on.
We did not really suffer from disagreements regarding implementations or goals, but rather we experienced other drawbacks. If communication was not adequate, members could unknowingly do something that was not initially agreed upon. Other times, members could work on functionalities which previously had been replaced, disregarded or simply discarded.

Goals were set, tasks were assigned, hinders were discussed.
By the end of each sprint everyone was updated on the weekly meeting unless cancelled.
Certain modules took longer than anticipated to implement, and when finally implemented, we often encountered problems like features not working exactly as we had anticipated.
Some found the progress to be moderate, others slow. Some are happy with the result, some not.

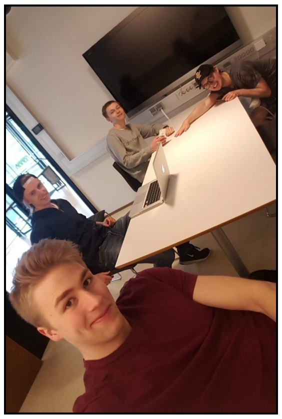
One thing is certain: we all learned from this informative experience and are thankful for the assignment as well as the challenges that came with it!

Presentation

We collectively decided on basing our project off of project case 4, which in short was to create an app to present air quality of big cities in Norway on a map. We would continue to use this project case as guidelines for our own project. We named our app “Pollution”, simply to reflect the fact that it presents pollution in the air in Norway. We will explain the initial mentioned functionalities for our app.

The Team

The team might not be flawless; everyone sat on the fence at one point. Not only did we experience misunderstandings, there were also several underestimations throughout. Tasks were believed to be easier than expected, but eventually proved to be harder than initially estimated. The team is a far cry from being perfect. On the other hand, you can argue the team to be just perfect for a good experience during this project. The purpose of this assignment was to learn from the experience of working in groups. This team has probably been one of the best at encountering hinders which in turn has given us tenfold more experience. Furthermore, every group member has also been polite, friendly and considerate, which is of the utmost importance for an experimental group.



Group photographs. Below is a short description of each team member in the order of the first of the above pictures.

Bavo

My main traits are solidarity, integrity, creativity.
Without a second thought, I will drop everything to support my group.
You can trust me not to lie. Set me free and I will be blunt, a true candour.
However, don't be too quick to judge. In the end, I'm still a mensch. An unfledged one that is.
Always will I remain antic and gregarious. Take it with a grain of salt.

On a final note, dark mode is my doing.
Just couldn't help myself.

Marius

I personally think I'm a pretty average worker, I do not necessarily put in the extra hours to get things perfect but do the minimum at least. I consider myself an honest person, and I try to meet up when we have a meeting. I always try my best to complete work within the sprints we have. I tend to be the type of person who procrastinates until there is little time left and then work a lot during the time left before the deadline. Luckily my group members are not procrastinators, which has helped motivate myself to work more evenly throughout the semester.

Tobias

I consider myself a diligent worker; I can sit with a task ‘till the end of of dawn. My defining traits are politeness, patience, and I'm always happy to learn new stuff. I also like to crack a joke here and there. These traits proved to be valuable, as I found myself with too much on my plate during this semester. Much attention which should have gone to the project was allowed to be absorbed by other tasks, and the effort balance between them was poor; I worked much on another subject for the first half of the semester, and had to commit harder to this subject for the second half. Nonetheless, I still felt it was important to do my part as a member of this group and usually provided input for the project, especially during meetings. I tried to always be available for help upon requests.

Jørgen

I always feel pressured into working on whatever issue we currently have at hand. I pay a lot of attention to the overall architecture, which has been helpful for certain aspects of this project such as deciding and suggesting realistic designs for the application overall. I consider myself to be polite and friendly, and I believe my group would agree.

Bjørn

I really enjoy figuring things out on my own, but this also sometimes leads me to not ask for help early enough. I tend to work in bursts of energy, and I regularly lapse into procrastination, especially if either I don't find the task at hand to be sufficiently interesting or stimulating, or if there is not much external pressure on me (e.g. upcoming deadlines, or other people depending on my work). In the early stages of the project I tried to use my experiences from my first two semesters at “design, bruk, interaksjon”. I thought they could be useful, steering the discussion of the group in terms of what was the purpose of what we were making, who would be our intended audience, and so on. I thought, rather discuss this than to jump straight into the technical details and implementations. Thinking more in terms of what and why, before we started getting too focused on (and limited by) the how.

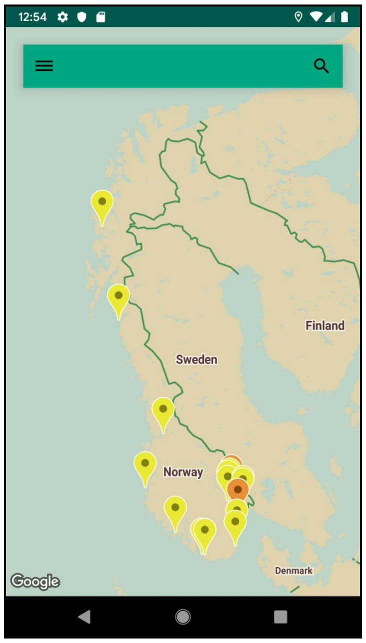
User Documentation

Main Functionalities

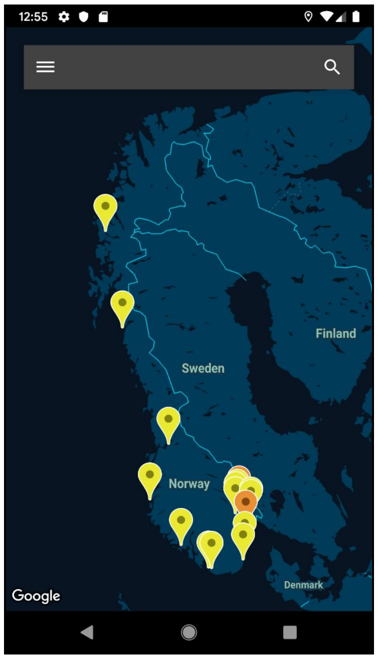
Map Activity

A map overview of Norway with colour overlays representing what the air quality is like in any given area/big city. Over each big city, the map should display a colour overlay according to the air quality levels retrieved from api.met.no. The colors were green, yellow, orange and red, representing good, fair, moderate and bad, respectively. The coloured overlays would be clickable by the user to allow the ability to enter a new activity with more detailed information about this particular city.

Our original vision for this functionality differs from our end product. With lack of progress in creating appropriate colour overlays, we decided to go with a new design for the map view functionality. The new design would place markers, coloured as mentioned before, over each large city on the map covering Norway. See pictures below.



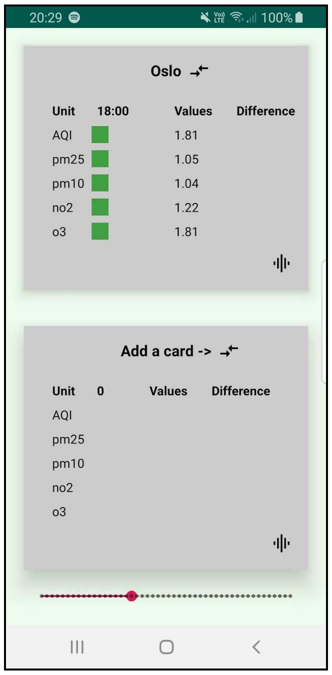
Light map style.



Dark map style (with Dark Mode).

Forecast Activity

An activity accessed through the map overview activity, presenting the user with more thorough data of the air quality in a specific area. This activity would have a headline stating which area or city in Norway the user is currently viewing air quality data from.
It would have colour coded fields (similar to the markers) next to numerical air quality values. The user could choose what time of the day to view by dragging a bar, which in turn would display the air quality data for that hour.
In this activity we also planned a “compare” functionality. Here the user could select another area or city and compare it alongside the initially chosen city or area with the same units and values. In addition, a new feature would show the difference between their counterpart’s value for every unit. Both places would be dependent on the same slider and would therefore show values for the same time. The design of this module is more or less equal to our originally planned design.



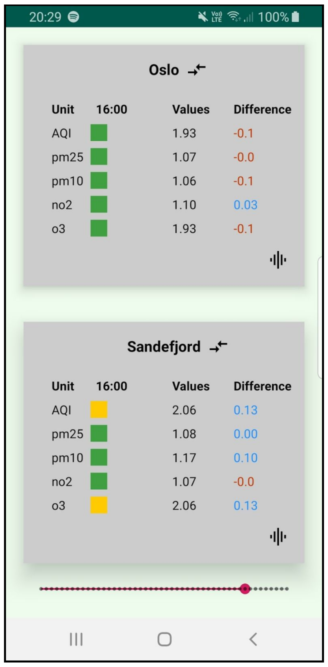
Oslo

Unit	18:00	Values	Difference
AQI	🟩	1.81	
pm25	🟩	1.05	
pm10	🟩	1.04	
no2	🟩	1.22	
o3	🟩	1.81	

Add a card ->

Unit	0	Values	Difference
AQI			
pm25			
pm10			
no2			
o3			

Single city selected.



Oslo

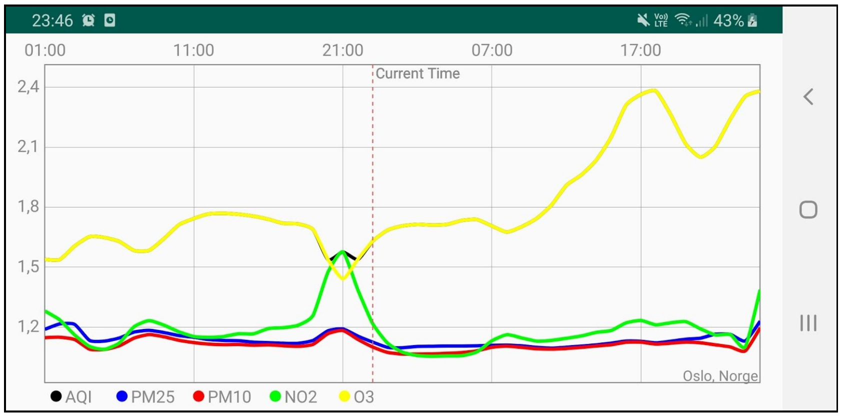
Unit	16:00	Values	Difference
AQI	🟩	1.93	-0.1
pm25	🟩	1.07	-0.0
pm10	🟩	1.06	-0.1
no2	🟩	1.10	0.03
o3	🟩	1.93	-0.1

Sandefjord

Unit	16:00	Values	Difference
AQI	🟨	2.06	0.13
pm25	🟩	1.08	0.00
pm10	🟩	1.17	0.10
no2	🟩	1.07	-0.0
o3	🟨	2.06	0.13

Two compared cities selected.

Graph Activity



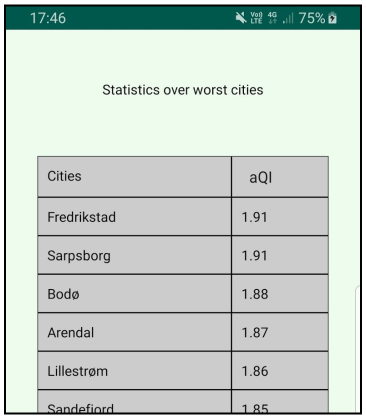
Similarly to the previously mentioned activity, we wanted to have an activity showing the same air quality data in the form of a graph. This activity would be a preference alternative to the forecast activity, giving the user an option to view the data in a different format. The graph activity would be accessed through the forecast activity, and would present exactly the same API returned values over the same period of time. We also wanted the option to return back to the forecast activity which again would display the same data, making the graph activity something like an optional toggle for how to display the presented AQI values.

The graph's X-axis represents hours starting at 01:00 (as this is the first hour you get air quality data for from the api) and continues 48 hours onwards, with a narrow dash indicating the current hour. The Y-axis represents the AQI values corresponding to the hour indicated.

The graph also has legends at the bottom explaining which colours are correlated to which AQI values, as well as the address of the chosen location.

The graph activity turned out more or less the way we had planned. Picture of graph underneath.

Statistics Activity



Statistics over worst cities

Cities	aQI
Fredrikstad	1.91
Sarpsborg	1.91
Bodø	1.88
Arendal	1.87
Lillestrøm	1.86
Sandefjord	1.85

An activity simply presenting the user with the ten cities with the worst, user-chosen, pollution unit values at the current hour and sorting them descendently. A screenshot of the Stats activity displaying statistics for the 10 worst AQI levels is shown below.

Favourites Activity

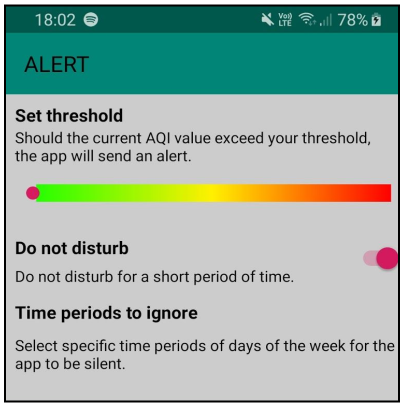
We also wanted to make the app more customisable. Therefore we thought it would be relevant for users to have the option to favourite one or several cities, and store these for easier access to information about these particular cities. We also planned to let users add AQI value thresholds for every favorited city. These thresholds would be used by an alert system to notify the user through a push-notification, informing the user that the AQI level has surpassed the user-defined threshold for this specific area. Due to the deadline, this was never implemented.

Alerts Activity

From an early stage, we agreed that an alert system would be one of our essential functionalities. We wanted the user to receive a notification when the current location's AQI was lower than a specified value (usually referenced as the threshold).
In the settings the user could turn off alerts, in which case they would not trigger unless the user turned them back on again. Furthermore, the alert system had a dedicated activity launched from MapsActivity, called "Alerts". Here, the user could adjust the threshold value and tweak other notification settings. There are two ways to limit notifications: do not disturb and daily time scopes. The user can temporarily enable a do not disturb mode, which would disable all notifications for a user-specified period of time. Do not disturb can be activated for one hour, three hours, one day, one week or one month.

Through what is called WeekActivity - accessed through AlertActivity, the user can restrict alerts within specified time-scopes for every day of the week. Finally, it was planned to have functionality that could exclude certain locations from receiving alerts, however, this function never made it to the final product because of time limitations.

As a whole, alerts are very dependent on location services. Without them, the application's alert system simply will not work; consequently, location services were integrated into MapsActivity. On the initial startup, the user is requested to allow location services with the additional option "do not ask again".


ALERT

Set threshold
Should the current AQI value exceed your threshold, the app will send an alert.

Do not disturb
Do not disturb for a short period of time.

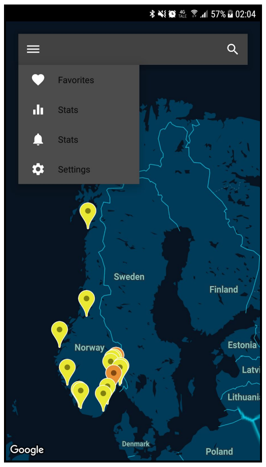
Time periods to ignore
Select specific time periods of days of the week for the app to be silent.

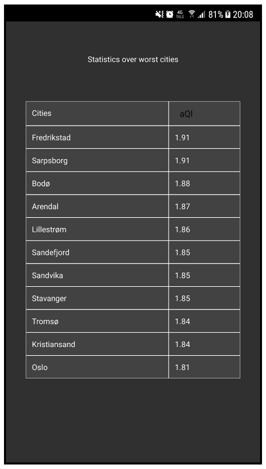
A screenshot of the Alerts activity where users can customise their alert preferences.

Other Features

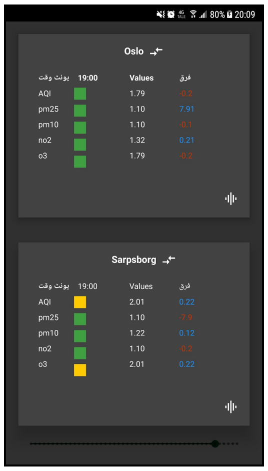
Dark Mode

As dark mode is a staple feature in most modern apps, we decided this was a necessity for our app and began working on an implementation. The option to switch to dark mode will darken all activities as well as the Google Map style in the app to a classic dark theme. The setting to switch is a toggle button that will revert the previous theme change whenever interacted with. The theme preference is saved on change using the storage implementation Storage, and thereby the user is always presented with the previous applied theme. See pictures underneath.




Statistics over worst cities

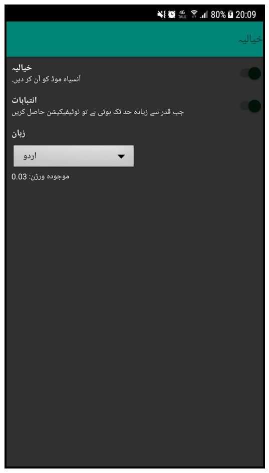
Cities	aQI
Fredrikstad	1.91
Sarpsborg	1.91
Bodø	1.88
Arendal	1.87
Lillestrøm	1.86
Sandefjord	1.85
Sandvika	1.85
Stavanger	1.85
Tromsø	1.84
Kristiansand	1.84
Oslo	1.81


Oslo

یونٹ وقت	19:00	Values	فرق
AQI	🟩	1.79	-0.2
pm25	🟩	1.10	7.91
pm10	🟩	1.10	-0.1
no2	🟩	1.32	0.21
o3	🟩	1.79	-0.2

Sarpsborg

یونٹ وقت	19:00	Values	فرق
AQI	🟨	2.01	0.22
pm25	🟩	1.10	-7.9
pm10	🟩	1.22	0.12
no2	🟩	1.10	-0.2
o3	🟨	2.01	0.22

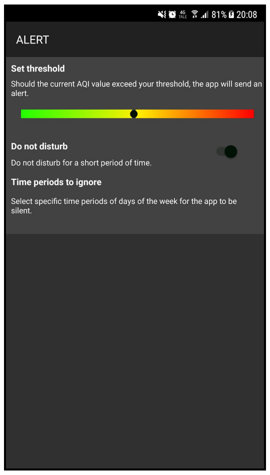

خیالیہ

خیالیہ
آنسیاہ موڈ کو آن کر دیں.

انتباہات
جب قدر سے زیادہ حد تک ہوتی ہے تو نوٹیفیکیشن حاصل کریں

زبان
اردو

موجودہ ورژن: 0.03


ALERT

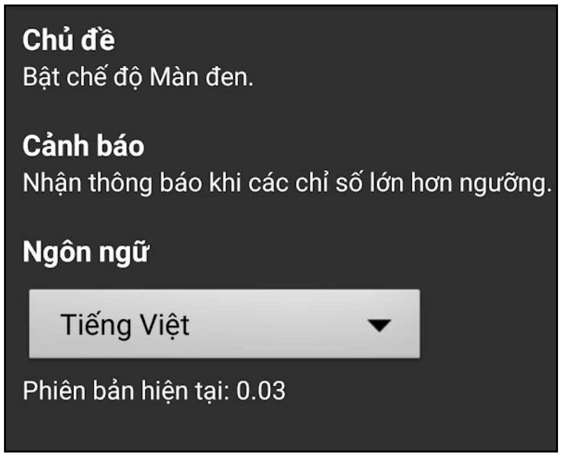
Set threshold
Should the current AQI value exceed your threshold, the app will send an alert.

Do not disturb
Do not disturb for a short period of time.

Time periods to ignore
Select specific time periods of days of the week for the app to be silent.

Language Support

We made it clear from the beginning that universal design is one of our primary concerns. It was important for us to allow everyone to easily navigate the app. A way to express this was by offering support for more languages than English; as a result, we have given room for functionality that lets the user pick a new default language to display information in our app. In addition to Norwegian, we thought that it was important to address "Nynorsk" in order to satisfy most Norwegians. It was also speculated to extend language support to Sami in order to fully reach out to all of Norway, but none of us was confident in translating to Sami correctly and neither did we know anyone who could help us out on this one.
We also added language support for Dutch, Afrikaans, Urdu, Spanish and Vietnamese.
The translations were conducted mostly by members of the group, but we also received help from a team member's friends, resulting in proper Urdu and Vietnamese translations.



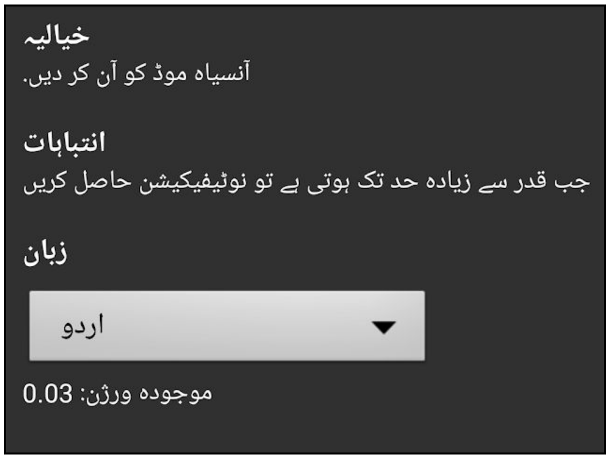
Chủ đề
Bật chế độ Màn đen.

Cảnh báo
Nhận thông báo khi các chỉ số lớn hơn ngưỡng.

Ngôn ngữ

Tiếng Việt

Phiên bản hiện tại: 0.03



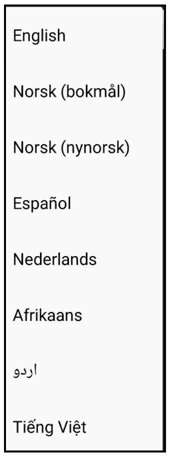
خیالیہ
آنسیاہ موڈ کو آن کر دیں۔

انتباہات
جب قدر سے زیادہ حد تک ہوتی ہے تو نوٹیفیکیشن حاصل کریں

زبان

اردو

موجودہ ورژن: 0.03



English
Norsk (bokmål)
Norsk (nynorsk)
Español
Nederlands
Afrikaans
اردو
Tiếng Việt

Here is a demonstration where a user changes the language from English to Urdu.

Focus on Norway

As our project case was to create an app displaying air quality data in Norway, we wanted to somehow make sure that it would be clear to the user that the application focused exclusively on Norway. We assumed it would suffice to make sure that when the maps activity was launched it would only show colour coded displays of the air quality values over areas in Norway, and the map view would be centered above Norway, as well as limiting the movement of the map to other countries. This feature underwent a lot of changes, perhaps second to the map itself. It was later completely overhauled. The process of developing this feature is written in detail later in this document, and a few screenshots are also provided.

Easter Egg Game

As a fun and pleasant surprise, we have also dedicated resources to implement a hidden easter egg game. We will not go into detail on the implementation, because that will destroy the point of a hidden easter egg game. Therefore, we encourage you to try to unlock it and play it. If you try to find it in the codes, you will soon learn that it is very well hidden, in fact, it is so secretive that it is even classified to most of us developers. But it is nevertheless a very big game, and if you find yourself uncertain whether to grade us between a lesser and a greater grade, keep this huge game in mind. We would also like to give a special thanks to Zahra, Lê Khánh Tùng and Bravo for the translations.

Information

The main focus of this project has always been to bring information to the people. No matter the nationality, users should have the possibility to perceive air quality and pollution values in Norway. We hoped this would help users make better choices in life or satisfy their curiosity regarding the pollution. In addition, we wanted to raise awareness concerning pollution in general, as some users might find the information useful in their quest for a cleaner planet. This was our first target group, because we thought they would be the most interested in our app. We had the impression that pollution was most relevant when talking about a greener planet, thus we thought it only to be natural that those interested would use our app. Maybe they would use it to prove a point, support an argument, learn something new, or anything else regarding a greener planet. Nonetheless, we believe that the data we provide through our app is relevant for a greener planet.
However, if this was not the case, then we would need some kind of fallback.

Discussing our second target group, we asked ourselves what kind of people could be our target group if not those interested in a greener world, and concluded that self-interest is ingrained in all of us.
Therefore, people who wanted to use it for personal motives, may it be family related or whatnot, could be interested. Some examples include those who want to move to Norway or

those who want to move to a different city within Norway, people who are sceptical to the air quality, or people affected by asthmatics. With these two groups, we felt we could establish some requirements and eventually have some interviews. But before we could do that, some other questions remained unanswered, like which platform it would run on, how you would access the app and which version was most viable for our purpose.

Free To Use

We already knew from the beginning on that the app would not be labeled for commercial use. We never had any intent to commercialise it since this is a project on a rather small scale with the purpose of learning about group projects. And even if we wanted to make money, we did not believe ourselves to be ready or capable of making a good product for the public, which is why we first need experience and practice. Commercialisation is not in our interest and we do not have any implementations to tell a pirate from an ordinary user. It would, however, be sad if our application was pirated and distributed under a different name (or commercially for that matter), because we still want the recognition - credit where credit is due.

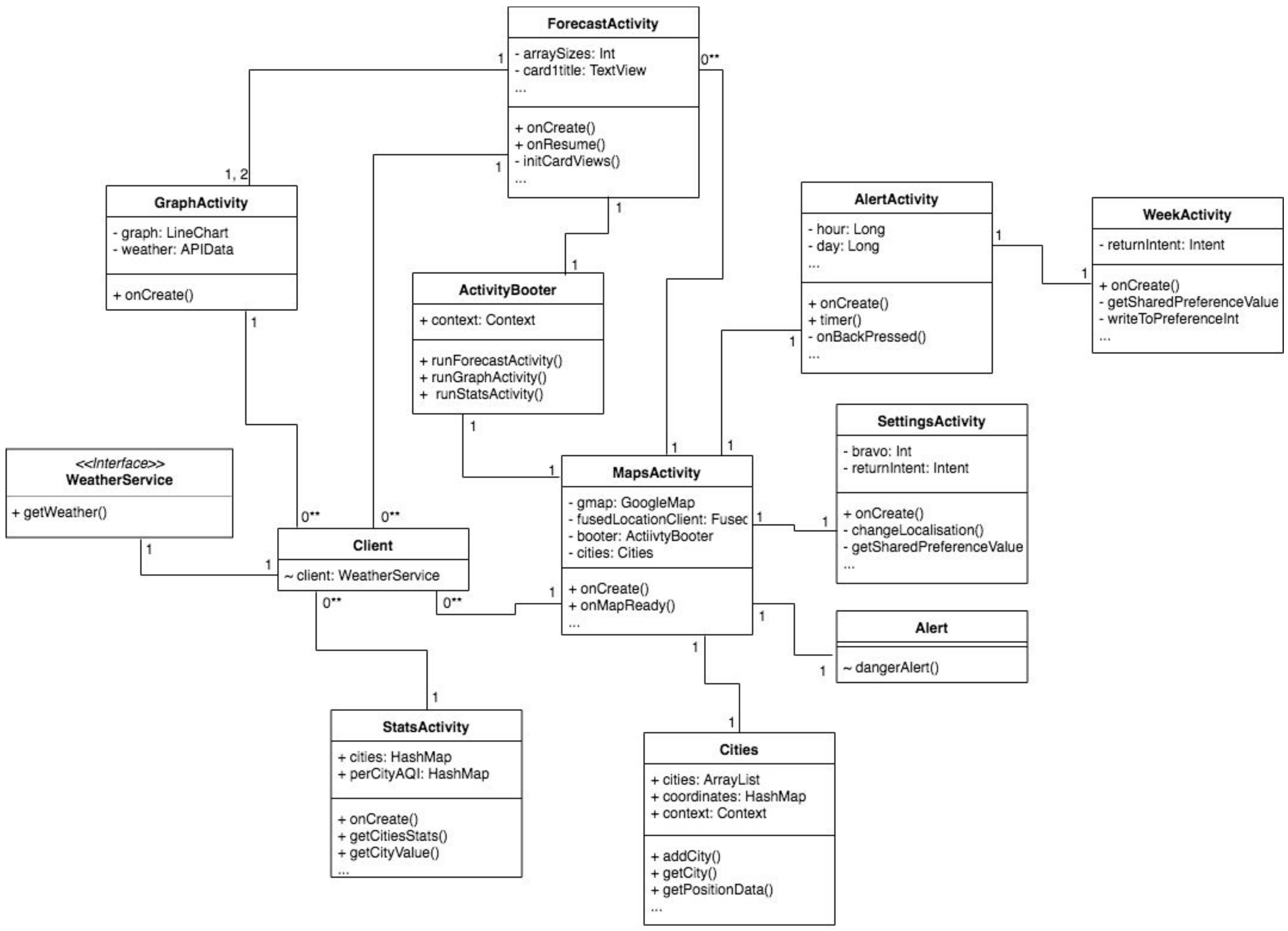
API Level

We were very fortunate in that almost all of our group members had Android phones, which enabled us to easily test the app on an actual device, rather on just the emulator. Our minimum API level has been set to 23. We believe Marshmallow to be sufficient since we wanted to have something not too new and not too old. According to our editing application, Android Studio, it would run on 62% of all devices. Unfortunately, that means some people might not be able to run our app, but for the sake of simplicity, we did not want to use a lot of time on the old ways of coding. Some members also had a strong interest in seeing what the higher API levels had to offer in terms of functionality. As mentioned previously, we wanted everyone who was interested to be able to use our app, which means that if this project had been on a bigger scale and not for learning purposes, we would definitely have considered a lower API level to support these older android phones.

On the other hand, even though we did not choose to support the older android API levels, there was consensus about implementing a user-friendly interface, multiple language support and perhaps scalable text. The interface is especially important for older people or anyone struggling with navigation in an android app. Although we did not follow material guidelines, the app should be rather easy to navigate through. We used contrasting colours to easily read text and elevation of cards containing text to give the impression of sections (made of cards), in for example the forecast activity. From experience, we were aware of the impact poor placement of functionalities has on touch screens. We discussed several different design options to avoid dysfunctional and tedious user interfaces.

Requirement Specification and Modelling

In this part of our report, we will dive into our main functional requirements, how we came to model and design them, how successful our modelling and design pattern usage was, and what we wish we would have done differently.



This is our class diagram. Only some of the attributes and methods are included, "..." indicates missing attributes/methods.

Modelling

Our modelling strategy on designs happened on the go, so to speak. We frequently changed- and created new models. In hindsight, modelling was something we should have spent more time considering. If we had been able to re-do the project case, we would have focused more on modelling. Our reasoning for why the modelling aspect of our project was not as thoroughly planned as it should have been is that everyone in our group thought of thorough modelling as some sort of a foreign idea. Therefore, our ability to create a good design pattern and hierarchy without following more strict modelling ideas was overestimated.

Modelling Strategy and Design Pattern

Our modelling idea was straightforward and simple: we would mainly focus on finding a solution that would work without worrying too much about whether it was the best solution to the problem. This resulted in one of our main activity classes to contain large amounts of functions and variables that had no real specific connection to this particular activity. In other words, we had created a god object. We eventually did an overhaul of these misplaced functionalities and variables, and cleaned up the structure of some of the app's classes.

In the early stages of the project, everyone mostly worked on their own modules. Everyone predominantly followed an idea resembling the design pattern "Model-View-Controller" (MVC) for each individual module or activity. This meant our standard design strategy was to develop a model with backend functionality which would fetch/calculate the correct data we wanted to present to the user. A view would then present this data to the user.
Assuming the activity in question would take input from the user to manipulate the presented data, we chose a design and implementation for a controller. The controller would let the user interact with the module in question, manipulating the data by altering the returned presented data.

Blunder and Miscalculation

One of our mistakes early on was that we never properly planned how our different modules would communicate. This resulted in our solutions being less than optimal when we eventually approached the point where we would connect our different modules.

Furthermore, we never thoroughly considered design patterns or how exactly we wanted to shape our modules. We would use wireframes and present different ideas to establish visual design of activities and decide on specific functionalities, however, we never properly discussed what would be relevant design patterns for our planned implementations, or how classes, activities and modules would communicate and rely on each other.

Following these early mistakes of not thoroughly considering our design patterns and models, we attempted to remedy our structures towards the end of the project with moderate success. This proved to be more challenging than anticipated as the problem had in hindsight started very early on, and making changes to our modelling and structures late in the project showed to be unrealistic because of the amount of work that would have had to be re-completed to forward certain changes.

Coupling and Cohesion

High readability and maintainability are some general goals of all programs, and in object-oriented programming, there are two important principles to measure how well a program is structured and organised: coupling and cohesion.

Coupling is the degree of the dependency of a software module to other modules, in other words, a module with high coupling is dependent on many other modules and vice versa. High coupling can result in difficulties concerning changes and modifications, since changing one thing might lead to a domino effect of more changes. Low coupling is therefore preferred.

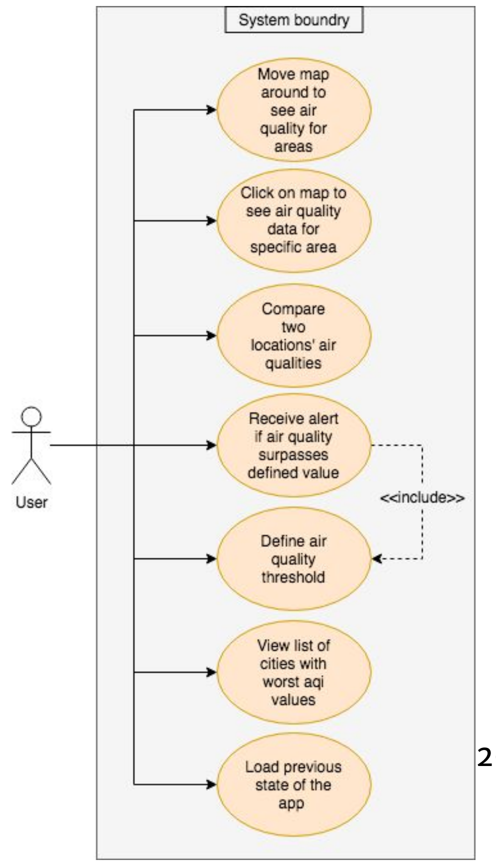
Cohesion, on the other hand, is a measure of what responsibility an object has and how focused that responsibility is, in other words: the degree of how much the different elements in an object belong together. An object with high cohesion has a moderate responsibility and only performs a few tasks within a functional area. The opposite being an object with responsibility for many tasks within different functional areas, which would lead to undesirable traits like difficult code to understand, low maintainability, testability and reusability. Therefore, an object should have high cohesion.

In the beginning of the project, one of our activity classes, `MapsActivity`, contained a lot of variables and functions that did not have a strong connection to the class like mentioned earlier. However, after an overhaul of the code, many functions and variables were moved to their own classes, giving most classes a smaller responsibility with fewer tasks within their own functional areas. With our latest iteration, most of our classes only perform a few tasks within its given functional area, and most of the elements inside the classes logically belong together. This has made the code more understandable and robust by giving the different objects higher cohesion. Although most classes have high cohesion, some do not, like the aforementioned activity. If we had more time, we could have moved functions like `getSharedPreferenceValue` and keyboard functions into new respective and appropriate classes.

As mentioned in Modelling strategy and design pattern, most classes have a degree of low coupling. Our classes (excluding imported libraries and their dependencies) are usually only dependent on the main class MapsActivity, which itself is not dependent on any other classes to run. This makes them easier to change since they only have one dependency. In conclusion, because most of the other classes only have one dependency, they can be described as having low coupling.

Threats Modeling

We never went through the process of threat modelling, as we deemed it unnecessary; threats were of no concern in our app. Considering the fact that we will not publish our app for commercial use, threats are of no paramount concern. In addition, in a scenario where the app would be breached, no valuable information could be stolen. If we were to focus more on threats, we would have used the OWASP[1] as a resource for guidelines. OWASP would help us


System boundry
Move map around to see air quality for areas
Click on map to see air quality data for specific area
Compare two locations' air qualities
Receive alert if air quality surpasses defined value
<<include>>
Define air quality threshold
View list of cities with worst aqi values
Load previous state of the app
User

[1] https://www.owasp.org/index.php/Main_Page

in discovering threats and contribute to reducing, or maybe even eliminating them.

Functional Requirements and Modelling

Universal design was also taken into consideration when developing the app, thus we tried to some WCAG guidelines. Create content that can be presented in different ways (for example simpler layout) without losing information or structure.[2] We achieved this by giving an alternative representation of the main forecast window: the graph representation.

Provide users enough time to read and use content.[3] Toast messages last long enough to assume everyone will manage to read them.

Do not design content in a way that is known to cause seizures.[4] No content in the app is designed in such a way that it should harm anyone with epilepsy.

These are our main functional requirements which we valued as highly significant for our app project:

Interactive Map

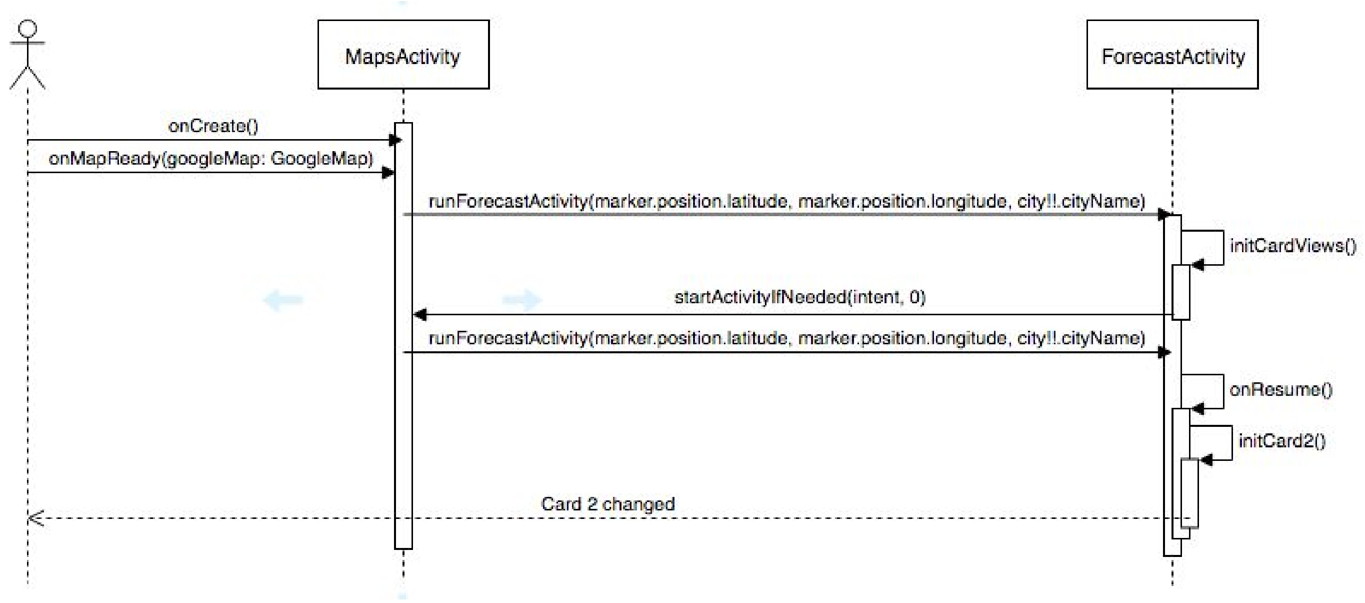
An interactive representation of Norway as a map. The user should be able to interact with the map to use other functionalities such as presenting air quality data for the location on the map they choose to press, or immediately move the map around to see simple and undetailed presentations of air quality values for different areas. We decided to use Google's Google Map API to implement our interactive map. Therefore, our modelling was heavily relying on their map API.

Location Comparison

We wanted to let the user choose and compare a location's values with another location's values. The two locations' values should be displayed in a way that lets the user easily differentiate

[2] https://www.w3.org/TR/WCAG20/#content-structure-separation
[3] https://www.w3.org/TR/WCAG20/#time-limits
[4] https://www.w3.org/TR/WCAG20/#seizure

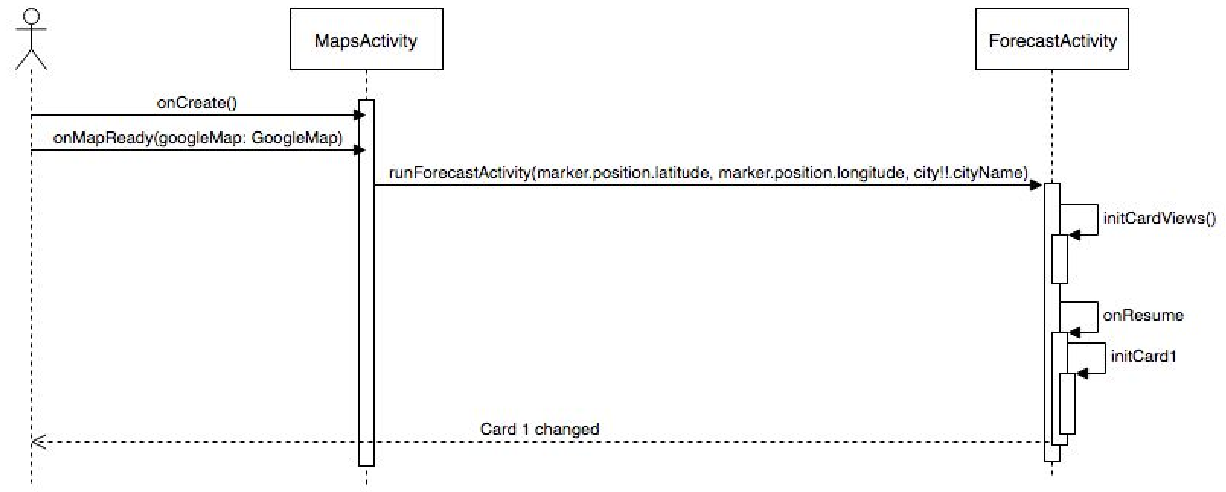
between the two and compare them side by side. This comparison applies for a given hour within that day.


MapsActivity
ForecastActivity
onCreate()
onMapReady(googleMap: GoogleMap)
runForecastActivity(marker.position.latitude, marker.position.longitude, city!!.cityName)
initCardViews()
startActivityIfNeeded(intent, 0)
runForecastActivity(marker.position.latitude, marker.position.longitude, city!!.cityName)
onResume()
initCard2()
Card 2 changed

Sequence diagram of comparing two locations' aqi values.

Displaying Forecast

One of the most important functional requirements for our project case is to display the air quality forecast data to the user. The user can, through the interactive map or otherwise, select an area or city, and the app will present the user with a display showing relevant values and information about this area.

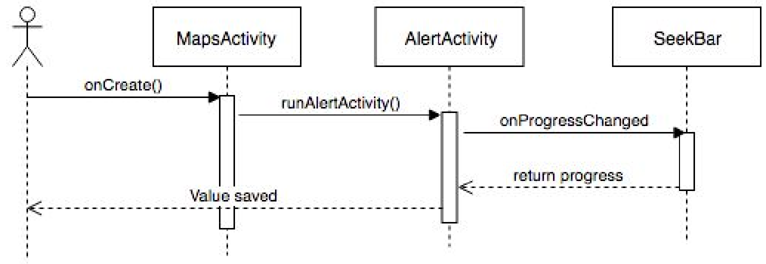

MapsActivity
ForecastActivity
onCreate()
onMapReady(googleMap: GoogleMap)
runForecastActivity(marker.position.latitude, marker.position.longitude, city!!.cityName)
initCardViews()
onResume
initCard1
Card 1 changed

Sequence diagram of displaying forecast

Alert System

An alert system is a core function of our app. We think it is important to notify users based on the air quality of their location. This heavily relies on location services, and alerts could potentially become an annoyance if not properly regulated. Therefore, our app should adapt to the user's needs, as expressed with alerts, language support and the themes system. This is why we developed an advanced configuration system, so that the user can tweak the alerts according to their preference. Ideas for functionality regarding the configuration system include: ability to temporarily disable alerts, exclude certain locations from allowing alerts and adjust specific time frames of the day. For instance, the user can disable alerts during times air quality is insignificant, or when alerts are inappropriate.

To summarise, the point of the alert system is to notify when the user's air quality is below a predefined threshold value.



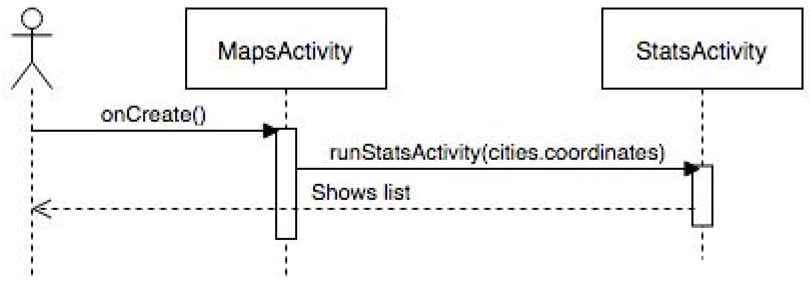
Sequence diagram of setting threshold value and saving its state.

Statistics

We wanted to provide something for the greener planet activists, so what better than a list of the worst polluters? In other words, we wanted the user to see a list of major cities sorted descendingly by their pollution values of the user's unit choice. This is also a handy way to know what cities to avoid for sensitive users.

Favourites

To broaden the application's customisation capabilities, we wanted to implement a favourites system. This system would allow users to favourite cities by clicking a favourite icon when viewing information about the city in question. Information about cities marked as favourites would be easier and quicker to access through a favourite activity.



Sequence diagram of displaying statistics.

Storage

After deciding on certain functionalities we wanted to implement, such as choosing the theme of the application, adding cities to a list of favourites, and more, we realised it would be necessary for the app to have the functionality to store the state of the app,or at least certain parts of it, between each time it is changed by the user. We thought it was necessary to somehow store data components such as certain settings so the app would not reset for each time it would be run.

Non-Functional Requirements

We primarily focused on usability and performance requirements for this part. Therefore, a lot of the "how" something was implemented was the main issue here.

Google Maps API

After some brief research on how we could best implement our idea of an interactive map, we decided that our best and obvious option would be to use the Google Maps API. Google provides an excellent API where developers can implement their Google Maps in an activity and app. The API also provides several other useful functions, such as: adding markers to the map object, styling the map with self-developed themes, and several other different uses related to latitude and longitude values. These functionalities proved to be useful for us because we could use coordinates to distinguish where we wanted air quality values from Norway when we fetched data from the MET API.

Graph Presentation

To give our users more alternatives to exactly how they would be presented with air quality data, we wanted both a numerical value activity displaying data, but also a graph activity displaying the same data. Users would freely decide with a simple controller whether they choose to have the data presented with colour codes and numerical representation, or a graph representation. The graph would contain linear graphs of the main statistics: AQI, PM25, PM10, NO2 and O3. It would also display where we currently are on the graph and the address of the location we are looking at. The user may also zoom into any points on the graph. This works hand in hand with universal design, as the interface may be too small for some users.

Language Settings

To make the design of our app more universal, we decided to implement an option to change the language of the app. We wanted the app to support several different mainstream languages, with English being the default language of the app. The language settings can be found in the settings activity.

Clickable Map and Other Features

Because we wanted our map implementation to present the user with a map object that would be interactive, we made the map object clickable. We also wanted to have rough representations of air quality values in big cities in Norway using colour coding. Markers should be placed on the 17 largest cities in Norway, and the map should be clickable, providing detailed air quality data through the forecast activity for each of the cities as well as anywhere else on the map the user chooses to click.

Filter Data Presentation

Though we default to displaying detailed data using several different terms and values, we wanted to give the user an option to filter which data they would be presented with. The purpose would be filtering out data units, to let the user precisely define which units they are interested in having displayed.

Product Documentation

Our strategy has always been to have our activity classes as our main classes. As we have progressed throughout the project, we have usually implemented most functionality inside our activity classes. We have, however, attempted to migrate most functionality and data that does not have a direct link or necessity from a specific activity out to other (data) classes, in order to remedy our structure as much as possible, and create a better structure suited for future implementations and maintenance.

In short, our goal for this project was to create a phone application with the following functional requirements:

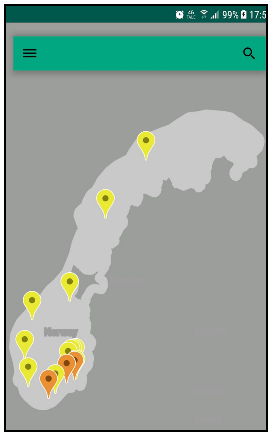
- An interactive map implementation.
- A forecast activity presenting detailed data to the user about one or two (compared) cities.
- A graph activity presenting a city's air quality data in a graph.
- A settings activity allowing the user to alter different aspects of the application.
- A stats activity showing statistics of air quality levels between cities.
- An alert system alerting users of rising air pollution or other specified events.
- A favourites activity allowing users to save cities as favourites for ease of access to information about these cities.

Interactive Map Implementation

We decided to use Google's Google Maps API to implement our interactive map. To make the map interactive, we made the map object clickable with an OnClickListener. By following the Google Maps default implementation in android studio, the map has the default functions: click-and-drag and zoom, all by touching the screen and interacting with the view. The map view also presents users with estimated air quality levels for each city. This is implemented by adding markers over each large city in Norway. The markers are created and added to the map object on launch, and for each marker, the app fetches data from met.api.no, and then sets the markers' colour appropriately depending on the value extracted. The functionality to create markers, add them to the map object, and colour them with a desired colour is all available in the Google Maps API library.

Frontend for this module presents the user with the map object upon launch. The coloured markers will already have been added to the map object.

Backend for this module starts on the launch of the application. The Google Map object is initialised, and predefined `LatLng` values representing, what we have defined to be, each large city in Norway are being used as arguments to make a call to the met.api.no, using the `Client` class. The api request returns air quality data for the area represented by the coordinates (`LatLng.latidude` and `LatLng.longitude`), and is used to make an assessment of what colour code is appropriate for the marker on this location. The marker is then created and set to the representative colour before being added to the Google Map object, using the same coordinates as arguments to set the marker in the correct area. For each marker created, a City data class object is created storing the marker, the coordinates and the name of the city. The coordinates and the city names are stored as `LatLng` and as a `String` in the `Cities` class, in a `HashMap<LatLng, String>()` object.



The `MapsActivity` class is one of the creators of the `ActivityBooter` class. It creates the class and uses it to launch other activities, such as: the `ForecastActivity`, the `GraphActivity` and the `StatsActivity`. The `ActivityBooter` class is therefore the creator of the `ForecastActivity`, the `GraphActivity` and the `StatsActivity` respectively.

Pollution Forecast

Our forecast activity is a view designed simply to present the user with air quality data, and have the option to compare two different sets of air quality data. The forecast activity is dependent on the `MapsActivity` activity, as this is where it will be launched from. However, the activity itself is always launched from an object of the class `ActivityBooter`, which only needs one parameter(`context`). An `ActivityBooter` object is initialised in all activities that have the capability to launch another activity. Each function in the object launches a respective activity, and sends the function's parameters to the new activity as it launches it. The function `runForecastActivity(lat: Double, lon: Double, title: String)` takes a latitude and longitude value as `Double`, and a title representing the area of the coordinates as a `String`.

As the forecast activity launches, it uses the coordinates to make a request to met.api.no.
Each unit's array has 48 index slots - representing each of the 48 hours, which are filled by the response of the api request response. With that said, the different pollution values are returned and stored in their respective arrays. For example, the third hour for a certain unit is found in the array at index three.

The activity's title represents the city name by a String obtained by one of the given arguments.
At the bottom, a scrollbar object is initiated consisting of 48 progresses representing the 48 hours in a day, and will be used to select the index of every array as previously explained.
Next in order, `TextViews` and the rectangle `Views` are initialised to display numerical values (found in the mentioned arrays) and background colours (according to their air quality levels), respectively.
Each `(Text)View` is updated as the time scrollbar is manipulated by the user.
E.g. `AQIValues[time-scrollbar.progress]` will find the AQI value for the specific hour set by the scrollbar.

Compare Function

The second card that represents the forecast of another chosen location's pollution follows the same pattern as mentioned before.

Pollution Statistics

Initialisation Phase

A simple layout would suffice for this task, which is why we used a plain table,using `LinearLayout`, to contain the names of cities and their counterpart pollution values. Therefore, the most relevant part of this module happens behind the scenes. Starting off in `onResume()`, the cities' names and coordinates are, respectively, received in the form of `HashMap<String, LatLng>` using the `Intent` by `MapsActivity`.

```
intent.extras["hashMap"] as HashMap<String, LatLng>
```

These are put into a `HashMap<Double, String>` called "cities". The order of these non-primitive data types simplified the process of sorting the values, which we discuss in Sorting. Furthermore, the `Spinner` is loaded with its adapter to create the menu containing the different pollution units to choose from.

Handling the API Request

The function `getCitiesStats(selectedUnit : String)` is the prelude and iterates through the `HashMap<Double, String>`, sending the city name, city coordinates and the chosen pollution unit to `getCityValue(key : String, pos : LatLng, pollutionUnit : String)`.
Luckily, *doAsync* enables a more concise way of fetching the API request using the given arguments. The result is sent as a parameter in `addCountryValue(city: String, aqiValue: Double)` by using `uiThread` - thanks to coroutines. The aforementioned function will add the received data into `perCityAQI : HashMap<Double, String>` and increment a counter until it reaches the number of countries defined in `nCities`. When this number is reached, it will call `addCountryValue(city : String, aqiValue : Double)`, which is responsible for sorting and displaying the data.

Sorting

We ended up using a rather unorthodox method of sorting the data. It began with the fact that it was more tedious to sort the values if the `HashMap` was on this form: `HashMap<String, Double>`, because you would have to access every key to then compare its value with another altered key's value and so on. Therefore, we tried changing the order. But because we were unsuccessful in using the standard methods for sorting by keys, we ended up making a new list of all retrieved keys, sorting the list, iterate the list and match it up with the `HashMap`'s value.

To display the data, we iterate through the prior mentioned list and for every `LinearLayout` we insert the data using the list's value as the `HashMap`'s key. At the end, the `HashMap` is cleared.

Alert System Implementation

The alert system consists of two activities. The first activity - the primary of the two, has a SeekBar in the focus. Here, the user can adjust the threshold, and a reference for the AQI will be indicated in form of toasts.[5] Furthermore, there is a Switch which introduces a popup. The popup comes from a separate XML file in layouts, and it is cropped to wrap its content (so it does not cover the entire screen). The popup lets the user select an item from a WheelView. A call is then sent to the local function timer(Long, Long), with the parameters depending on the item selected. The function uses the object CountDownTimer[6], which in turn uses threads to perform checks every once in a while, depending on specified interval. If the specified time has run out, it toggles a boolean to false. As long as that boolean is true, alerts cannot be sent. The third functionality of the activity is to launch the second alert activity, which is done through two TextViews.

The second activity has the purpose of adjusting the time frames. There are seven Switches and RangeSeekBars[7], representing each day of the week. If a switch for a day is turned off, the time frame for that day will not be taken into account. The RangeSeekBars work mostly like regular SeekBars, except they have two thumbs instead of the standard single thumb. The thought here is that the user can set a minimum and maximum time for a day in a 24 hour scope. If the Switch for that day is turned on, alerts will not be sent between the minimum and maximum set times - the time frame. The implementation for this activity is very smooth and simple: since there are sets of seven for each view, and they share the same functionality, all the views were declared in an array, and all the coding was conducted in a for each loop.

[5] https://airnow.gov/index.cfm?action=aqibasics.aqi.

[6] android.os.CountDownTimer.

[7] com.yahoo.mobile.client.android.util.rangeseekbar.RangeSeekBar.

Finally, the alerts are sent using the function alertConditions() from MapsActivity. The function is called after the user has allowed location services. It contains a doAsync[8], which uses threads to constantly run its code. The code is a tangled mess of if-statements to check if all the conditions are met in order to send the alert. If the conditions are met, a call to the function dangerAlert(Context, String, Double, Int) from the class Alert is made. The function uses the former two arguments to create the alert, and the latter two to customise it - these variables are used to format the string resource. After the alert is sent, a call to the function timer(Long, Long), local to MapsActivity, is made. This timer disables alerts for an hour. This acts as a cooldown, so the user will not be bombarded with alerts.

Graph Implementation

For the graph implementation, we decided to use the MPAndroidChart[9] library by PhilJaychart for the different aqi values. With this library, you can create line charts, bar charts, pie charts and many more, but we only use line chart here. The graph implementation consists of two parts: the kotlin file GraphActivity and the xml file activity_graph.

We start by getting the latitude, longitude and the title of the city from the intent that we receive from ForecastActivity. Then we convert the title of the city to an address object using a geocoder, which will be used to create a the description for the graph in order to indicate what city we are looking at. Furthermore, we make a call to api.met.no to get our api data by making a client object and calling the `getWeather()` function with the latitude and longitude from the intent as parameters.

Now that we have the api data, we begin setting different settings for the graph object, like making it draggable, enabling pinch zoom, setting the description etc. Then we have to draw the lines in the graph, and to fill them, we have to make arrayLists of the aqi data and fill them with entries. An entry is simply just a point on the graph defined by an X- and a Y-value. From the api, you will get 48 hours of data into the future with some data every hour. Therefore, we have a for-loop iterating through time array in the aqi data object and getting the different aqi values for every hour. For every hour we also create a new `Entry(x.toFloat(), aqi.toFloat())`, where

[8] org.jetbrains.anko.doAsync.

[9] https://github.com/PhilJay/MPAndroidChart

the first parameter is the time (X-axis) and the second parameter is the aqi-value (Y-Axis). Then we added the entry to its respective arrayList.

When all the points are in the arrayList, we create a new `LineDataSet(aqiData, “AQI”)` which is the line itself with the first parameter being the list of entries and the second parameter being the name of the legend at the bottom. On this object, we can set different settings for the line like the colour, the width, whether we want the points to be invisible or not etc. We then put all these lines into an arrayList and set the graphs’ data to be this arrayList of lines. Now we get the legend of the graph by calling `graph.legend`, and then we can set different settings on the legend like its text size, colour of the legend form etc.

To finish, we made custom X-axis values representing the clock by hard coding in the 48 hours starting from 01:00. We had to hard code it in because if we just used the from variable in the times in the aqi data, the X-axis became too cluttered and was unreadable. To set new X-axis values, we create a new ValueFormatter and override the getAxisLabel() function to return the new X-Axis values. Now we get the local time in Norway from Calendar.getInstance() and mark this time on the graph using a LimitLine(), which is just a straight line going down from the X-axis.

Settings Implementation

The settings activity is accessed through a button, located in the map’s menu. Settings has four features: toggle themes, toggle alerts, switch languages and display current app version. The first three of which are dependent on a call to `getSharedPreferenceValue(String): Boolean`. Upon opening the activity, we want the views’ states to remain untouched since the last time changes were made. This serves the purpose of consistency. If all changes revert upon closing an activity, the user may feel as if the app is not working properly. The function’s counterpart is `writeToPreference(String, Boolean)`, which sets the variable.

Toggle Theme

We used a Switch to represent the toggle button, called `themeBtn`, which would serve the purpose of choosing between a dark and light theme. The button controls a boolean, which in turn determines the theme at a given point. When toggled, all modules would implement the theme set in the `SharedPreference` object `settings` which is changed using the switch in question.

This alteration of `settings` is achieved using `writeToPreferences(prefkey: String, prefValue: Boolean)`, which receives the `String` “theme” and `true/false` respectively. When this is set, the `onSwitchChangedStateListener` will call `recreate()` to restart the `Activity`. Like all other `Activities`, the activity will call `getSharedPreferenceValue(prefKey: String): Boolean` on start up. Depending on the `Boolean` returned, the layout will be set accordingly using either `setTheme(R.style.DarkTheme)` or `setTheme(R.style.LightTheme)`, which sets the default style layout to be used. Every Layout will refer to `attr.xml` which accesses `styles.xml`.

Alert

A Switch allows the user to turn on/off notifications. The Switch is accompanied by a boolean, which depends on the Switch value. The boolean is declared globally for it to be fetched by MapsActivity, where it is used as one of the conditions for allowing alerts to be sent. It is not much to it.

Language

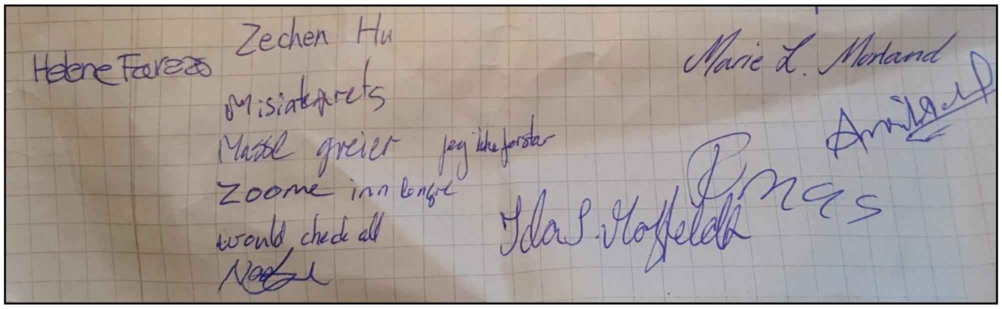
Language selection is represented by a Spinner. The spinner’s content is set through a string list, where the language names are hardcoded. We wanted the languages to be displayed in their own format, for example, it says “Norsk (bokmål)” rather than “Norwegian (bokmål)”. Therefore we deemed it unnecessary to fetch strings from resources. For the spinner’s state to remain even after closing the activity, we implemented code which shares the same functionality as `writeToPreferences(String, Boolean)` and `getSharedPreferenceValue(String): Boolean`, except it revolves around a string variable rather than a boolean. Upon selecting an object from the spinner, a call to local function `changeLocalisation(String)` is made. The parameter sent depends on item selected, and a when-statement is used to call the function accordingly. There was a small problem to the spinner however. The localisation did not update after selecting the item; only after closing the activity. To solve this, we refreshed the activity when the user touched the spinner. This introduced a new problem: the activity would refresh even when a new language was not selected. To combat this, we eventually made a simple if-check in order to see if a new language is actually selected. The function `changeLocalisation(String)` changes the default library resources are fetched from. In the directory res/values/strings, there is a String file for each of the languages.

Test Documentation

As with any project, we naturally ran a lot of different tests throughout, although from early on, we never focused heavily on setting up modules for- or structured testing. The goal with testing was to maintain functional code throughout the project. Our reasoning for not focusing more on structurised testing throughout is that we did not find it necessary. Using normal debugging methods and running the application on emulators or our android phones seemed sufficient for most of our testing purposes.

We set up a collection of tasks to gain insight from people that were not involved in our project. The tasks in order are: find air quality closest to you (1), two locations' air qualities in the future (2), toggle dark mode (3), find the city with the worst O_3 value (4) and disable alerts between 12:00-16:00 on Wednesdays (5). The averages of the results were calculated and compared with the estimation time. In addition, the collected potential comments of interviews were added. The result for any failed attempts or time-consuming tasks would be set to 120 seconds.

Task	Result (seconds)	Avg.	Est.	Comment
1	59, 37, 48, 52, 13, 35, 17, 30, 37, 80	41	120	Capped zoom was frustrating, moving the map was difficult. Initial map rotation was confusing.
2	38, 30, 83 , 30, 120, 89, 81, 25, 70, 77	64	60	Iphone users missed their back button.
3	40, 10, 10, 7, 17, 5, 25, 38s, 30, 84	27	30	Dark mode too dark, thus text was hard to read.
4	45, 45, 44, 37 40, 7, 8, 57, 15, 106	40	40	Black text may become invisible due to dark mode.
5	90, 94, 29, 120, 5, 7, 14, 9, 15, 120	50	50	Due to android version of the used device, users were not able to open this setting.



[illegible] Zechen Hu Marie L. Morland

Misiatprets

Massl greier jeg ikke forstår

Zoome inn langt

would check all

[illegible] [illegible] [illegible] [illegible]

To legitimise our tests, we have received the signature from most testers.

Unit Testing

Several units had to be isolated and tested for different reasons, the following are examples of tests performed on different units.

Bugs

Certain bugs proved harder to get rid of than others. Therefore, we sometimes used unit testing to eliminate said bugs. The approach would be to isolate the app to its core functions in a different project folder (not ideal, but it did the trick) and continue testing, adding features incrementally until the bug appeared. A bug which stalled the progression was the following line:

```
implementation 'com.google.android.gms:play-services-maps:16.1.0'
```

We asked others to help, but nobody was able to figure out why this line said it conflicted with another version,even though it appeared not to do so. After following this poor practice of testing to find out when the bug appeared, the exact line was found and dealt with. This was, however, a peculiar situation and debugging etc. would not work so effectively in this scenario. We learned to try out the debugger and use breakpoints in Android Studio for future bugs that could be resolved by checking values incrementally throughout the execution of the application.

MapsActivity (API Request)

To get the API data, we had to do a request. Because we struggled implementing this and met on several problems underway, we decided it would be best if we just made a testing module for this. That way, we could test the module and see if we could get data everytime we made changes in the experimental phase, because we did encounter many problems along the way when experimenting. Also, later on if we wanted to compare the result with the API data in real time, we could always just use this one functional testing module to check its validity, instead of running the entire app for itself.

GraphActivity Testing

To test the graph, we started by hard coding in a couple of entries into the `ArrayList` and drawing those lines just to see if the lines came out correctly. Then we put in actual data and checked if the points were correct. On an interview we performed, it was not apparent for the interviewee that you could zoom in and move the graph around to inspect the values more closely. This could be fixed by integrating the ability to double click on the graph for zooming, since this was what he originally tried. Also, it was preferable if PM10 and O3 were written like PM_{10} and O_3 instead, since it was confusing at first what they actually meant. Under the interviews, a bug also presented itself that we had not encountered yet. Sometimes, if you clicked too fast on the graph icon in ForecastActivity, the app would crash, and initially we did not understand what caused this.

Alerts Testing

Alerts in the settings worked pretty much out of the box, and testing was not necessary. On the other hand, programming the alert and week activities was much harder, and as a result required far more testing. In the alert activity, the user is first approached by a SeekBar. For the SeekBar, it was a challenge to display the value selected. We ended up using a drawable gradient to colour the seekbar, and then a configured toast to give a reference for the value; the toast will be cancelled and a new one will be sent, depending on predefined value ranges. To reach this conclusion, testing and much experimentation were commenced, until we found a design/implementation we were satisfied with. Moving on, the popup opened by a switch was surprisingly easy to implement, but still had one important issue: the app would crash if an item was selected. We quickly realised it only crashed after selecting the first item. It did not take us long to figure out it was an indexing mistake. Finally, the alert system can also be traced to MapsActivity. There, the code consists mostly of a set of if-statements. The function here did not work, and the way we tested it was to write messages to the log after almost every line. This is where the majority of testing regarding alerts took place. It took some time, but everything eventually worked - very much thanks to writing and examining to the log.

Integration Design

Every group member shared the same bachelor program, and we did not have the privilege of having a member specialised for design for instance. Accordingly, every group member originally had about the same knowledge regarding Android Studio, Kotlin and general software development planning and design. Since every member worked on their own set of assignments, everyone became specialised in their modules. As a result, these modules grew quite isolated from each other, and most code developed by one developer would be alien to the other. Eventually, all branches were merged into one. During this merge, everyone naturally learned much about the other branches for the purpose of resolving conflicts.

Genius Design

From the beginning, we ended up doing a lot of choices based on our opinions and experiences. Therefore, little or no testing was used to guide us to the decisions we ended up with. However, everytime we were unsure, one of the team members would eagerly go and find someone to ask their opinion on the matter. This would lead to quick resolves in any insecurities the team met along the way. Still, too little testing was done, and in aftermath, we wish we had done more. This was also due to a lack of experience, because nobody was sure how and for what to perform testing. There is also the factor of confidence that played a huge role in this subject, because asking strangers was not really discussed, while asking family members was discussed but not acted upon.

A lot was discussed in the groups, and although many members would be apathetic to the choices made, a consensus was reached in the end of almost every discussion. Because of our approach, we decided to have some integration tests at the end to see how well the user was able to use the app. The testing was basically assignments to find given functions or do certain things within an amount of time. The lower the better of course. Results revealed a lot of problems with the app, and if we only had done this earlier on, we could have made it more user-friendly. Now we know and will remember this for future work. An upside with genius design, though, is that features be added quickly to the app because the team does not have to consult with anyone outside the team. The disadvantage, is as mentioned previously, that sometimes you might end up having to rebuild certain things from the ground up because of crucial information revealed through the users after testing.

In general we have tested the app (integration testing) only for the things we wanted to happen. Rarely would we check for things that we do not want to happen. For this, unit testing and integration testing is crucial. Because we have not done much of this, there might be a lot of undiscovered bugs in the app, and these are of course the worst. It most certainly is not user-friendly. Which is another reason for testing more than we have done.

Integration Testings

For every iteration or newly added feature, the team would test the app to check if it was working accordingly. The results of running the app could be: crashes, which would have to be resolved before sharing the new iteration; bugs, which could be included and mentioned in the new iteration, so it could be fixed later and imperfect solutions, which would require a new approach after having a discussion.

Towards the end, we conducted several interviews to help us discover and prioritise tasks. The interviews were performed on: students, who had some technical insight; parents, who we thought would struggle the most and random people with unknown backgrounds, who had unprecedented opinions as opposed to those with more IT experience.

The last mentioned group of people had opinions different than the students, most of which had never been mentioned before. We believe this is due to different backgrounds and therefore rather different mindsets than the students at the Institute of Informatics. This diversity of mindsets was much appreciated and has lead to a few changes that were not initially considered.

As for the interviews, we wanted to see how fast the users could finish certain tasks, and if they had any comments. This way we could see if some tasks were finished faster or slower than our expectations, hopefully learning something new in the process. For example, we saw some users rather opening the Statistics to view the value closest to them, instead of using one of the markers or searching for it using the search bar. This was a slightly surprising, but definitely interesting observation.
Testing of the application as a whole was largely dependent on each developer's style. If a developer wrote a code for a branch and discovered a bug, that developer would fix it himself. If

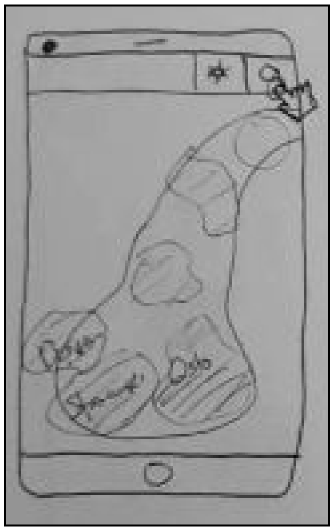
he was not able to resolve it himself, he would hopefully reach out to the other developers, usually via Trello. But due to lack of communication, unfortunately this would not always be the case. If a developer found a bug in another branch, he would ideally note it on Trello, and the developer responsible for that branch would look into it. Finally, many bugs were also uncovered from interviews conducted later. To summarise, integration testing proved to be quite useful. One could speculate that it is easier to discover bugs from code developed by another person, perhaps because of unfamiliarity to the system. In contrast, would the app be developed by a single developer rather than a group, chances are that a lot of obvious and some less obvious bugs would remain.

Conducting the interviews can be time-consuming and might not always rewarding. However, because one of the members was much accustomed to interviewing, this went smoothly and rather efficiently. With the right tools, we believe interviews to be beneficial and worth it no matter the situation of your genius design.

Process Documentation

Main Features

Map



The Google Map activity has been one of the most central modules from the beginning of the project. It has served as a sort of main menu as well as one of the most central activities, linking most of the other modules together. After creating our first activity - the maps activity, most other modules were designed with the assumption they would be launched and/or used by the maps activity. This resulted in the maps activity becoming a “god class”, containing nearly all functionalities of the app. This continued throughout most of the project,until the end, when we did a complete overhaul over several functionalities and moved them into a better class structure to relieve the maps activity of its large responsibility.

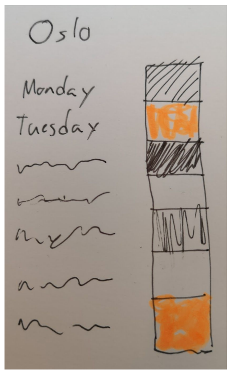
The maps activity also went through several large re-designs throughout. Our initial plan for the activity was to have a clickable Google Map object with coloured overlays over each large city in Norway. The coloured overlays would represent the air quality levels above the area it would cover, and be set to an appropriate colour depending on the values. On the right you can see an early wireframe of Google Maps activity.
Clicking a coloured area was intended to send the user into the forecast activity, with information about the clicked area as arguments.

Throughout the project we struggled implementing colour overlays. After having issues figuring out how to do the implementation for several weeks, we decided to overhaul our design. We moved on to making a new design for the maps activity, using technology we now had more experience with, to implement similar solutions to the activity in a timely fashion.

Our new design replaced the coloured overlays with markers. One marker would be coloured appropriately (using data fetched from met.api.no using the city's coordinates) and placed over the corresponding city using the city's coordinates. The map itself was decided to still be clickable, where the click would use coordinates from the map to make an API request to fetch air pollution data, as well as using a geocoder to get the name of the area. A bug was however later discovered with this functionality. If a user was to press outside of solid land (in an ocean or otherwise) on the map, the application would crash. Due to time limitations and other pressing issues, this functionality was then discarded.

To see the final result of maps activity, see MapsActivity.

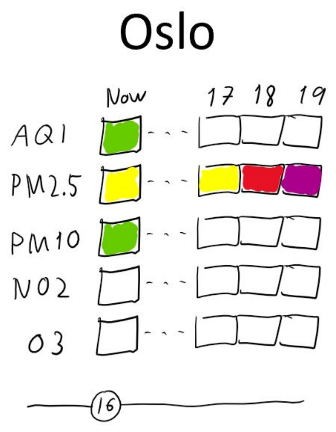
Forecast
To the right is a very early sketch of what visualisation of the API data might look like.



An early version of the Forecast view was one of the first things we implemented after we had managed to retrieve and organise the data from the API, and had a basic map functionality working.

We tried using things like include and recycler view to simplify the code, but we were unsuccessful due to our inexperience with the platform and the growing complexity of what we were trying to do. Because of this, we ended up duplicating a fair amount of identic code. This was not an ideal solution, as it made changing the functionality and appearance quite a bit more tedious and impractical. Efforts were made in the group to “clean up” the code, which made things a bit simpler, but still not as elegant as it could have been, given more time and attention. Due to this, we also ended up with not easily being able to make some design changes we wanted to because of the impracticality of changing the code.

At one point, we had the idea of having a multiple rectangles horizontally to display the air quality values for the next few hours. This idea was largely scrapped due to the added complexity of implementation because of the issues described earlier. Different designs were considered for the bottom scroll bar, including one where the time would show on the scrollbar itself, but in the end we went for a more basic design.



Later sketch with data for the next four hours. The time (hour) is shown on the scrollbar itself.

We decided to add compare functionality, which allowed the user to select a second location and see information for both in one screen. The button in the bottom right corner of each card was added as a way to easily open the graph view of either of the two locations. We also decided to include the difference of the individual values between the two different locations as a way for the user to easily see which of the them had the best air quality at the given time.

Graph

In the planning stages, we all agreed that an alternative representation of ForecastActivity would be nice. Therefore, we came up with the idea that a graph of the same data could be a good alternative. Originally, we only we only made one line in the graph describing aqi, but we ended up adding the rest of the values as well because we also used the rest in ForecastActivity. In addition, we experimented with different settings for the graph, like text size, text color, straight lines with points on the graph or a smooth graph with no points, and other minor edits. In the end, we felt like a smooth line was preferable to the dashed line, since it became cluttered with all the points.

The last couple of features we added were customised X-axis values and a red line marking the current time. Having actual times on the X-axis felt easier to understand than 1h, 2h, 3h and so on. Marking the current time helps to read the graph.

Statistics

A member was enthusiastic about adding some sort of statistics regarding the pollution in the different cities. However, this was not prioritised in the beginning.
When discussing something to motivate people in the direction of a greener planet, the statistics idea was brought up again. The thought was that it would lead to a natural competitiveness. Therefore, it was quickly implemented by the same member who wanted it. This natural interest for the module helped implementing this module in no time. However, the idea for some time was to have the statistics show only one pollution unit for a chosen amount of countries, which was quite dissatisfying. As an improvement, we allowed the user to choose which unit to view, and based on this generate the statistics.

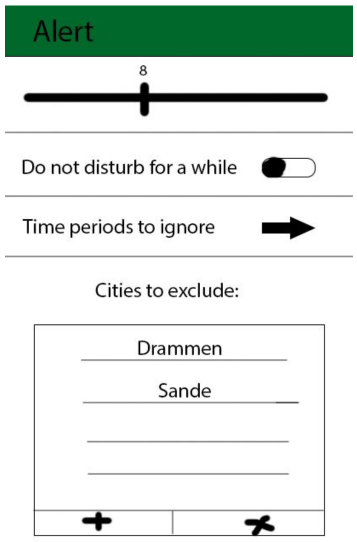
Settings

To begin with, we were on the fence regarding the settings: should the settings be directly on the map, should it be opened by some interactable object, or maybe settings as a whole is unnecessary. We were divided on this case, but ultimately we agreed to have a separate activity dedicated to settings. The reason was: we had planned many configurations, and we thought that having these configuration options on the map would the map with clutter. Since the app is meant for a smartphone, it was important to only have necessities on the map in order to maximise usability.

Configurations were easily added to a vertical layout. At first we wanted to have the threshold bar in the settings, but eventually emigrated it to its own activity.

Alert System

The alert system was always regarded as one of our big features. To start with, the idea was quite basic, but later expanded. Initially, we only wanted the ability to send a notification to the user should the current AQI value exceed a predefined threshold and of course the freedom of toggling the functionality, preferably via settings. After that had been properly implemented, we soon realised that we were not satisfied - we wanted more functionality revolving it. To be fully content, we decided it was necessary to allow more room for configuration around limiting alerts, specifically by including an "exclude location" system along with some other vague plans. That way, we would give the user freedom around alerts, so it would not become so much of a nuisance. We ended up implemented the features just fine, except for the "exclude location" concept: we concluded that resources would be better spent elsewhere at this point. In other words: we sacrificed it in order to accommodate other features. The reason we scrapped that concept rather than the others is simply because it was the most difficult of them and it would demand the most attention. To conclude, the alert system was initially planned to be quite basic, but its features were expanded over time, but at the same time perhaps not as much as we wished. Above is an early wireframe before the implementation on the activity was commenced. As seen on right: the planned exclude city box.



Other Features

Favourites

When brainstorming ideas for the application, we came up with the idea to be able to favorite a specific city. When you would click on the marker of a town and ForecastActivity started, a star or an appropriate button would appear somewhere on the card of the town. Clicking the favorite button would add this city to a favorites activity, where you could easily view all cities you had previously added and their aqi values. The idea was to have some sort of scrollable list of card views containing the information about the place. However, this was not a priority and unfortunately there was no time to implement it.

Filter

The filter concept was one of our early ideas. Imagine if there is no concern for good air quality: a user thinks it is irrelevant to see map pins where the air quality value is low (good). The filter would allow that user to hide such pins and only display relevant ones. It would contribute to reducing clutter on the map, which in turn would increase the app's usability. Furthermore, we thought it would be best if it was accompanied by a SeekBar, and the user would be able to see the effects live. For that, however, it would be necessary to place the bar on the map - which increases clutter. As other more important modules occupied the resources needed to implement the filter, we were forced to disregard it as a whole.

Dark Mode

It was decided early on that dark mode was to be one of the prioritised implementations in our application. The process for implementing the dark mode was quite straight forward. It did conflict, however, with target Norway. Since the target Norway feature was eventually discarded, the conflict resolved itself. The themes were slightly tweaked with time after our liking, and we ended up with a quite different look other than the original.

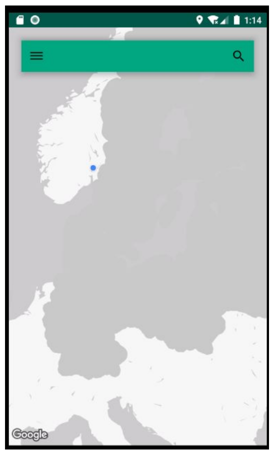
Target Norway

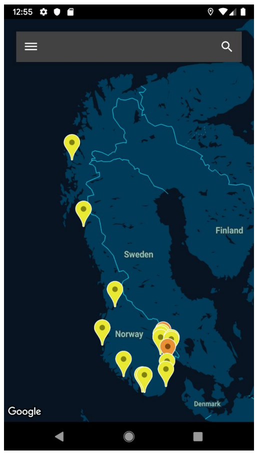
One of our original ideas was to restrict camera movement to mainly focus on Norway. As we experimented with the idea however, we quickly deemed it necessary to not only restrict movement, but also exclude everything but Norway. The thought was that Norway is the scope of the project and everything else is irrelevant, thus we wanted to make sure this was presented clearly to the user. It turned out that it was impossible to have Google Maps only include one country. To compensate, we developed the idea of removing the area around Norway visually, or as we came to call it: camouflage. We experimented with GeoJson to create polygons on the map, and then colour the specific polygons with the colour of the ocean. The first hinder we came across was that the colour of the ocean depends on the theme in use (as seen below). Thankfully, it was not very difficult to fully integrate the camouflage concept with the dark mode concept. Camouflage still had a major flaw though: it was very heavy. The GeoJson file consisted of thousands of coordinates; the file took nearly six megabytes of space, but far worse: it took a lot of time to load all the polygons, which in turn delayed the load time for the map. This concept had certainly taken a toll on the launch speed and overall reliability, which was definitely not what we had in mind for our non-functional requirements. Furthermore, the polygons were often inaccurate, which left remnants of small islands and coastlines in the Baltic Sea visible. Means to cover these up were made, which further deprived our time. Another flaw of the camouflage was that it was heavily dependant on proper movement restrictions. Since polygons load so slowly, we only drew the essential area around Norway: Denmark, Germany, Sweden, Poland, the Baltic states, Finland and Western Russia. If the movement was not properly restricted, the user may get outside of our intended boundaries, and spot areas untouched by the camouflage. All these flaws piled up, which as a result drained our time.

On a meeting, it was suggested to remove this functionality. The benefits of keeping it would be: centering the attention to Norway: at fully zoomed out, it might look as if Sweden or the Baltic Sea is the focus of the app. The other benefit is that it tells the user that we do not support locations other than Norway: they are simply not in our scope. The big disadvantage is that it is poorly optimised. We concluded that it was too much of a sacrifice, and that we must prioritise a good user experience and proper functionally. The functionality was scrapped as a whole; it just became too much of a hassle.

We did not abandon the idea, however, and a compromise was reached. When it comes to restricting camera movement, we adapted a new method: we thought that it should suffice to simply rotate the map, but paired with proper zoom and movement restrictions. Unfortunately, we never managed to implement these restrictions.

Below is the camouflage concept at its latest stage. As seen on figure one, this concept was largely dependent on a proper movement restriction in order for the user not to go beyond our boundaries. Below is also a picture of the finished map, but rotated to fit better with this document:





Language Settings

Initially, language support was not even taken into account. It was decided at the first meeting that English would be the language of the app, and to extend support to more languages was not considered until much later in the project. Subsequently, our String.xml file was poorly structured and was not written with language support in mind. Many strings had duplicates and there were also many unnecessary ones. We chose English as the default language because the language would be generally used in all our tools: the coding itself, the report, in notes, and design and in other medias such as Trello. Naturally, English fits best and created a logical flow. With the ability to change the default language via settings, we believed the pick for the default language would have a minimum impact. About midway through the project, one group member brought up the idea of looking into language support. The idea was quickly rejected by some members, thinking it would be too difficult, but ultimately, the idea was backed and put into sprint 5. The language support would serve many purposes, the primary of which was to reach out to as many customers as possible; universal design is a central concern of the app after all. At first, the idea was not very fleshed out, and we only wanted to add Norwegian support and functionality for the user switch between the two. Towards the end of the project, we realised that we should add even more languages. We agreed, however, that they would be implemented after the app was finished in order to avoid tedious editing. Accordingly, when we realised we wanted to add language support, we had to reorganise all the app's strings: clean up hardcoded strings, merge duplicates into one and remove some unnecessary abundances. Concurrently, we experimented with different tools for the xml-file, such as lists. We concluded that it was unnecessary and not worth the hassle, and that we were content with plain strings with the occasional formatting. After all, we ended up with only about a hundred strings or so. To summarise, we went from supporting one language to two, and then to eight. Furthermore, the reason we constructed this module was to push universal design and reach out to more people.

Minigame

We wanted to integrate a minigame into the map. The point was to encourage people to interact with the map and explore statistics. We thought it would be simple and fun to implement, however, because of its low priority and always having several other pressing issues, it was never worked on.

Planning

We noticed the necessity of keeping track of everything that was needed for the project.
The tasks, modules or specific functions had to be organised in a way that made it easy to prioritise under any circumstance. This is where the product backlog comes in. All members had learned about it, but never utilised it. Three characteristics represented by acronyms were used to guide us: DEEP, DIVE and INVEST[10].

We wanted the items in the backlog to be explained with their estimation at a level of detail that reflected its position in the backlog. Furthermore, the ordering had to be linear with the possibility of changing the order, adding or removing items. DEEP covered all of these criterias: **D**etailed appropriately, **E**stimated appropriately, **E**mergent, **P**rioritised as needed.

The user stories would usually be the starting point for making a product backlog item. Therefore we had "INVEST" in the back of our mind when making the user stories. The user stories should be independent and not overlap each other in the sense of functionality, they should be negotiable for later sprints, they have to bring value to the table, they have to be estimable for the product backlog, sized according to a single sprint, and the user stories should also be testable or have a clear required function that can be tested.

10
https://resources.collab.net/blogs/product-backlog-is-deep-invest-wisely-and-dive-carefully

Last but not least, after using INVEST to create user stories, which are then put into the product backlog according to DEEP, we changed the backlog to fulfill the DIVE criteria:
Dependencies are prioritised higher than their dependants. **E**stimated effort as mentioned earlier. However, we did not bother focusing on the "**I**nsure against risks" and "Business **V**alue", since the app in itself had no business value or risks and the technicalities would be clumsy to test with an app that was unfinished.

The product and sprint backlogs definitely improved our situation by providing us a better overview, we follow the same format/guidelines as Scrum.org[11].

We decided early on that it would be best not to use the waterfall model from kanban. Instead we used a hybrid between kanban and scrum, scrumban, where we created our own mixture of both of them to suit our group the best. An important feature from scrum we took good use of were the sprints. Our sprint cycles lasted a week where we every sprint planned some features that needed to be implemented, and at the end of the sprint we would hold a retrospective on thursdays. On this meeting we discussed what had been implemented, any issues we encountered, and what needed to be done in the next sprint. A feature we used from kanban was the product backlog. We would have a backlog of all the features that needed to be implemented, and at the beginning of a sprint we took some elements from the backlog and began work on implementation for them.

An important part of the planning process was to not create waste in the development. Therefore we made sure that we did not have a functionalities and unnecessary complexity in our app. We did not use a lot of unnecessary time and effort on something that did not give any value to the end product, thus avoiding technical debt. Which is why we focused on implementing less features that supported better a development flow. This also went hand in hand with scaling: to not scale up the project with unnecessary functionality just for a "profit" increase. If we had scaled up, a lot of our focus would have landed on individual tasks instead of the entire project as a whole.

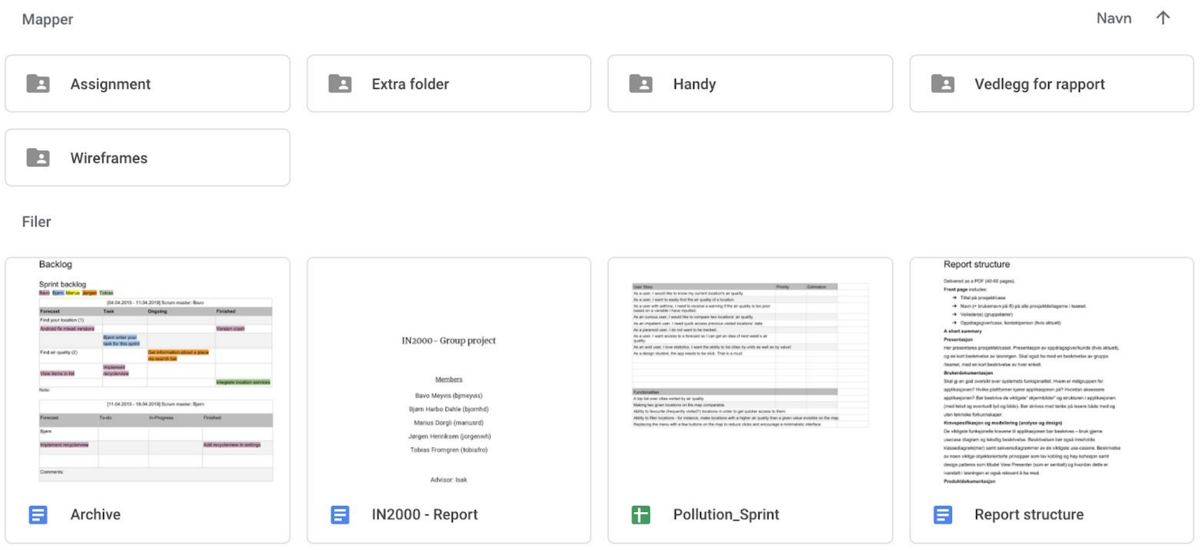
To add to the planning process, it was also important for us to use a correct and precise language. We tried to use related jargon where they fit and avoid pidgin language to reduce

[11] https://www.scrum.org/resources/what-is-a-sprint-backlog

confusion. Such confusions are counter-productive and may even result in devastating problems. To conclude, learning to adapt to proper development language is important for teamwork and will likely play an important role later on in working life with similar teamwork structures.

Google Docs

Little time was used to decide where we would keep the product backlog. Fortunately, later on, we would find the perfect candidate. For a long time, however, we would give Google Docs the privilege of storing our product backlog.



Overview over the Google Documents and folders shared in the group. Folders containing useful files and links, as well as the report document file and more.

As you can see, the product backlog is hard to maintain and modify. Moving items, or prioritising them for that matter, seems impossible. Not to mention there is no filter system to sort by estimation or priority, all of which are important for ease of navigation. Also, we had to decrease the padding of the document to make room for the user stories, which should go without saying is ridiculous. We even connected it to a Google Sheets to give us more functionality, but because the document and sheets would not update each other on any changes done to one or the other, this also became practically impossible to cope with. The same goes for the functionality log, which later would be merged with the product backlog.

Not only was Google Docs used for the Product Backlog, it was also used for the sprint backlogs. We were faced with a lot of challenges concerning the relationship between Google Docs and our sprint backlogs. First of all, the width of the document made it hard for the sprint backlogs to contain long tasks and many tasks would result in a clutter. A work-around was to minimise the margins of the document, but this was a far cry from a good solution. A creative way of colorising every member and using their color on the task they were assigned was also used. However, this was extremely tedious. Another concern was the fact that the product was supposed to follow the DEEP criteria, which it certainly did not fulfil. One of its criterias, which in this case was crucial, was for items' description to be specified at a level of detail according to its position in the backlog. You could not even get more details than the title of the item for crying out loud. In addition, the scrolling to the correct sprint was also annoying. In hindsight, this product backlog was an utter mess and simply inept. To summarise, the sprint backlogs were rarely accessed or modified. Trello was therefore introduced to the group which made everything a ton easier.

Trello

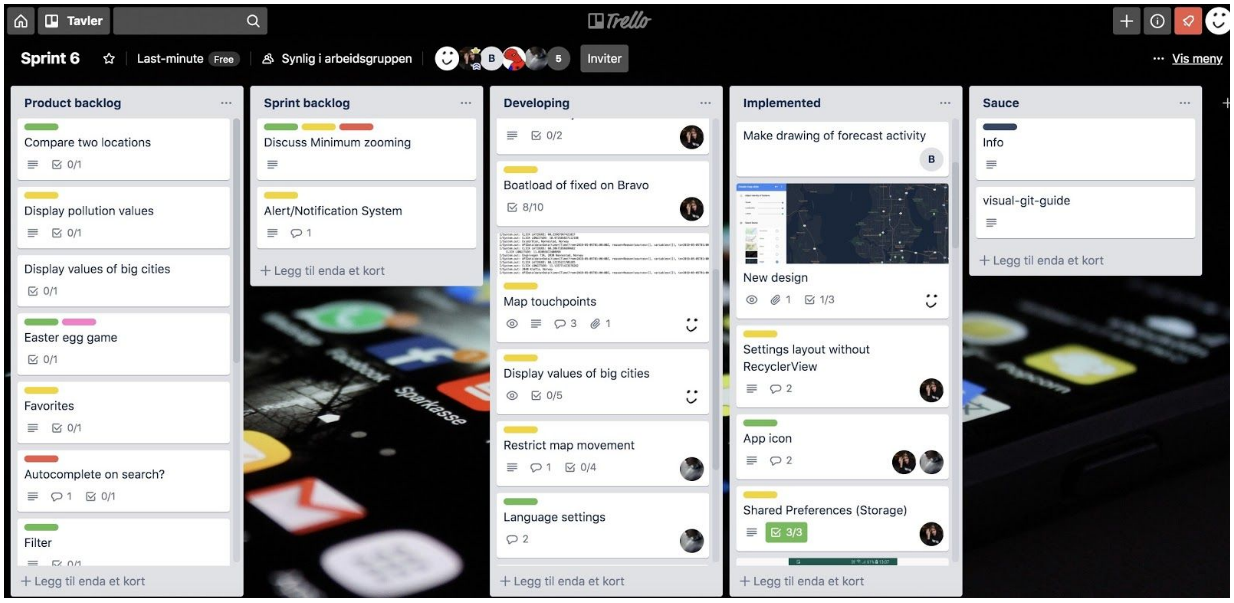
"*The single most important practice is the retrospective because it allows the team to learn about, improve, and adapt its process.*" - Project Management Institute & Agile Alliance, September 2017.

Trello is a visual table tool used to organise and keep track of notes, tasks (with checklists) or other miscellaneous. We created our own Trello table online and used these one week at a time to keep track of what needed to be worked on, and what was currently being worked on or had been completed. We could create tasks and place them in different categories that we defined to represent what needed to be done, what was currently being done, and what had already been done. Each task could in the table be assigned one- or several members. Thereby, we could easily keep track of exactly who were working on what tasks at any given point. Each member could assign or retire from any given tasks themselves.

In the early stages, before we discovered Trello, we used Google Documents to keep track of our product backlog. However, after implementing Trello, the product backlog was hastily incorporated into our Trello tables, which resulted in a much easier and better overview for us over our product backlog as well as other tasks and checklists.

As well as having a good overview over our previous, current and future tasks, we also had an option to mark each task with a priority mark. These marks would represent how immediate the completion of a task needed to be. A task could be marked as blocked if it was dependent on other tasks being completed before it could be processed. If a task was blocked by another task, the task needing to be completed to unblock the previously mentioned task would be marked with a high priority such as “urgent” or “ASAP” (meaning “as soon as possible”). Each task could also be implemented with a checklist, if the task contained several smaller subtasks to be completed in order to fully complete the main task at hand. These task were usually accompanied with descriptions and pictures, and sometimes also user stories or use cases.

All in all, Trello was highly useful for us because it worked as a great tool to keep track of - and visualise our tasks. It was easy to find tasks’ requirements and Trello provided a good overview of all the sprint’s tasks.



A screenshot showing our 6th sprint’s Trello table.

There were also several other integrations available if the Trello board was used for an agile method like SCRUM, thus you could for example see statistics of every sprint. We did not use any of the other optional features because of the struggle with the subscription system in the application itself.

Team-Work

Collective Leadership

From the beginning of our project we decided that everyone's votes should be equally as impactful. All decisions making a direct impact on the project was to be made collectively. However, to upkeep healthy progression, we decided to have someone lead the conversation and keep it relevant. This was solved by having a weekly facilitator, where all group members would rotate on being the facilitator for a week each. The facilitator's job would be to bring up the (pressing) issues of the current week, and help assign workloads to each group member for the next sprint. The facilitator's responsibility was neither to assign work to each group member, nor make decisions for the group, but to guide,create initiative and follow up.

To make sure we could effectively and fairly assign workloads and tasks, we would have "bidding sales" for each week's tasks. Anyone having a particular interest in one of the tasks would be assigned the task in question. Otherwise the facilitator would suggest tasks to each group member, and move tasks between group members (even during the sprints) if need be.

The idea of rotating as facilitators was not always successful. Certain weeks the role would be forgotten altogether, often because certain group members were less comfortable leading the conversation.

Group Goals

Our workflow primarily consisted of weekly sprints along with a new set of tasks. Often the tasks would have relations to each other, and certain tasks could be dependent on others. Therefore, we agreed that the weekly goals of the team would always have higher priority than group member's personal goals. We never experienced any difficulties with this as most group member's personal goals were the tasks they had been assigned from the product backlog.

Team-Work Issues and Solutions

Throughout the project we encountered several different issues weakening the team-work. Some of the main difficulties we encountered and solved were:

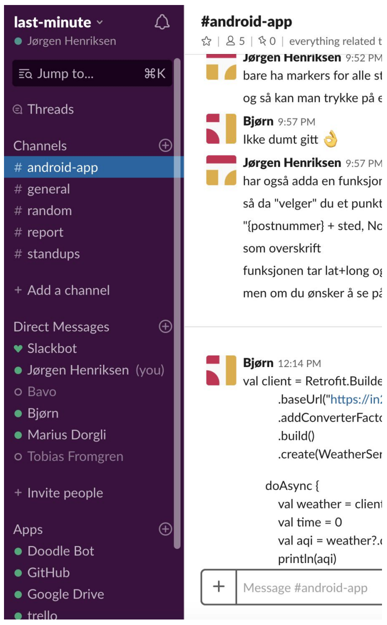
- If a group member could not make it to a sprint retrospective, we would use Trello to make sure the missing group member was kept updated with the new ongoing tasks.
- If a group member did not meet workload expectations we would push them and try getting more in touch. However, if this still did not produce better results, we would inform someone at UiO of the situation.
- If a member naturally took the lead, that person would try to stay in the background to see if this helped letting others talk. However, unwanted silences would commonly emerge; therefore, more initialising members had little choice but to take the role as facilitator.

Communication

A handful of communication applications and programs have been used throughout our project. Some applications have been very useful, while others were slower to catch on to and have not been used to its full potential within the group.

Slack

Slack[12] has been used as our main way of communicating throughout. It is an online communication device where we set up our own server, and used it to create our own text channels to effectively communicate relevant information between each other at any given time. We had different channels for different topics, such as “android-app”, used for anything related to our project app; “report”, used for anything related to our project case report; or “standups”, used to plan and set up group meetings and get togethers; and “general”, for miscellaneous.



Jørgen Henriksen 9:52 PM
bare ha markers for alle st
og så kan man trykke på e

Bjørn 9:57 PM
Ikke dumt gitt 👌

Jørgen Henriksen 9:57 PM
har også adda en funksjor
så da "velger" du et punkt
"{postnummer} + sted, Nc
som overskrift
funksjonen tar lat+long og
men om du ønsker å se på

Bjørn 12:14 PM

```
val client = Retrofit.Builde
        .baseUrl("https://in
        .addConverterFacto
        .build()
        .create(WeatherSer

    doAsync {
        val weather = client
        val time = 0
        val aqi = weather?.c
        println(aqi)
```

Our team’s slack server.

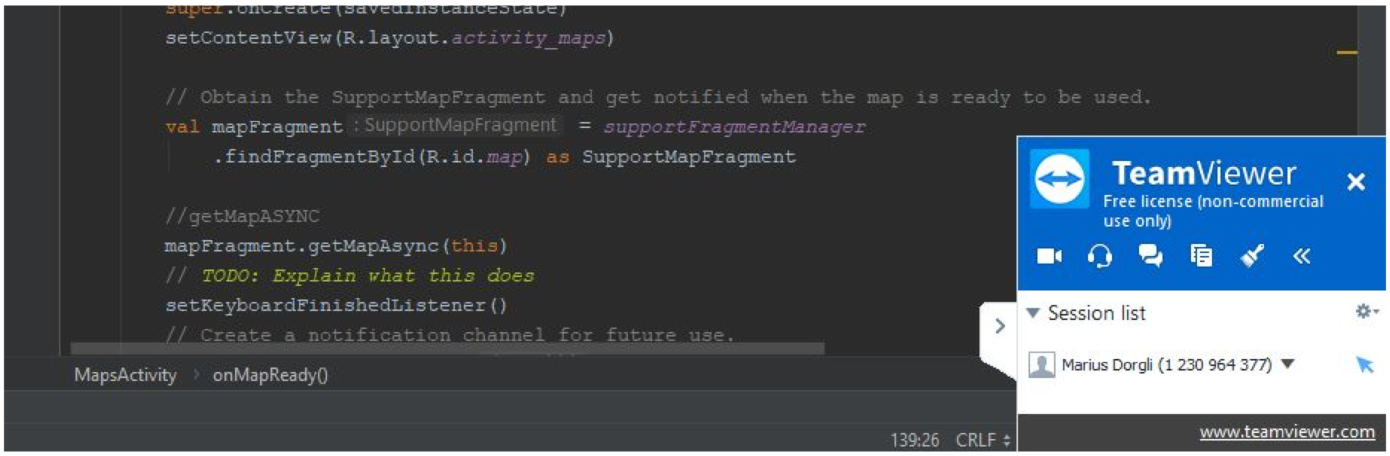
Physical Meetings

We also decided to create a discord server so we could talk with each other more often without having to physically meet up. There were a lot of advantages with meeting up personally. It is faster to discuss: you can quickly explain things using your surroundings (e.g. whiteboard), conversations feel more engaging, and after breaking the ice you can joke around and make it a pleasant experience for everyone. Some disadvantages might be the fact that people are less engaged when coding because they are not in their natural habitat or some experience difficulties expressing themselves. Meetings might be an annoyance if you have to do something else somewhere else. In other words, you are bound to stay until the meeting is over and you cannot pop in and pop out whenever it suits you. More importantly, because some of the group members lived far away or had other tasks in life, it would not be that convenient with personal meetups.

12 https://slack.com/intl/en-no/

Teamviewer

The Teamviewer[13] application allows users to connect to remote computers, provide remote support and collaborate online. Teamviewer had functionalities such as being able to control your own desktop computer from your laptop anywhere, have online meetings where we could all see and, if need be, control each others computers and screens to edit each others' code and projects, or work together one-on-one, seeing the other person's screen live, which was helpful in seeing what is going as things happened throughout. Whenever someone was in distress or had a question, we could together scroll and discuss the code on that person's computer without physically being there. This was especially useful for when we merged branches. This was great and all until we discovered we were not using their application in accordance to their "commercial use" rules.[14] This was due to the fact that the application was being used to cooperate on a project which in terms of their guidelines and rules is considered commercial use. To continue using Teamviewer, this meant purchasing a license.
Although Teamviewer was helpful, we did not want to buy the license, which is why we ceased using the application altogether.



```
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_maps)

// Obtain the SupportMapFragment and get notified when the map is ready to be used.
val mapFragment :SupportMapFragment = supportFragmentManager
    .findFragmentById(R.id.map) as SupportMapFragment

//getMapASYNC
mapFragment.getMapAsync(this)
// TODO: Explain what this does
setKeyboardFinishedListener()
// Create a notification channel for future use.
```

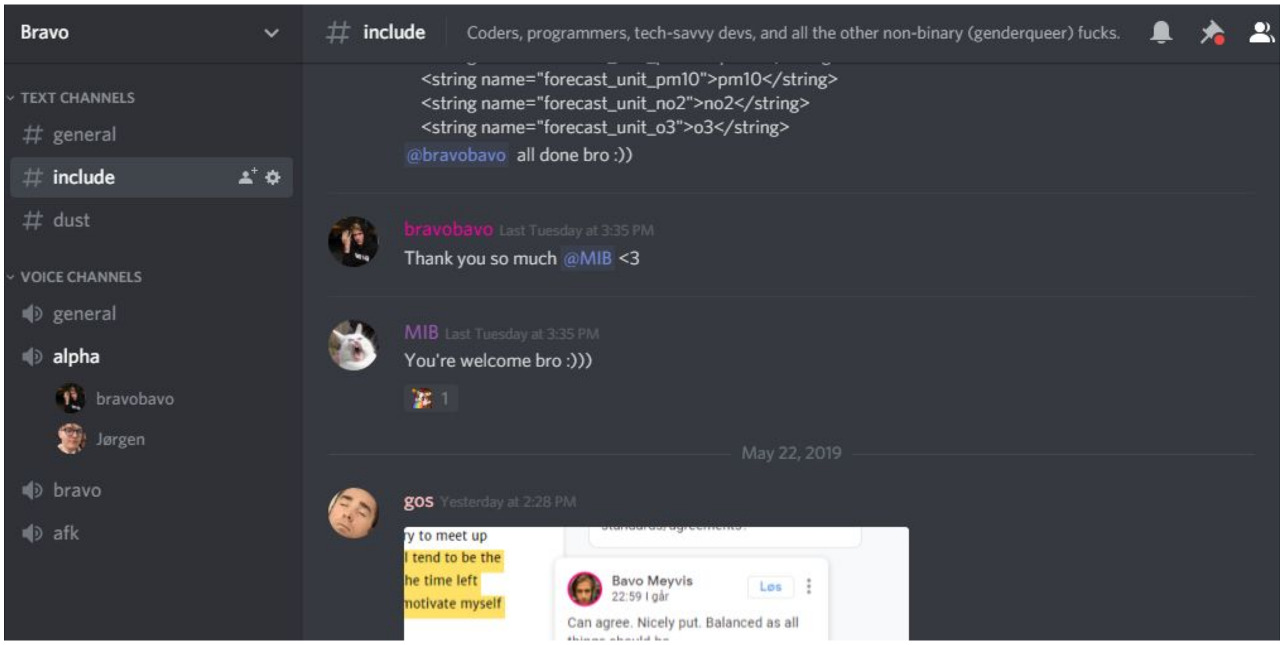
MapsActivity › onMapReady()
139:26 CRLF
TeamViewer
Free license (non-commercial use only)
Session list
Marius Dorgli (1 230 964 377)
www.teamviewer.com

Discord

Discord[15] is a cross-platform used for chatting, voice communication online, and screen-sharing as a video stream. Although designed for gamers, it was suitable for our cause. In our case,

[13] https://www.teamviewer.com/no/
[14] https://www.teamviewer.com/en/pricing/commercial-use/
[15] https://discordapp.com/

discord was not used as commonly as Slack for group work, however, we frequently used discord for one-on-one voice communication calls, to discuss or plan ideas and implementations for our app or our report. In the event of our group deciding it was necessary with a spontaneous and immediate meeting where a certain topic needed to be discussed, discord was a great tool to use for a simple group voice communicated meeting. Although slack was our main way of communicating altogether, discord was a great tool for any voice communication purposes.



# include Coders, programmers, tech-savvy devs, and all the other non-binary (genderqueer) fucks.

```
<string name="forecast_unit_pm10">pm10</string>
<string name="forecast_unit_no2">no2</string>
<string name="forecast_unit_o3">o3</string>
```

@bravobavo all done bro :))

bravobavo Last Tuesday at 3:35 PM
Thank you so much @MIB <3

MIB Last Tuesday at 3:35 PM
You're welcome bro :)))

May 22, 2019

gos Yesterday at 2:28 PM

y to meet up
I tend to be the
he time left
motivate myself

Bavo Meyvis 22:59 I går
Can agree. Nicely put. Balanced as all

Team members using the discord application to conduct a meeting.

References

Sprint 5: https://trello.com/invite/b/uycxaedN/53f53d9a47521f7ab27ad684a0957d50/sprint-5

Sprint 6: https://trello.com/invite/b/a4UH0uyK/bbbfa4e22b55917f11c87f7ba47753fb/sprint-6

Sprint 7: https://trello.com/invite/b/0fHmfM88/294776e3714100867e9ea2e86f6ddfa8/sprint-7

Sprint 8: https://trello.com/invite/b/qpioujrq/aec5b1a573b0b79d2e8d10f66c8c948c/sprint-8