BACHELOROPPGAVE:

Sprek i Gjøvik

FORFATTERE:
Markus Brovold
Anders Hagebakken

DATO:
May 19, 2014

Sammendrag av Bacheloroppgaven

Tittel:	Sprek i Gjøvik	Nr: -
		Dato: May 19, 2014
Deltakere:	Markus Brovold Anders Hagebakken	
Veiledere:	Mariusz Nowostawski	
Oppdragsgiver:	Gjø-Vard Orienteering	
Kontaktperson:	Arnfinn Pedersen, arnfinn@system-tre.no, 971 28 628	
Stikkord	Android, iOS, Database, Systemutvikling, Java, Objective-C, PHP, MySQL	

Antall sider: 143	Antall vedlegg: 7	Tilgjengelighet: Åpen

Kort beskrivelse av bacheloroppgaven:

Sprek i Gjøvik-prosjektet forsøker å engasjere til fysisk aktivitet i befolkningen. Gjennom applikasjonen Stolpejakten kan brukere registrere besøk på stolper som befinner seg på ulike steder i nærmiljøet ved hjelp av kameraet på mobiltelefonen. Brukeren kan også vise kart med alle stolper i området og egen posisjon ved hjelp av GPS, eller se på besøksstatistikk for både seg selv og andre brukere.

Brukere kan også få oversikt over egen aktivitet gjennom nettstedet vi har utviklet, stolpejakten.no. Vi har i tillegg utviklet administrasjonsmoduler til nettstedet, slik at arrangørene kan stå for den daglige driften.

For å knytte det hele sammen har vi utviklet en databaseløsning for prosjektet, som håndterer brukere, stolper, områder og besøk.

Applikasjonen er utviklet for Android i Android Studio og for iOS i Xcode. De er tilgjengelige for nedlasting gjennom Google Play og Apple App Store.

Summary of Graduate Project

Title:	Sprek i Gjøvik		Nr: -
			Date: May 19, 2014
Participants:	Markus Brovold		
	Anders Hagebakken		
Supervisor:	Mariusz Nowostawski		
Employer:	Gjø-Vard Orienteering		
Contact person:	Arnfinn Pedersen, arnfinn@system-tre.no, 971 28 628		
Keywords	Android, iOS, Database, Software Engineering, Java, Objective-C, PHP, MySQL		
Pages: 143	Appendixes: 7	Availability: Open	

Short description of the main project:
Sprek i Gjøvik is a public health initiative to stimulate physical activity in the population.

Through the application Stolpejakten, users use the smartphone camera to scan QR-codes from the poles placed widespread in the municipalities. An interactive map, which displays the location of the poles and the current location is built into the app. Various statistics is also available, like visited poles and a leaderboard.

Users can also track their activity through the web site we developed, stolpejakten.no. At this web site, we also included possibilities for administrators to manage poles, areas and news.

To make the different parts of the project work together, we created a database with a interface for the applications. This back-end system manages users, poles, areas and visits.

The Android application is developed in Android Studio, while the iOS application is developed in Xcode. Available for download in Google Play and Apple App Store.

Preface

This report document elaborates the process of developing a system consisting of an Android application, iOS application, website and server back-end.

After less than 5 months of development, the system is up-and-running. After being available to the public for only one week, more than 1500 users from the local community have registered, and more than 13000 unique visits have been made. The Android application has been downloaded 500 times, while the one for iOS has nearly 800 downloads.

We would like to thank our customer Gjø-Vard Orienteering, represented by Arnfinn Pedersen and Bjørn Arild Godager, for trusting us with this important assignment.

Contents

Preface . . . . **iii**
Contents . . . . **iv**
List of Figures . . . . **vii**
List of Code Examples . . . . **ix**
List of Abbreviations . . . . **vi**
1 Introduction **1**
1.1 Project description 1
1.2 Document structure 2
1.3 Project organization 3
1.3.1 Agile Software Development 3
1.3.2 Organization 4
2 Background **5**
2.1 Android 5
2.1.1 Activities 5
2.1.2 Fragments 6
2.1.3 Asynchronous task 6
2.1.4 Services 6
2.1.5 IntentService 7
2.1.6 Life cycle 7
2.1.7 Sensor 8
2.2 Database 9
2.2.1 Normalization 10
3 Requirement Specification **11**
3.1 Functional Requirements 11
3.1.1 Product Backlog - Feature List 11
3.1.2 Use Case Diagrams 13
3.1.3 High-Level Use Cases 16
3.1.4 Expanded Use Cases 20
3.1.5 Domain Model 25
3.2 Supplementary Requirements 26
3.2.1 Functionality 26
3.2.2 Usability 26
3.2.3 Reliability 27
3.2.4 Performance 27
3.2.5 Supportability 28

3.2.6 Legal Requirements 28
3.2.7 Licensing 28
3.2.8 Partial Releases 28
3.3 Constraints 29
3.3.1 Tools 29
3.3.2 Coding conventions 29
3.3.3 Data Storage 29
3.3.4 Hardware 30
3.3.5 Android version 30
4 Design and Architecture 31
4.1 Architecture 31
4.1.1 Deployment - Client Server 31
4.1.2 Structure - Three-Tier 32
4.2 Design 34
4.2.1 Application 34
4.2.2 Website 37
4.2.3 Database Abstraction Layer 39
4.2.4 Database 40
5 Implementation 41
5.1 Tools 41
5.2 Server-Side 42
5.2.1 Using the Facade Pattern 42
5.2.2 Database 44
5.2.3 Abstraction layer 46
5.2.4 Website 49
5.3 Android Application 63
5.3.1 User Interface 63
5.3.2 User management 64
5.3.3 Map 67
5.3.4 Sensor Usage 74
5.3.5 User Data Storage 76
5.3.6 Performance and Optimization 80
5.3.7 Network Communication 83
5.4 iOS Application 87
5.4.1 Organizing the development 87
5.4.2 Application design 89
5.4.3 Map 90

5.4.4 User data . . . 96
5.4.5 QR scanner and pole visits . . . 100
5.4.6 Supported devices . . . 101
5.4.7 Graphics . . . 102
5.5 Deployment . . . 103
6 Testing and Quality Assurance . . . 106
6.1 Unit testing . . . 106
6.2 Beta testing . . . 108
6.3 User feedback . . . 110
6.4 Field test and Quality control . . . 113
7 Conclusion . . . 115
7.1 Assignment Evaluation and Results . . . 115
7.2 Group work evaluation . . . 116
7.3 Further development and maintenance . . . 116
7.4 Conclusion . . . 118
Bibliography . . . 119
Appendix . . . 121
A Project agreement . . . 122
B Project Plan . . . 124
C Correspondence with Service Provider regarding Shared SSL . . . 134
D Meetings . . . 135
D.1 First meeting with Bjørn Godager - 13.01.2014 . . . 135
D.2 Status meeting - 21.02.2014 . . . 137
D.3 Status meeting - 21.03.2014 . . . 140
D.4 Status meeting - 24.03.2014 . . . 142

List of Figures

1 Android Life Cycle . . . 7
2 Feature List items in the BitBucket issue tracker . . . 12
3 Use Case Diagram - Website administrator functionality . . . 13
4 Use Case Diagram - Application User functionality . . . 14
5 Use Case Diagram - Website User functionality . . . 15
6 Domain model . . . 25
7 Android versions in Norway. ICS = Ice Cream Sandwich . . . 30
8 Client-Server Structure . . . 31
9 Three-Tier Layered Architecture . . . 32
10 Application Component diagram . . . 34
11 Application Class diagram . . . 36
12 Website Activity Diagram . . . 37
13 Database Abstraction Layer . . . 39
14 Database Design . . . 40
15 Facade Pattern Example . . . 42
16 Database Design . . . 44
17 Buddypress Registration Form . . . 50
18 Buddypress User Profile . . . 50
19 Register Poles on Phone . . . 52
20 Register Poles on Computer . . . 52
21 User Platform Statistics . . . 54
22 End-User Website Menu . . . 54
23 Administrator Website Menu . . . 54
24 First Page with Logged in User . . . 55
25 News Story Categories . . . 57
26 Pole List on Website . . . 58
27 Family Members Functionality . . . 60
28 Team Functionality . . . 61
29 Grid Menu . . . 64
30 List Menu . . . 64
31 ActionBar Menu . . . 64
32 Legend . . . 70
33 Pole Altitude Example . . . 72
34 GPS Accuracy Symbols . . . 74
35 Pole Visit Submission Flowchart . . . 76

36 ViewHolder Pattern example 81
37 Model View Controller Application Structure 89
38 Final Map View 95
39 Final Pole List View 95
40 Final User View 98
41 User Settings View 98
42 Normal Display 102
43 Retina Display 102
44 Icons 8 Example Icons 102
45 End-to-End Example Test 107
46 Number of Users in Database 110
47 Number of Pole Visits in Database 110
48 Manual Visit Activity 111
49 3.5-inch Screen Issue with QR Scanner Button 112
50 Propper QR Scanner Button on 4-inch Screens 112
51 Markus in the Field 114
52 Anders in the Field 114
53 Total Downloads in App Store 118
54 Total downloads in Google Play 118

List of Code Examples

5.1 PDO_CONNECT Class 46
5.2 Prepared Statements with PDO Class 47
5.3 Returned Array 48
5.4 Returned JSON-Array 48
5.5 CSS Responsive Website 53
5.6 Website - Administrator Functionality 59
5.7 Website - Register Poles Table 62
5.8 Login.php 65
5.9 Login Method from Android Application 66
5.10 TileMill CSS 68
5.11 Visible Bounds 69
5.12 Legend 71
5.13 Pole Altitude 73
5.14 GPS Accuracy Indicator 75
5.15 Visit Poles 77
5.16 Poles Table in SQLite 78
5.17 Pole Content Table in SQLite 78
5.18 Usage of SharedPreferences in User.class 79
5.19 ViewHolder 80
5.20 ViewHolder getView 82
5.21 ASyncTask 83
5.22 Add Parameter 84
5.23 FetchDataTask Usage 84
5.24 IntentService Usage in UserActivity 85
5.25 IntentService Usage in UserActivity 86
5.26 Loading mbtiles 91
5.27 Add Poles to Map 92
5.28 Create Marker Layer 93
5.29 Get Pole Altitude from Database 94
5.30 Upload User Visits 97
5.31 Save/Cancel User Setting 99
5.32 Scan QR Code 100
5.33 Convert iPhone Storyboard to iPad Storyboard 101
5.34 URL Class 104
6.1 Register Visit Test from Application to Server 108
6.2 QR Scanner Issue with Sony Devices 111

List of Abbreviations

ADT - Android Development Tools
CRUD - Create Read Update Delete
CSS - Cascading Style Sheet
FDD - Feature-Driven Development
IDE - Integrated development environment
JSON - JavaScript Object Notation
MVC - Model-View-Controller
PDO - PHP Data Objects
PHP - PHP: Hypertext Preprocessor
RAM - Random Access Memory
RDBMS - Relational Database Management System
SQL - Structured Query Language
TDD - Test-Driven Development

1. Introduction

1.1. Project description

Gjø-Vard Orienteering club piloted a low-threshold exercise project during summer 2013 called “Sprek i Gjøvik” (meaning; Active in Gjøvik). The project combines outdoor activities with modern technologies. The goal is to find a set of poles placed around the city, either by using a traditional map and compass, or by using an application on a mobile device.

Its popularity has grown quickly since the launch in mid-August 2013, with more than 1000 participants.

The first mobile app for Sprek i Gjøvik has been bought from a Swedish similar project. Because this solution has some adverse weaknesses, the project group in Gjøvik wants a better solution, which fully can be administered by Sprek i Gjøvik. This new solution can also be rolled out to similar projects in other norwegian cities.

The project has a wide support in both Norwegian Orienteering, Oppland Country Council, Gjøvik Municipality and more.

The project will be extended to cover both North and South Gjøvik with 50-100 poles by May 1st 2014, and several new locations by the summer. The target number of participants is 2000 by the end of summer 2014.

Our assignment is to create the website, Android application and back-end server. The back-end server has to handle all the users, areas and poles included in the project. The application has to display a map of an area with poles, and store visits made by users on the back-end server. Finally the website has to have functionality which allows a user to register their visits.

1.2. Document structure

We have decided to divide the project report into seven chapters.

Introduction
The 1st chapter contains the project description with the background information about the project we are a part of.

Background
The 2nd chapter is a short chapter which contains some of the terminology we will be using in the implementation chapter.

Requirement Specification
The 3rd chapter contains both the functional and supplementary system requirements of the entire system.

Design and Architecture
The 4th chapter elaborates the architectural and design decisions we made during the inception phase.

Implementation
The 5th chapter contains the elaboration of our solution. It contains a lot of code examples and figures of the interesting and challenging components of the system. It also contains screenshots of the different user interfaces, both of the application and the website.

Testing and Quality Assurance
The 6th chapter explains how we tested during the development phase and how we conducted quality assurance.

Conclusion
The 7th chapter discusses the outcome of the project and what we have achieved during the project period.

In addition to the previous mentioned seven chapters we have an appendixes with relevant information, such as the project agreement and meeting logs.

1.3. Project organization

1.3.1 Agile Software Development

According to Schwaber and Sutherland, a Scrum team should not have more than nine members and no less than three. A team too large creates a more complex process which is hard to manage. While a smaller team might have problems delivering on time [1]. Is is unnecessary for us as a group of two members to implement Scrum 100%. Which is why will not be following Scrum to the letter. We wish to use most of our time developing, rather than estimating, calculating member velocity and maintaining a Burndown Chart.

We will be following the principles of Scrum and borrow some artifacts from other agile processes instead. From Scrum we will be using iterations / sprints. At the end of each sprint we will be conducting status meetings with the Customer where we demonstrate the new features. After each demonstration we will evaluate the features, and plan the next sprint. The project has a deadline, and the user should not have access to the system until then. Which is why we will not release a new feature after each increment to the user, but demonstrate it to the Customer instead.

The group will implement a product backlog, but not the same Product Backlog artifact from Scrum. Instead we will implement Feature List from FDD. The Feature List fits us better, since there is less planning and estimation, and less complexity since the group consists of two members. The Customer also wants us to come up with new features. By using the Feature List it allows us to control the Feature List, rather than a Customer controlling the Product Backlog. It also allows us to prioritize which features to deliver after each sprint. Of course, we will have to finish the requirements set by the Customer by the deadline.

1.3.2 Organization

When we were accepted as the group to work with this assignment, we quickly scheduled a meeting with Bjørn Godager (Meeting D.1). Bjørn is both a assistant professor at HiG and a member of Gjø-Vard Orienteering. The purpose of the meeting was to get a feel of the project. The assignment was vague, and we wanted to know what they actually wanted us develop. We agreed to develop an Android application, provide a new website and a database solution.

The first thing we did after the meeting was to get the project agreement signed by both the Customer and the Supervisor (Appendix A). After signing the agreement and getting to know the task at hand, we created our initial project plan (Appendix B). The project plan has been a great resource, mainly the Gantt diagram, which kept us on schedule the entire project period. We have previous experience with poor planning and a lot of code-and-fix. This was not something we wanted to happen during our bachelor assignment.

When the Design and Architecture was in place, we could start on the development phase, which we were eager to get started on. Since the group only consists of two members, structuring our work has not an obstacle. We have met up, alternating between working at each other's homes, every day of the week. The daily structure has ensured that we had control and worked efficiency during the entire project period.

We have been working on separate parts of the system parallel with each other, side by side for support and guidance. We have been selecting different features we wanted to work on. For instance during the first part of the development phase, one did the back-end and one did on the front-end component of the website. While developing the application we split the workload in the same fashion as we did for the server-side. One did the user feature and one did the map feature. By separating the workload, we have been able to produce the features the Customer required, and we wanted to implement.

2. Background

We feel it is necessary to include a short technological background chapter for a reader to be able to understand some the terminology used in the implementation chapter.

2.1. Android

2.1.1 Activities

One of the most basic elements of in developing an Android application is an Activity. An activity is the first thing you see in an application. For instance, if you create a simple "Hello World" application, what you see on the screen is an activity with the text "Hello World". An application can consist of many activities. For instance, in an e-mail application, an activity is used to display your inbox, another activity is used when you are writing an e-mail. To create an activity, it has to extend the "Activity" class.

An activity can start another activity. One can also pass data between the activities using "Interprocess communication" (IPC). There are different ways of implementing IPC, for instance, one can store flat files with the necessary content, or store something in a SQLite database, then access it the desired activity. One can also use what is called Shared Preferences, which works like Java's Preferences, which means one can store data on a device. For instance User login information, so a user do not have to re-enter login information each time a user opens an application. One can also use what is called an "Intent". We can put string, booleans and integers as well as other data types into and intent, and pass them on to the new activity, then get them from that intent. In an activity, one can add what is called Fragments.

2.1.2 Fragments

"Fragments" can be used to "divide" an Activity into fragments. An application can consist of many fragments. As we will elaborate further in our implementation chapter, we are using fragment as the container of our mapview. Another usage, is to have different layout in landscape and portrait orientation. Then one can add an extra fragment next to the fragment which was visible in portrait orientation.

2.1.3 Asynchronous task

An Android application has to work in a certain way. An application does not lock to wait for input. If you ask a user for username and password, the remaining code will be executed. If one wants to display an image or text from Internet, one has to do this in a certain fashion. Android does not allow HTTP connections on the main thread. By using Asynchronous task, you can preform asynchronous off the main thread. This way, one make sure the application does not hang on the main thread while processing a huge image or downloading large amounts of text. One should also add some form of progress bar to inform the user something is going on behind the scenes. We will for instance do all the communication with the database interface / abstraction layer with this mechanism.

2.1.4 Services

An application can also make use of what is called a "Service". A service runs in the background, and has no user interface. One has to extend the Service class to create a service. An activity can work as a user interface for a service. A good example is a music player, which presents you with the option of playing/pausing, next song etc. When you lock the screen, the activity goes away, but the service keep playing the music. Another example would be something like RunKeeper, which tracks your GPS position while your are running. The background service tracks your position, then when the activity resumes, you can see how far you have ran and you can see where you have been running with the data collected from the service.

2.1.5 IntentService

A special occurence of Services in Android is the IntentService. The IntentService provides a set of methods which makes it easy to broadcast a result using Intents. IntentServices can be used to download some data and then send a broadcast back to the Activity it was started from. When the Activity receives the broadcast, it can update for instance a ImageView. The way this differs from a ASyncTask is that a IntentService does not block the UI-thread, which enables the user to interact with the application uninterrupted.

2.1.6 Life cycle

An activity has a life cycle. Figure 1 from the Android Developer website illustrates how a life cycle looks. When you press the application icon to start the application the main launcher activity's onCreate() will be fired first. This is an inherited method from the "Activity" class. This method will be fired every time a user changes orientation from portrait to landscape and back. Then the onStart() and onResume() method will be fired. After that, we have a visible activity. When a user switches activity, the onPause() and onStop() will be fired. When a user returns to that activity, it will re-fire the onStart().

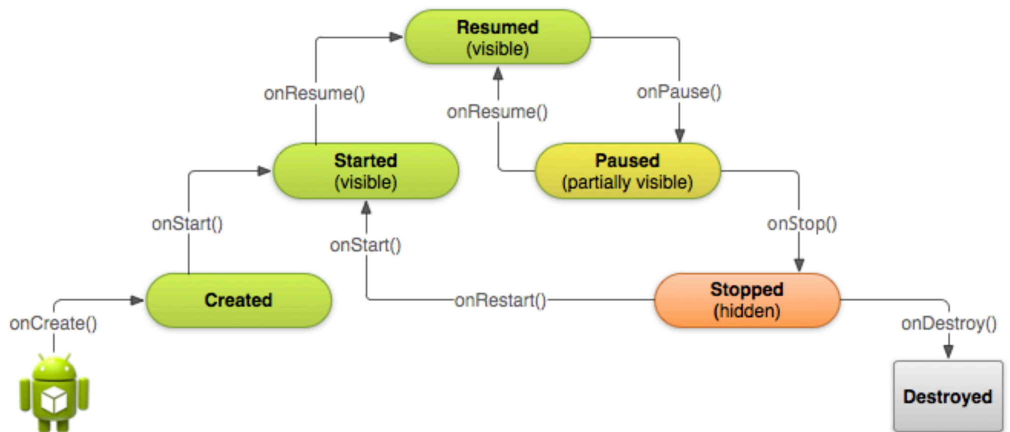


Figure 1: Android Life Cycle
[2]

Since Android "resets" the activity when you either rotate or change to another activity, one has to make sure to store what was visible in the activity, such as text. If you fill in a textfield, and the screen displays the content of that textfield, it has to store it for when a user might rotate the device. So it has to be stored before you rotate, then restored in the onCreate()/onResume().

2.1.7 Sensor

Android has support for every sensor that comes with a device. To make use of this sensors, one has to add a SensorManager, which is an interface for communicating with the different sensors in a device. With the SensorManager, one can listen to the different sensors. For instance, GPS, accelerometer and orientation (can be used to make a compass).

2.2. Database

Databases are collections of data. The task of a database is to store data in a systematic way, for instance in tables, so that they can be easily organized and used. We will use relational databases in our project. When a database is relational, there is a set of relations between the different tables. For instance when a person has a phone number, the person is in one table, and the phone number is in another table. To decide which phone number belongs to who, a relation exists between the two tables. Since MySQL is one of our constraints, we will focus on MySQL. In a relational database, a table consists of columns with the various data-types. The first thing to do is to create a table. This can often be done with a graphical user interface, like myPhpAdmin, or by writing sentences of SQL-code, called statements. This example shows how to create a new table with the columns firstname, lastname and birthday. The name of the will be Persons:

```
1 CREATE TABLE Persons
2   (firstname,
3   lastname,
4   birthday)
```

This is maybe the simplest form of a table, with no relations and no data types. To specify what the columns should contain, we add some data types to the statement.

```
1 CREATE TABLE Persons
2   (firstname varchar(40),
3   lastname varchar(40),
4   birthday date)
```

A primary key should also exist in every table in the database. The primary key must be a unique value for each row. If a column in a table is guaranteed to be unique, it can be used as the primary key. In many cases however, it is necessary to add a integer value which increments by one for each new row. In our Persons-table, it is not impossible for two persons to have the same name and the same birthday. For the RDMS to distinguish these persons, we add the column personid, and set it to be the primary key.

```
CREATE TABLE Persons
  (personid int(10) AUTO_INCREMENT,
  firstname varchar(40),
  lastname varchar(40),
  birthday date,
  PRIMARY KEY(personid))
```

We can now create another table, Phonenumbers. Since it is possible for one person to have several phone numbers, personid cannot be the primary key of the Phonenumbers table. Because it is also possible for several persons to share a phone number, the phone number cannot be the primary key. To solve this problem, we add a phonenumberid to the table. The statement to create this table would then be:

```
CREATE TABLE Phonenumbers
  (phonenumberid int(10) AUTO_INCREMENT,
  phonenumber int(20),
  personid int(10),
  PRIMARY_KEY(phonenumberid)
```

Now that we have two tables, we can create the relationships. If we are dealing with complicated tables it often is easier to create the relationships, or constraint as they are called in MySQL. In this case, a row in the phone-number should be deleted if a person is deleted, but a person should not be deleted if a phone number is deleted. Since we already created the tables, we can use the ALTER TABLE statement.

```
ALTER TABLE Phonenumbers
  (CONSTRAINT has_phonenumber
    FOREIGN KEY (personid)
       REFERENCES Persons (personid)
  ON DELETE CASCADE
  ON UPDATE NO ACTION)
```

2.2.1 Normalization

Database normalization is a method to ensure that data is not redundant, i.e. the same data should not appear more than one place in the database. The first task to normalize the database is to make sure all columns/attributes are atomic. The attribute "car" is a nonatomic attribute, if it for instance contains "1985 BMW 3series". The attributes "car_make", "car_model", "production_year" is atomic, and would in this example contain the values (BMW, 3series, 1985). An attribute is atomic when further decomposing the value would not be meaningful.

3. Requirement Specification

3.1. Functional Requirements

3.1.1 Product Backlog - Feature List

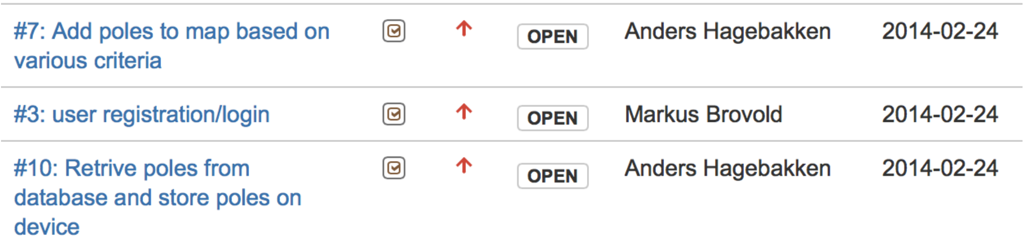
As mentioned earlier, our development process is only loosely based on Scrum, which is why we have decided to discard the Product Backlog artifact. Instead we have chosen to borrow an artifact from Feature Driven Development called Feature List. The reason behind this is that we do not want to use a lot of time estimating beforehand. We feel this is the best option for us. By doing it in this manor, we get a more agile development process, since we are only two people, and can simple choose feature after feature from the list. We do not need to plan as much ahead as if we were a team of for instance eight.

Our Feature List will have the standard format <action> <result> <object> [4]. For instance: Display pole list in map activity to a application user

ID	Name <action> <result> <object>
1	Display map in application to a application user
2	Display registration form on website to a website user
3	Display registration form in application to a application user
4	Display CRUD options for family members to a website user
5	Display CRUD options for poles to area administrator
6	Display CRUD team options to website user
7	Display QR Scanner in application to a application user
8	Store pole visits on device for a application user
9	Upload pole visits from device to database for a application user
10	Display pole list in map activity to a application user
11	Display form to register pole visits to website user
12	Store pole visits data provided by website user

We will use the processes of FDD as a guideline and select features from the Feature List. Which for us means that each of us can work on different features parallel with each other. By following this pattern, it allows us to start on a feature, complete it, implement it and then select a new feature until the feature list is empty.

The same Feature List is added to the Bitbucket issue tracker (Figure 2). By using the issue tracker, it allows us to separate the core features from the trivial ones by categorizing the features. Then we can select one feature, assign it to our selves, then set the status from "New" to "Open". After we have finished the feature, we change the status to "Resolved", which means the feature is done and implemented into the system.



Issue	Status	Assignee	Date
#7: Add poles to map based on various criteria	OPEN	Anders Hagebakken	2014-02-24
#3: user registration/login	OPEN	Markus Brovold	2014-02-24
#10: Retrive poles from database and store poles on device	OPEN	Anders Hagebakken	2014-02-24

Figure 2: Feature List items in the BitBucket issue tracker

3.1.2 Use Case Diagrams

Use Case Diagram - Website administrator functionality

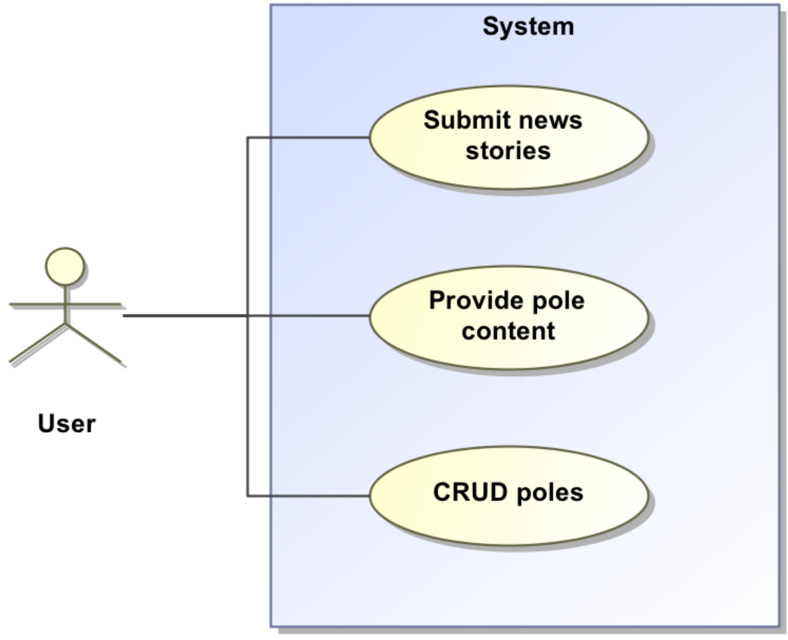


Figure 3: Use Case Diagram - Website administrator functionality

The Use Case Diagram for a Website administrator is simple and straight forward. The User/Actor (Website administrator) need to have the options to write news stories to the area of which an administrator administrates. Then there must be CRUD options for all poles, which allows the administrator to handle an area and its poles without needing to include us. The last options is to provide the pole content. Some of the poles will be sponsored and some will be placed on a historical location. Therefor the Customer wants to be able to add a description to that pole.

Use Case Diagram - Application User functionality

Figure 4: Use Case Diagram - Application User functionality

The user must be able to register in the application. Then one maybe should be provided with the option if they want to create or join a team. A user should also be able to view the map and poles based on different criteria, such as display only one difficulty or only unvisited poles. The tag pole option is a core functionality of the application. This means that the visit must be stored on the device, and uploaded to the server. At last, we should provide the user with some statistics, both their own visits and show a leaderboard with the users who have visited the most poles.

Use Case Diagram - Website User functionality

Figure 5: Use Case Diagram - Website User functionality

The website user will have a lot of the same options as the application user. They should be able to download a PDF version of the map, view the same statistics as in the application and provide feedback on either missing poles or general feedback. The user should be able to submit their visits in a form using the character code (QR code value) from the pole. Then a user should be able to add family members to its account. At last, a user has to be able to create a team and display the team data (members and visits).

3.1.3 High-Level Use Cases

The High-Level use cases splits the system into User / Administrator and website / application. This allows us to elaborate the Use cases and split the Feature List into smaller tasks.

Use Case	**User registration**
Primary Actor:	User
Goal:	The user data is stored in the database
Description:	The user should be presented a registration form both in app and on the webpage. Fields should at least be email, first name, last name, area affiliation. Maybe join/create team and add family members to account.

Use Case	**Download map**
Primary Actor:	User
Goal:	The user has downloaded the map
Description:	The user should be able to download the map in A4 size, ready for printing.

Use Case	**Provide feedback**
Primary Actor:	User
Goal:	The user provides Gjø-Vard Orienteering with feedback
Description:	The user should be presented with a feedback form.

Use Case	**Administrator CRUD pole**
Primary Actor:	Administrator
Goal:	A new pole is CRUD in the database
Description:	When an administrator is logged in, one must be able to do CRUD operations on a pole

Use Case	**View statistics**
Primary Actor:	User
Goal:	Get information about visited poles
Description:	User should choose between different statistics

Use Case	**View map with poles**
Primary Actor:	End-User
Goal:	Get position on map, both user position and pole position
Description:	Possibility to show a list of poles, which can be sorted on on difficulty.

Use Case	**Tag pole**
Primary Actor:	End-User
Goal:	Get points for visited poles, keep track of visits
Description:	User should be able to register a pole as visited.

Use Case	**Manual pole submission**
Primary Actor:	User
Goal:	The user gets their visited poles stored in database
Description:	The user should be presented with a form where one can submit pole QR codes manually.

Use Case	**Team CRUD**
Primary Actor:	User
Goal:	The team data gets stored in database
Description:	The user should be able to do CRUD operations on a team

Use Case	**Add news**
Primary Actor:	Administrator
Goal:	The users can read news
Description:	The administrator should be able to submit news to a news feed.

Use Case	**Provide pole content**
Primary Actor:	Administrator
Goal:	The users can read information about certain poles
Description:	The administrator should be able to submit pole information

3.1.4 Expanded Use Cases

We selected some of the core functionality and the more complex tasks in the system which has to work in a certain fashion. There must be a control of the system flow in these selected use cases, that is why we decided to expand them. Then we know how and when to handle the different errors.

Every use case consists of a Scope, in our case it is for the website and/or the application. Then we have a Primary Actor, which is either an End-User or an Website Administrator. Next item is the Precondition which as to be for filled before the Main Success Scenario can occur. Second to last, it is the Postconditions, which in our cases ensures the either presentation, storage and/or integrity of the user data.

At last, we have the Main Success Scenario with Extensions. Main Success Scenario occurs when every condition is for filled. If something goes wrong, it is "handled" in the Extension.

Use Case	**User registration**
Scope:	Website and application
Primary Actor:	End-User
Preconditions:	User must have Internet access on device (PC/Android)
Postconditions:	User data must be stored in database after registration

Main Success Scenario:

1. User fills in user data in registration form:
    1. Email address
    2. Name
    3. Join team/Create team
        1. Submit team name
        2. Join team
    4. Add family members to account
1. User presses submit button
2. Data stored in database

Extensions:

2 Invalid login data:
  1. System shows failure message
  2. User returns to step 1

2 Invalid user data:
  1. System shows failure message
  2. User returns to 1 and must fix errors

Use Case	**View statistics**
Scope:	Website and application
Primary Actor:	User
Preconditions:	To show own accomplishments, user must be signed in
Preconditions:	Internet connection
Postconditions:	The data is presented to the user

Main Success Scenario:

1. User chooses which statistics to show
   1. Own user
   2. All users
1. Results is brought to screen

Extensions:

2 Invalid login data:
   1. System shows failure message
   2. User can look at all users scores

2 No data:
   1. System shows failure message

3 No internet connection:
   1. System shows failure message

Use Case	**View map with poles**
Scope:	Application
Primary Actor:	User
Preconditions:	The user must have GPS activated
Postconditions:	The map is presented to the User

Main Success Scenario:

1. User opens map activity in app
2. The user's GPS location is displayed in map
3. User options in map should be:
    1. User can toggle between displaying visited/unvisited/all poles in map
    2. User can select displaying only poles with a certain difficulty
    3. Map activity displays the user's preferences
1. Clicking on marker displays lat/long, id and name
2. User can now go look for poles

Extensions:

2 GPS is disabled:

1. System shows failure message
2. User enables GPS - returns to activity.

Use Case	**Add pole codes manually**
Scope:	Website
Primary Actor:	User
Preconditions:	The user must be signed in
Postconditions:	The data must be stored in database

Main Success Scenario:

1. User is presented with code submission form
2. The user submits pole codes
3. Database stores the data about the user's visits
4. Web page provides feedback (success/failure)

Extensions:

2 Illegal pole codes:
   1. System shows failure message
   2. User corrects the error, re-submits

3 Database error:
   1. System shows failure message
   2. User must try again later

3.1.5 Domain Model

Our domain model illustrates how the system is intwined. With the domain model (Figure 6), we have decided to split it into three different sections: Data, Logical and Presentation. The reason being that we want some structure in our model, since we will be using it later as the base for our architectural structure.

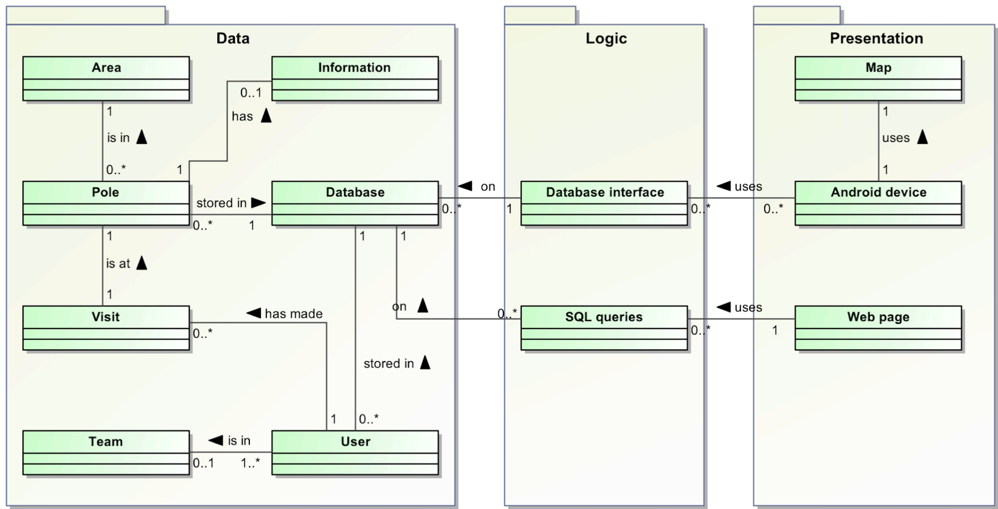


Figure 6: Domain model

The data layer illustrates the basic entity relationships in the database, which will be elaborated further in Database Design (Section 4.2.4). The communication between the application and the website should be handled in the logical layer. The logical layer consists of the database interface / abstraction layer. This layer will be used by the application. Next we have the SQL queries which will used by the website. We might use the same interface for the website to ensure low coupling, but this will be decided later. At last we have the presentation layer, which is a graphical interface to the user, either in the form of the website or the application. The application will use the for mentioned logical layer to communicate with the data layer.

3.2. Supplementary Requirements

3.2.1 Functionality

All the different functionality has already been described in the use cases. As seen in our use cases, our system have two different user groups. We have the user and we have the administrator. The user is split into two different user groups, one application user and one web user. An application user and a web user will have a lot of the same functionality. We have decided to limit some of the functionality to only be available on the web page. Such as adding a family member to your account and CRUD a team. This will help us limit the potential errors which might occur during user registration.

3.2.2 Usability

The previous application was, according to our customer, not a success. It was not very user friendly and not very intuitive. We have to ensure our application will be well received by the users. That is why we will create an application which will be easier to use and understand. The user group consists of a lot of different people of all ages, hence there will be a great span when it comes to technological skills. That is why we will provide a walkthrough when the user first launches the application. This walkthrough will of course be available in the help menu of the application. By doing this we make sure the user gets an introduction in how the application works. Hopefully this will ensure the users continue to use the application, instead of using the manual pen and paper submission-method.

3.2.3 Reliability

A device or application might malfunction. That is why we have to implement a mechanism to store the user data while the user is out and about looking for poles. This will ensure that the user data is not lost incase of a application malfunction.

Our data is stored in a database from a service provider which has daily database backup. This will ensure that the user data does not get lost. Incase of a malfunction, we might have to do a roll-back. The users must to be notified if data loss occurs. The service provider also guarantees 99.9% uptime.

3.2.4 Performance

We have to consider a device's battery life and therefore do minimal processing on a device. We should do all the map pre-processing on a computer, then attach the map data depending on the selected map technology. If a task is taking some time, we need to display a progress bar so the user knows something is happening. The last thing we want is Android Not Responding. To make sure the application runs smoothly, we should download all required data at start, and use some splash mechanism while the data is downloading. Because of these mechanisms we ensure the application is responsive and do not freeze while loading activities. The most important thing is to provide the user with the proper information, if the application is doing a lot of work.

3.2.5 Supportability

We will develop the application in Norwegian and English. Android applications can easily be translated and localized. If someone requests another application language, the website's news feed can be used to advertise for volunteer translators.

Not all requirements are specified yet, but if we are going to implement e.g. distance to pole from your GPS location, we can have the user select between imperial or metric units. Different units of measurement might also be implemented in pole content, e.g. “This marker is located 300 meters/328 yards above sea level”.

3.2.6 Legal Requirements

Data is stored in a safe place by a professional company. We do not handle any sensitive or private user data.

3.2.7 Licensing

We will only use open source libraries. The project does not have any commercial interest. It is a non-profit volunteer project from Gjø-Vard Orienteering.

3.2.8 Partial Releases

After each sprint we will provide a new prototype of the application to demonstrate for our customer. There will be no public releases in Google Play until the application is considered done, and can be tested by the Customer.

3.3. Constraints

3.3.1 Tools

The Android Application should be developed using Android Studio as the IDE. This means we will be using Gradle as build system, since it is used as standard in the IDE. Git will be used for version control and source code management through a private repository on Bitbucket.org.

3.3.2 Coding conventions

For the application we will be using standard Java/Android coding conventions [5]. Naming variables and methods will be done in English, and in camel case style. Class member variables should have the m_* prefix. An example: m_ThisIsMyVariable. Every class will be under the package *“no.hig.andmark.sprek.“*.

Coding style and conventions for the website will depend on the programming language.

The code should be commented in such a manner the future development and maintenance can easily be continued by someone else.

3.3.3 Data Storage

The application should use SQLite for local storage on device. The server from the service provider comes with MySQL only.

3.3.4 Hardware

Android Application

The application will need access to the built-in GPS sensor in the device. Depending on the detail level of the maps, the application will use a larger amount of RAM. Access to the camera is necessary to use the QR-scanner.

Website

The website should fit all popular web browsers in resolutions from small windows to fullscreen windows on high-resolution clients.

3.3.5 Android version

84% of Android users who downloaded the application RuterBillett (a public transport ticket-app for Oslo and Akershus) between 18 December 2012 and 18 December 2013 was using Android 4.0 (API level 13) or higher [6]. This number is a good guideline for us on which version the Norwegian Android-user has. We will be targeting 2.3 and up, but if we need functionality which require a higher version, we will adjust thereafter. Although we want to support as many devices as possible, the functionality should not suffer at the expense of device supportability.

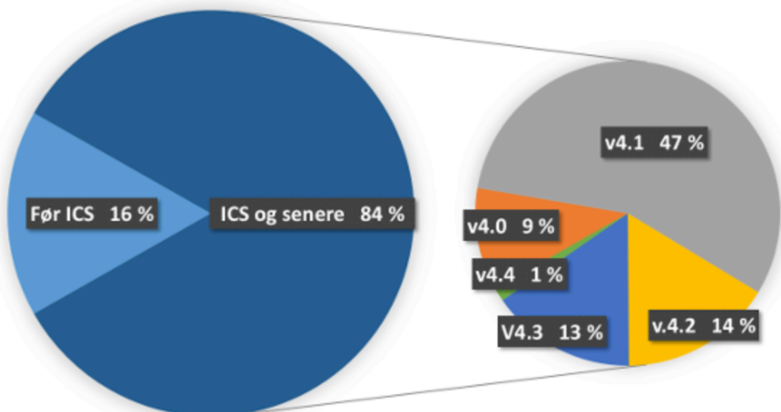


Figure 7: Android versions in Norway. ICS = Ice Cream Sandwich

4. Design and Architecture

4.1. Architecture

4.1.1 Deployment - Client Server

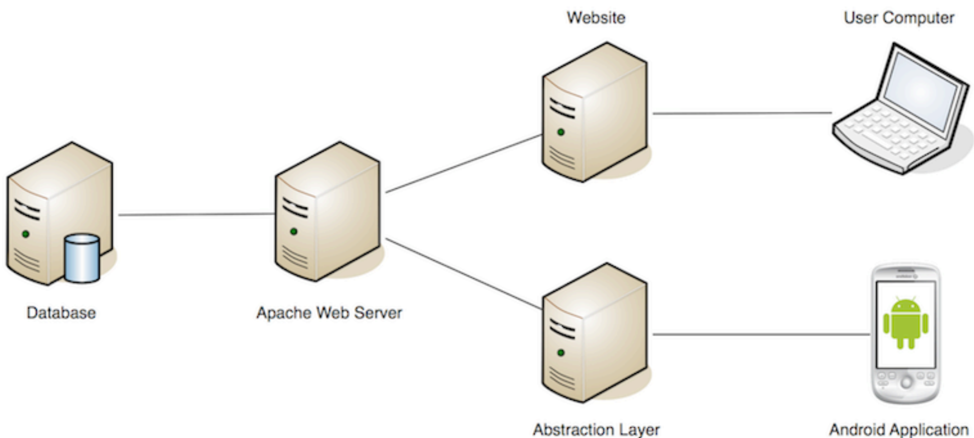


Figure 8: Client-Server Structure

We will distribute the entire system using a client-server architecture. One of our restrictions is using an Apache web server to host the website. Another restriction is will be using MySQL as the RDBMS.

A user will have the possibility to use the application, the website or both. The application and the website will mostly have equal features. For an application to communicate with the database, we have to provide an interface. The interface, or a database abstraction layer, must function independent of platform.

Most of the data processing will be done on the server. The application should primary be used to handle the presentation. By doing most of the processing on the server, we make sure the device's battery life is spared. The website will also be responsible for presenting the data in the database.

4.1.2 Structure - Three-Tier

The system will be structured using a three-tier architecture (illustrated in Figure 9). All of communication in the system will be using the HTTP protocol. By implementing this structure we also ensure that the presentation layer does not directly communicate with the data layer. The communication between the layers are linear, and all of the communication must go through the logical layer. This is a security measure, and ensures the integrity of the data stored in the database. The data will always be processed by the abstraction layer before it will get inserted into the database. If the data format is not correct, it should not get inserted.

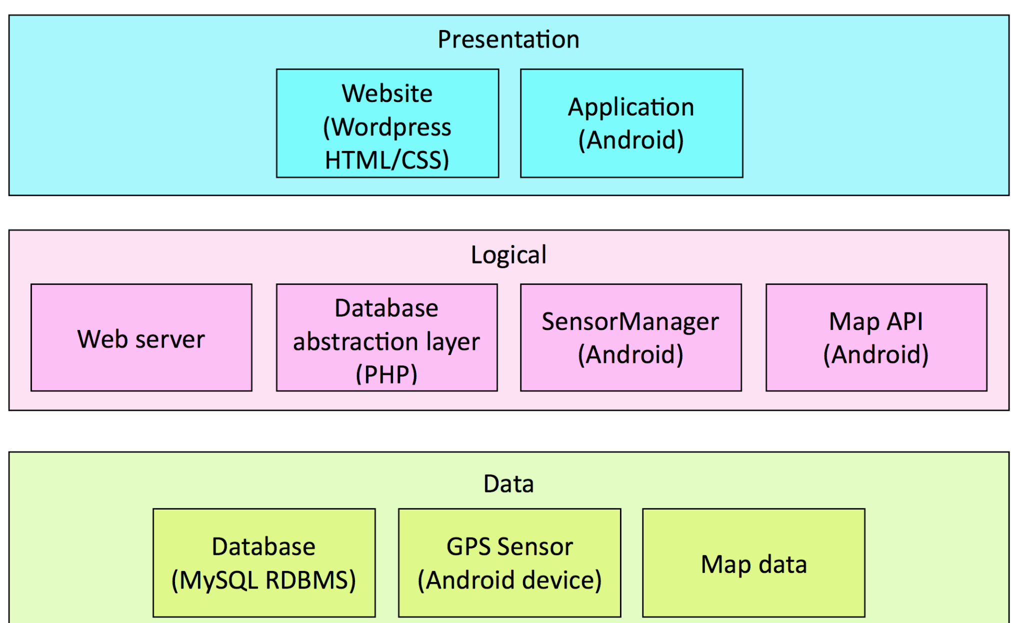


Figure 9: Three-Tier Layered Architecture

Figure 9 represents the system structure. First we have the presentation layer which will be responsible for providing a user interface to the user. We have two elements in the layer: Website and Application. These different components will be responsible for displaying the same data on different platforms.

Secondly we have the logical layer. This layer provides the communication between the data layer and the presentation layer. The website will be hosted on an Apache web server. We will also store the database abstraction layer on the Apache web server for communication with the back-end server. The

SensorManager is the native package in Android which allows us to access the GPS sensor on the different devices. Finally we need a map API, which will be used to retrieve and display map from the map data source.

Finally we have the data layer. This is our source of data, such as the GPS sensor. The map data source will be determined in the elaboration phase. At last we have the database where all the user data is stored.

By using the three-tier structure, we provide a clean user interface to the data stored in the data layer. A User Interface which can be used by both system users (Administrator and End-User). We also simplify the maintenance process, as well as simplifying the future development. It ensures loose coupling between the layers and allows development work to be done separate in the different layers without compromising other functions in the system. For instance we can edit the abstraction layer without having to do changes in the application nor the website.

4.2. Design

4.2.1 Application

Component Diagram

The application must to be designed generically. This means developing an application which can be used regardless of map and pole location. This ensures scalability. Which means if Gjø-Vard Orienteering want to expand the project and include new areas and poles, it can be done without changing anything in the code. Figure 10 represents the Component diagram of activities which will included in the project.

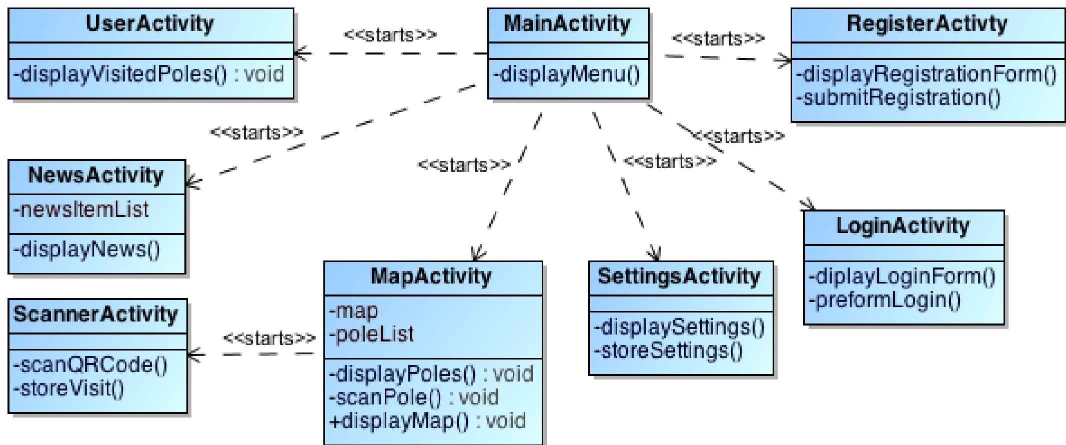


Figure 10: Application Component diagram

LoginActivity

Let user type credentials and submit. Should initiate a connection to the server to verify. Should contain ability to switch to RegisterActivity if user is not already registered.

MainActivity

View a welcome-screen to the user with menu to navigate in the application. Should appear immediately after login is finished.

MapActivity
Displays the map with poles. Should also allow user to view a list of poles. Should contain a button to start ScannerActivity.

NewsActivity
Should display the latest news for the user-selected area from the web site.

RegisterActivity
Lets the user register for the first time. Should initiate a connection to the server to verify details and create new user. Should also contain a button to switch to LoginActivity, if user is already registered.

ScannerActivity
Let the user use the camera to scan QR-code on pole. User-interaction aside from pointing the device in the right direction should not be necessary. Should give feedback to user when QR has been scanned, or if any errors occur.

UserActivity
View information about logged in user. Should display number of poles taken and current ranking. Should also view a leaderboard, and a list of poles taken by logged in user.

SettingsActivity
Display the settings that the user could change, such as "enable GPS" and "Colorblind assistant". Should also view current version of application, and a form to submit feedback to developer.

Class Diagram

We will also integrate the different classes which we know of, such as Pole and Area. There is also need for a DatabaseHandler class which will be used to access and store the different Areas and Poles. The DatabaseHandler will be accessed by the MapActivity component. It is also necessary to implement a class which represents a User, hence the User class. The User class data will be accessed by the UserActivity represented in Figure 10. We will also implement a NewsActivity, which means we need to populate the activity with news items, hence the NewsItem class.

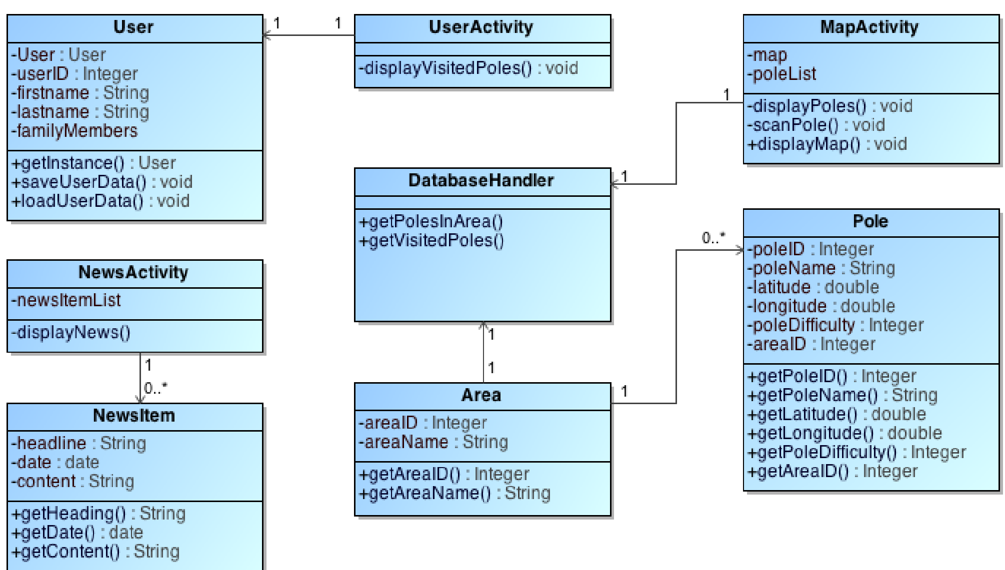


Figure 11: Application Class diagram

The Component diagram (Figure 12) shows the flow of the functionality we need to implement in the website. There will be different menus based on if you are an area administrator or an End-User. We have elaborated the basic functionality in the use case diagrams, and this activity diagram will be the basis of controlling the flow in the implementation of the different features of the website.

4.2.2 Website

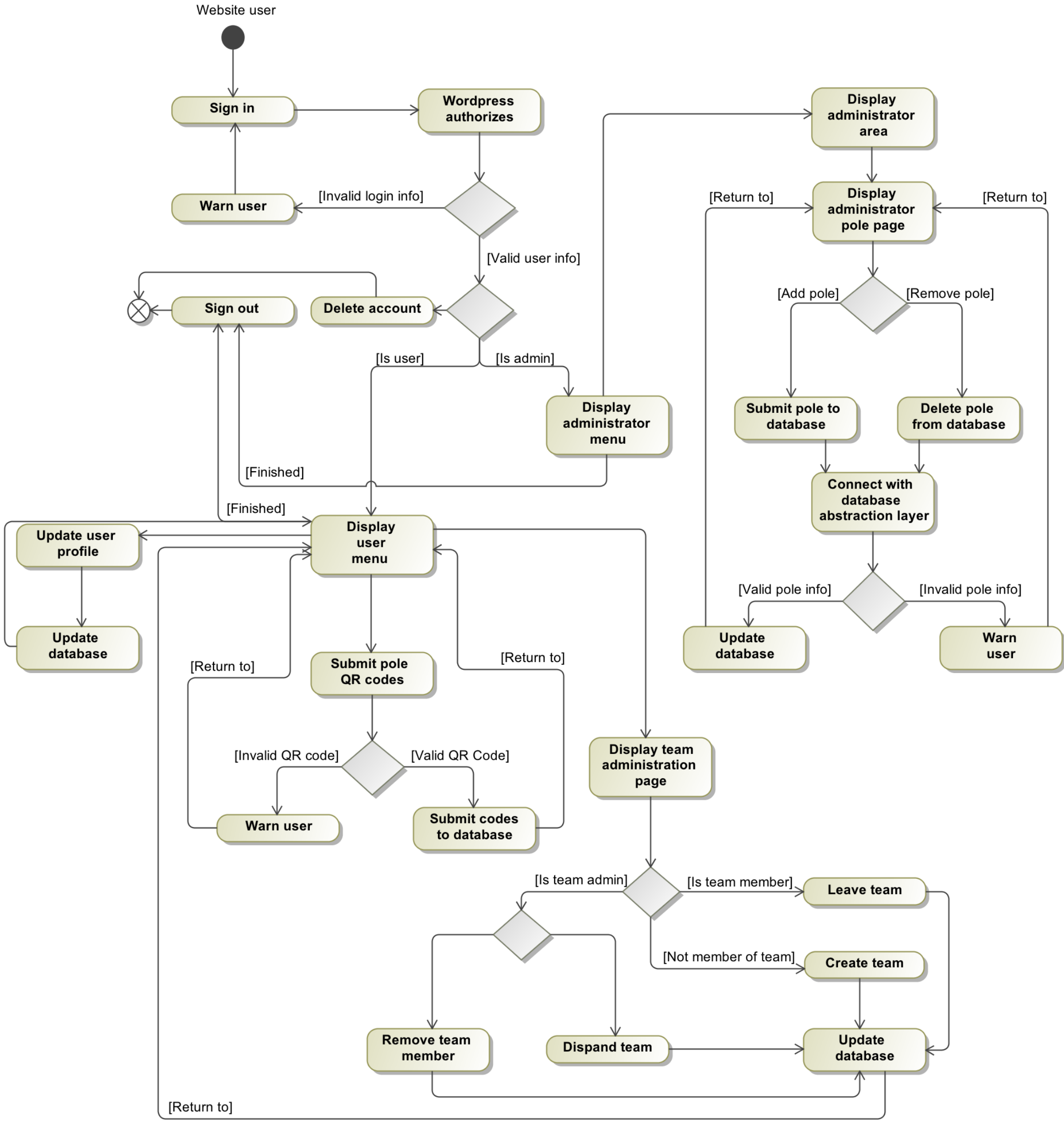


Figure 12: Website Activity Diagram

We will also integrate the different classes which we know of, such as Pole and Area. There is also need for a DatabaseHandler class which will be used to access and store the different Areas and Poles. The DatabaseHandler will be accessed by the MapActivity. It is also necessary to implement a class which represents a User, hence the User class. The User class data will be accessed by the UserActivity represented in Figure 10. We will also implement a NewsActivity, which means we need to populate the activity with news items,

hence the NewsItem class.

The activity diagram (Figure 12) shows the flow of the functionality we need to implement in the website. There will be different menus based on if you are an area administrator or an ordinary user. We have elaborated the basic functionality in the use case diagrams, and this activity diagram will be the basis of controlling the flow in the implementation of the different features of the website.

4.2.3 Database Abstraction Layer

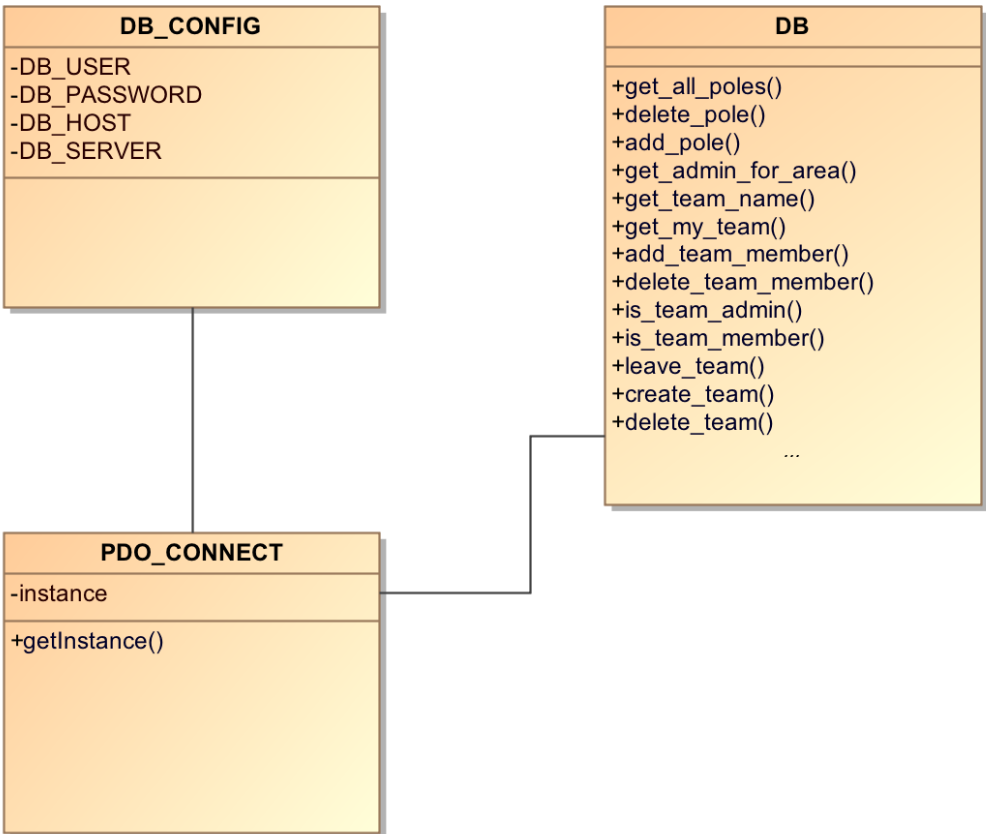


Figure 13: Database Abstraction Layer

The abstraction layer will need the three classes in Figure 16. We need a configuration class, DB_CONFIG, to handle the connection variables. Then we need a class, PDO_CONNECT, which that handles the actual connection. At last we need a class which we can use to submit and request data, the DB class. This class might be used by both the website, as well as the application.

4.2.4 Database

Based on the application class diagram (Figure 11) we know which classes we need to implement in the database. Figure 14 illustrates the initial database design. We know we need a User table to store all the user data in. Then there is also need for an Area and Pole table to store the different areas and poles. The different poles will have some information attached to then, therefor we also need a table to hold the Pole information, hence the Information table. At last we know we need to store the visits which the users makes in a table, hence the Visits table.

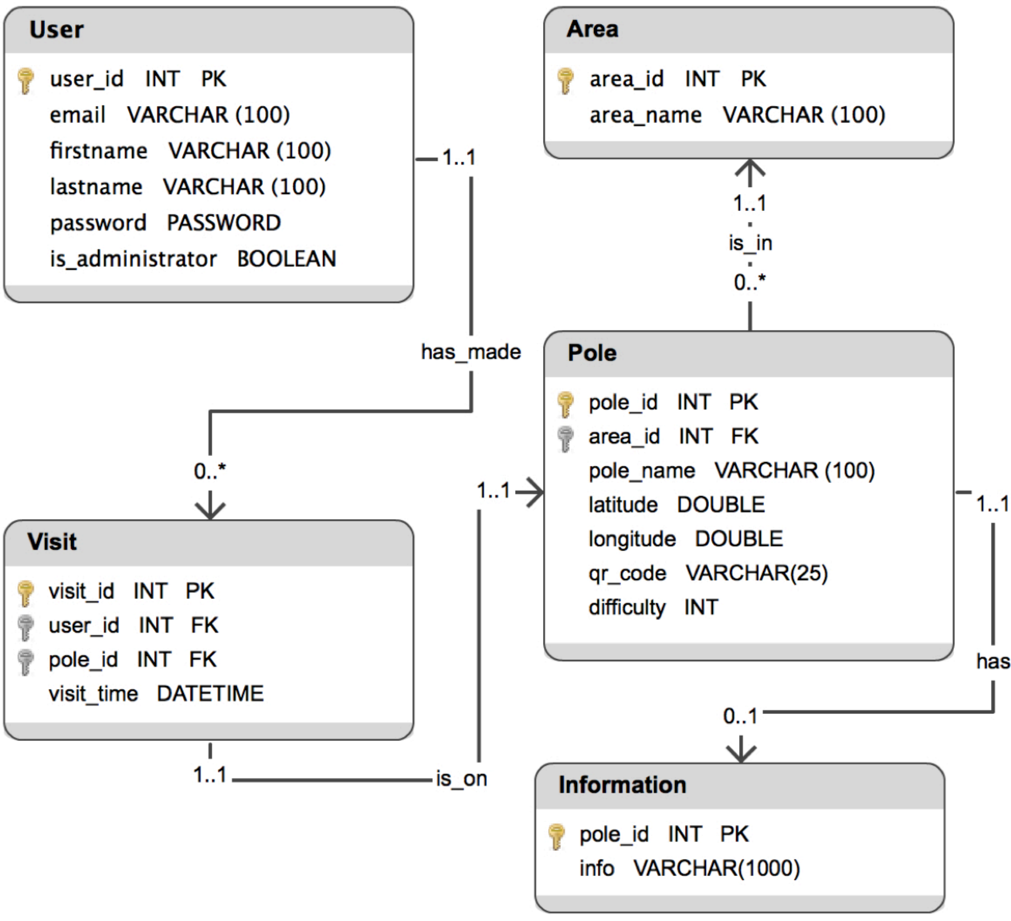


Figure 14: Database Design

5. Implementation

This chapter with elaborate how we implemented each part of the system. We used the Design and Architecture chapter as a guide on how the system should be implemented and how to handle interaction between the different components. The chapter is divided into four different sections: Tools, Server-Side, Website and Application. Tools explains all the tools we have used during the project period. All the way from signing the Project agreement to submitting the report to Fronter. The next section explains how we implemented the Server component of the system. The last two sections explains how we implemented the Website and the Application.

5.1. Tools

We used BitBucket as the Git service provider, issue tracker and feature list handler for the entire development phase. As a Git graphical client we went with SourceTree because it supports Mac (which all of the group members use). When it comes to IDE we choose Android Studio over Eclipse because we wanted to try this tool since it is made explicitly for Android application development. For unit testing in Android Studio we went with JUnit.

For the website development we went with FileZilla as the FTP client. As text editor for the PHP files in the abstraction layer we went with Texastic because it has great syntax highlighting and is made for Mac. As database administration tool we went with phpMyAdmin because this is what the service provider had pre-installed.

During the entire project phase we used Google Docs to collaborate on the content. We used TexShop (Latex client for Mac) to finalize the thesis. To create all our graphics and diagrams, we used Magic Draw, Creately, Omnigraffle and Astah Professional.

5.2. Server-Side

5.2.1 Using the Facade Pattern

The Facade pattern is a very commonly used pattern in software development. It is found in every piece of software attached to a hardware, for instance a mobile device. When a user wants to turn on a mobile device, you do not want to start the processor, the RAM and the storage medium separately. It would take forever. Instead, a simple facade is available to the user: a power button. Another example would be a travel website. A user do not care what is going on in the background. The system searches different airlines, hotels and prices for you while you wait. You simply enter the date and location of where you want to go, and you are represented with the result (Illustrated in Figure 15). To explain it in a simple fashion: The pattern takes many different and complex parts of a system, and presents it in a simple manner to the user [7].

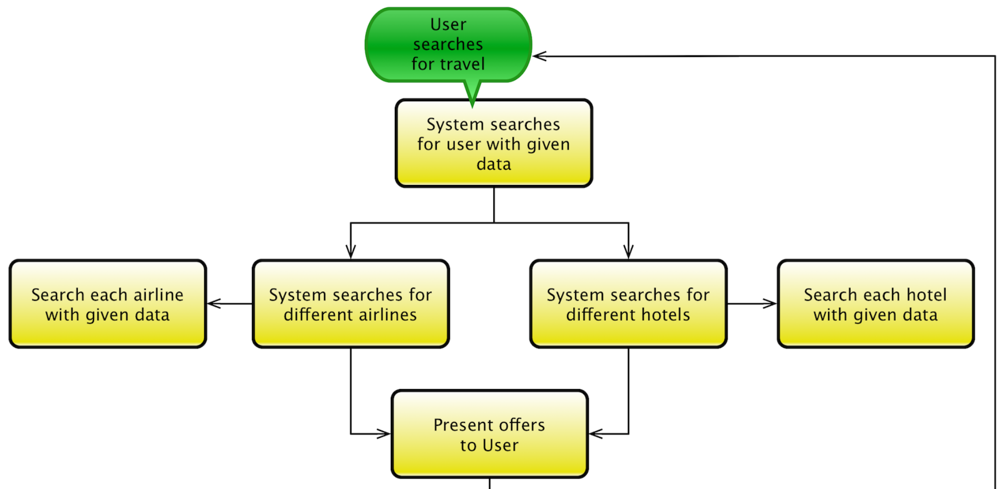


Figure 15: Facade Pattern Example

In our case, we are for instance using Facade for the Administrator page where they can do CRUD operations on poles. Instead of an administrator doing work directly in phpMyAdmin to do all the queries manually, we created different pages on the website for the administrator. For example, one page presents the administrator with all the existing poles. It also provides the possibility to do CRUD operations in a single form. By implementing the facade pattern, we

make sure the administrator do not ruin the different tables in the database by doing operations directly in phpMyAdmin. It also simplifies the process of CRUD operations on poles.

We also implemented the facade pattern in our database abstraction layer. Since we only will be developing the Android application, someone else might be developing the iOS application. Hence we have to develop a clean and simple wrapper for all of the database functionality, to handle the data processing between the application and the web server. This will also be a security measure, because we do not want the application to have direct interaction with the database.

Because of the cross-platform functionality, we have to use a data format which works regardless platform. By using JSON objects as our data format, we make sure that both Android and iOS can use our interface. It comes down to the different applications how the provided user data from the interface are displayed. Both platforms have to use the correct format to submit data through the interface. The format will be elaborated in the abstraction layer reference/documentation. This way, we provide a simple presentation to other developers which need to access the data. By documenting the abstraction layer we also help whomever is going to handle the feature development. This provides both the iOS developer, us and the feature developers with a clean facade to the database.

5.2.2 Database

Design

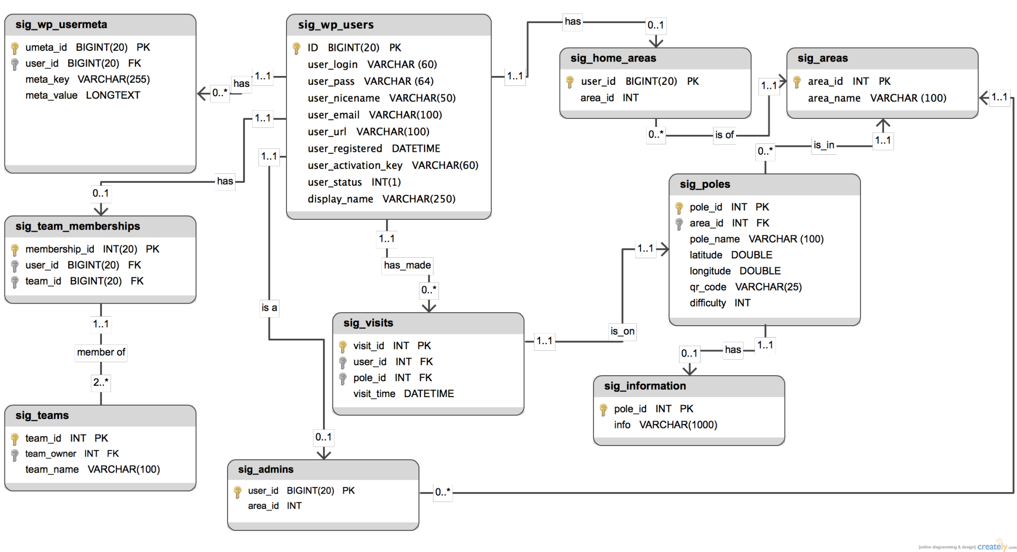


Figure 16: Database Design

Figure 16 elaborates the database design. The tables sig_wp_users and sig_wp_usermeta comes with WordPress, the remaining is designed by us. In this chapter, we will discuss this solution, and explain why we have done it this way.

When a user creates a profile, a new row in sig_wp_users is created. One user will also get several rows in the sig_wp_usermeta-table. Usermeta is attributes for a user with description in “meta_key” and the actual attribute in “meta_value”. Some of the attributes is only interesting for WordPress, but some of them are also interesting for us. The attribute we will use from this table is meta_key = “sig_wp_user_level”. The meta_value of this row is an integer from 0 to 10, where 0 is a “standard user” without administrator privileges, and 10 is the superadmin with all rights.

If a user visits a pole on the map, a row is inserted in the table sig_visits. The visit has a pole_id, which represents a sig_poles.pole_id. A pole can be placed in one out of many different areas.

A user can also be part of one team, if so, a row in the table sig_team_memberships is created. If a user creates a new team, a row in sig_teams will be inserted, as well as a row in sig_team_memberships. In sig_teams, the row “team_owner” will contain the (sig_wp_users.ID), which represents the administrator of the team.

5.2.3 Abstraction layer

The abstraction layer is the layer with the database on the server on the "inside", and the applications and website on the "outside". When implementing an abstraction layer, there is a lot of concerns. The top priorities are security and integrity.

One of the most basic concerns in the abstraction layer is SQL-injections. A SQL-injection is when a user can enter malicious SQL-statements into a input field, i.e. the username-field. If the query for instance is "SELECT FROM sig_wp_users WHERE ID = $user_id", the user could write "1; DROP TABLE sig_wp_users" into the input-field. This would delete the entire user-table, which would affect all users. Using PDO with prepared statements ensures no SQL-injections. [8] The PDO-implementation is split out in several files. This was done to keep the database password safe from other developers (the original intention was to include a external development team to create iOS-application). The class that connects to the database, 'PDO_CONNECT' (Code Example 5.1) is placed in a separate file 'db_connect.php'. It returns a instance of a PDO connected to the database, giving the programmer access to the database without revealing the password.

Code Example 5.1 PDO_CONNECT Class

```
1  class PDO_CONNECT {
2
3    private static $instance = null;
4
5    // Gets a connected PDO
6    public static function get() {
7      if(self::$instance == null) {
8        // Connect to the file containing database-credentials
9        require_once __DIR__ . '/db_config.php';
10       $instance_name="mysql:host=localhost;dbname=".DB_DATABASE."; charset=utf8";
11
12       try {
13         self::$instance = new PDO($instance_name, DB_USER, DB_PASSWORD);
14       } catch(PDOException $e) { throw $e; }
15     }
16     return self::$instance;
17   }
18 }
```

To get access to the database through the PDO_CONNECT, programmers can write PDO_CONNECT::get(), which acquires an instance ready to use. Further, the programmer can create a prepared statement, shown in Code Example 5.2.

Code Example 5.2 Prepared Statements with PDO Class

```
function get_all_areas() {

    // Prepare statement to get areas with boundaries
    $prepared_statement = PDO_CONNECT::get()->prepare("
      SELECT sig_areas.*,
             sig_bounds.bound_north,
             sig_bounds.bound_south,
             sig_bounds.bound_east,
             sig_bounds.bound_west
      FROM sig_areas
      JOIN sig_bounds
      ON sig_areas.area_id = sig_bounds.area_id
    ");

    // Execute the statement
    $prepared_statement->execute();

    // Fetch an array indexed by column name
    $result = $prepared_statement->fetchAll(PDO::FETCH_ASSOC);

    return $result;
  }
```

All functions as the one in Code Example 5.2 is contained in the class DB. The return-array from this particular function could for instance contain the data in Code Example 5.3. The web site uses this class directly, but the mobile applications cannot connect directly to the DB-class. To solve this, we have created several php-files which connects to the DB class and processes the result on server. The php-file that uses the code in Code Example 5.2 only transforms the returned array from 5.3 into a JSON-formatted (as in Code Example 5.4 array, using "print json_encode(DB::get_all_areas());. A more complicated example with more processing on server is shown in Section 5.3

Code Example 5.3 Returned Array

```
1  // Returned array from get\_all\_areas using return \$result:
2  Array (
3    [0] => Array (
4      ["area_id"] => 1
5      ["area_name"] => Gjovik
6      ["bound_north"] => 60.825
7      ["bound_south"] => 60.785
8      ["bound_east"] => 10.705
9      ["bound_west"] => 10.635 )
10   [1] => Array (
11     ["area_id"] => 2
12     ["area_name"] => Raufoss
13     ["bound_north"] => 60.735
14     ["bound_south"] => 60.713
15     ["bound_east"] => 10.615
16     ["bound_west"] => 10.5875 )
17 )
```

Code Example 5.4 Returned JSON-Array

```
1  // Returned json-array from get\_all\_areas using return json\_encode(\$result):
2    {
3        "area_id":"1",
4        "area_name":"Gjovik",
5        "bound_north":"60.825",
6        "bound_south":"60.785",
7        "bound_east":"10.705",
8        "bound_west":"10.635"
9    },
10   // Next area here
```

5.2.4 Website

Integrating Wordpress

During the first meeting with Bjørn Godager (Meeting D.1) we agreed on using Wordpress as the basis for the website. This meant that we did not have to develop a Content Management System from scratch, which allowed us to focus on the main features of the website instead.

The feature list was established in the Requirement Specification. By extracting the features of the website listed in table below, we established which features to develop first.

ID	Name <action> <result> <object>
2	Display registration form on website to a website user
4	Display CRUD options for family members to a website user
5	Display CRUD options for poles to area administrator
6	Display CRUD team options to website user
11	Display form to register pole visits to website user
12	Store pole visits data provided by website user

The first feature allows the a End-User to register using the website. We did not want any of the End-Users to be able to login using the Wordpress Dashboard. The reason being that the Wordpress Dashboard is meant for the administrators, ant not the End-User. We then had to figure out how to allow users to register without using the standard Wordpress registration form. After trying different solutions on how to implement user registration, we found that the Buddypress plugin was the best alternative. Buddypress is a social network plugin for Wordpress, which allows Wordpress websites to enable social networking. There are a lot of features in Buddypress we were not interested in, but there was also some features we wanted, such as a registration form (Figure 17).

Figure 17: Buddypress Registration Form

Buddypress allows us to present a separate registration form, and as a bonus it allows us to add User profile functionality to the website (Figure 18. User profiles means that the End-User can edit their own profile without having to access the Wordpress Dashboard. This was exactly what we wanted to accomplish. The user can add a profile picture/avatar which can be displayed in the application along with on the website (only visible to the user itself).

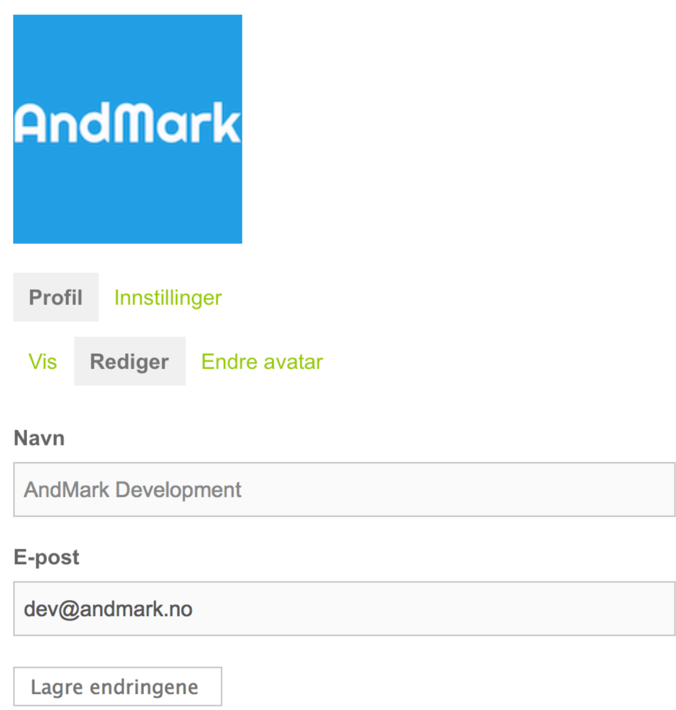


Figure 18: Buddypress User Profile

Another plugin we added is called "Remove dashboard access from non-admins", which does exactly what is is called. It makes sure only area administrators can access the Wordpress Dashboard. The End-User (non-admins) gets redirected to first page of the website if they try to access the Dashboard. That plugin along with Buddypress ensures that the End-User stays out of the Dashboard, but are able to edit and control their own profile. We also added the plugin "Exec-PHP", which allows Wordpress Pages to execute PHP code. This allows us to implement our own php functions to the website. This plugin was useful, since almost all of our functionality are executed in different Wordpress Pages. There were some other alternatives, but "Exec-PHP" allowed us to encapsulate the php code with normal start and finish tags *<?php code here; ?>*. A few other plugins required either:
[PHP] code here; [/PHP] or *[code=php] code here; [/code]*.

We also added a plugin called "JSON API". This plugin works as an API which allows us to download news stories, in the application as JSON format. For instance we can simply do a request to the API:

http://www.stolpejakten.no/api/get_category_posts/?slug=gjovik

Where the slug, in this example *gjovik*, gets swapped with the home area set by the user in the application.

The last plugin we added is called "Restrict Categories". This plugin allows the website administrator (us for this year's project) to limit area administrators to write posts in certain categories. Which means that we allow the administrator for area *Gjøvik* to write news stories which regards only the users in *Gjøvik*. The reason behind this is that the Customer only wanted separate news stories for all the different areas.

User Interface

Since the website is a Wordpress site, we simply had to find a theme the Customer the liked. We went with a theme called "GreenChili". It is a minimalistic theme and it responsive, which means it works on smaller screens such as mobile devices. Figures 19 and 20 shows the responsiveness of the theme on different screen sizes.

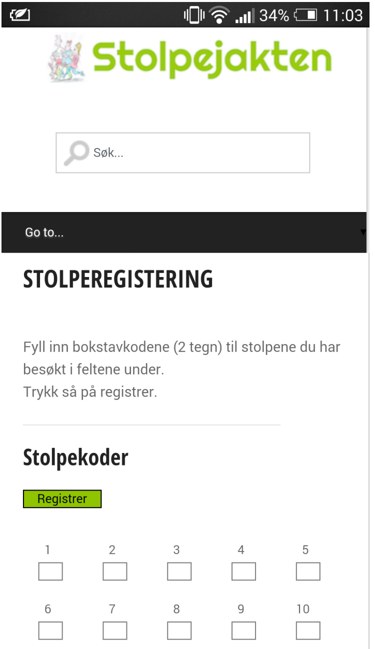


Figure 19: Register Poles on Phone

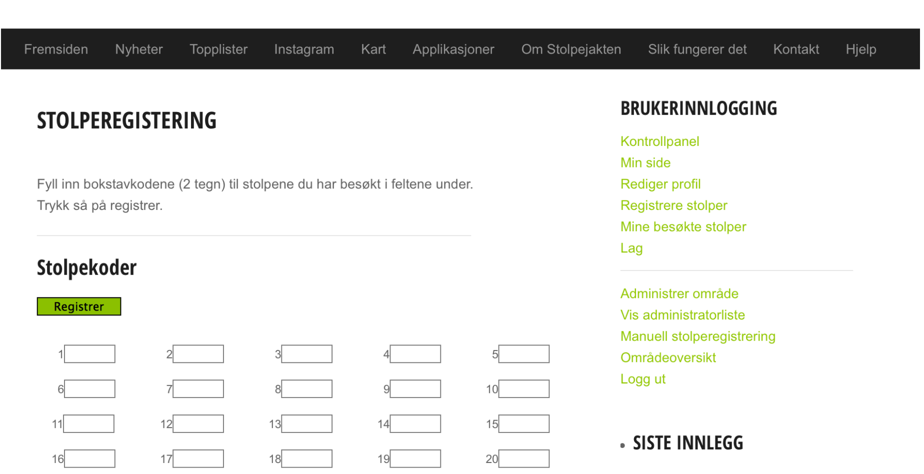


Figure 20: Register Poles on Computer

It was not enough to use a responsive theme to ensure our was content presented equally responsive. We had to implement some simple CSS style to the different pages (Code Example 5.5).

Code Example 5.5 CSS Responsive Website

```
<style>
<!-- Row width inside parent (table) -->
tr {
height:100%;
}
<!-- Entire table width -->
table {
width:100%;
}
<!-- Text field margins and width, 60% width -->
input[type=text] {
margin-right: 5px;: 5px;
width:60%;
}
</style>
```

Since we use tables to arrange the data, we had to ensure that the table width is 100% of the content area. In the first few weeks of development, we used static sizes, such as *width:300px*. This soon became an issue regarding the responsiveness of the website. It made the table too wide for some phones, and to see the content a user had to change the phone orientation to landscape. A fixed size website is outdated, since the common Internet user might not access the websites with a computer [9]. The users are accessing the website from different devices.

Figure 21 shows the different platforms the users are accessing the website. The diagram a great deal of users accessing the website using iPhone, iPad and Android. Therefor the implementation of fluid width of the website was a good idea. The alternative would be to alienate a group of users from accessing the website properly, by using the fixed width.

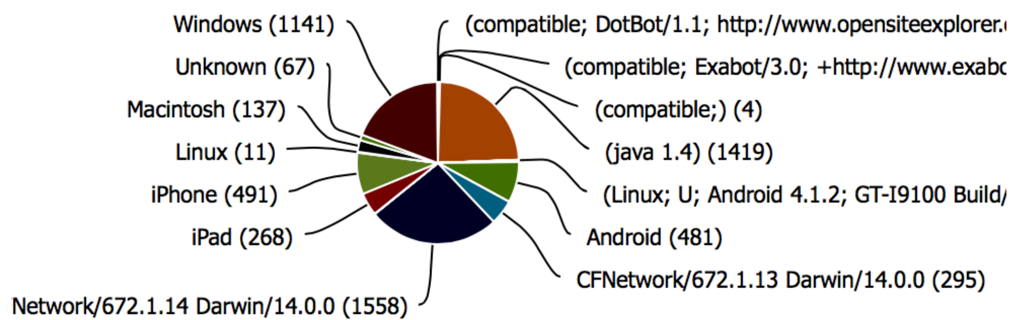


Figure 21: User Platform Statistics

Besides being responsive, GreenChili came with a few widgets we could implement in our system. What we did was to customize the theme's login widget to include permalinks to our Wordpress Pages. Another alternative would have been to create our own widget. We felt that customizing the already existing widget was the best option. The reason was that this was the easiest and the least time consuming option. The altered widget loads menu items based on if you are an End-User (Figure 22) and/or an Administrator (Figure 23).

BRUKERINNLOGGING

Min side
Rediger profil
Registrere stolper
Mine besøkte stolper
Lag
Logg ut

Figure 22: End-User Website Menu

BRUKERINNLOGGING

Kontrollpanel
Min side
Rediger profil
Registrere stolper
Mine besøkte stolper
Lag

Administrer område
Vis administratorliste
Manuell stolperegistrering
Logg ut

Figure 23: Administrator Website Menu

An Administrator needs to be able to check their visits, edit their profile and use all of the other features equal to an End-User. The menu loads everything under the line in Figure 23, if an Administrator is logged in. If the logged in user is an End-User, only the menu above the line is displayed. We decided to divide the menu as we did, because it separates the different User-levels in a simple fashion. Which meant that we have to execute one line of code (if statement), whether or not to display the Administrator options of the menu below the End-User menu.

Gjø-Vard Orienteering was pleased with the final look of the website (Figure 24). The website has a top bar menu, which allows simple navigation. The Customer and we felt implementation of a simple navigation menu was necessary, because according to the Customer, the technical skills of the users varies.

Figure 24: First Page with Logged in User

User levels

An issue we had to resolve was how to separate users from area administrators. Initially we tested the native Wordpress user levels, such as Subscriber and Administrator. For instance a Subscriber can not delete posts, but an Administrator can. Since area administrators will manage poles, and write news stories for their area as well, we decided to take advantage of the Wordpress feature and integrate it with our own code.

We had to combine our own sig_admins table in the database, and the native Wordpress user levels. The reason being that Wordpress uses its own permissions to allow/deny users access to the Dashboard. Which meant we could not exclude the user levels completely. In our case, the only reason an area administrator should access the Dashboard is to manage news stories. To check if a user is a Administrator (can manage poles), we created the simple method *sig_is_admin()*. The method returns a boolean if a user is administrator or not, and the content of the administrator page is loaded there after.

To ensure that a user is registered as a Subscriber, we used the Buddypress registration form with a specific role (user level): *[Register role="subscribers"]*. Which means when a user registers, it gets registered as a Subscriber. Subscriber the lowest user level besides "None". In our case, Subscriber and "None" has the same limitations.

By integrating our own code into Wordpress pages, we ensure that the content is loaded based on the correct user level. Since an area only can have one administrator (in our sig_admins) to manage poles, we had to implement it in this manor. Because an area can have multiple Authors (Wordpress User Level for managing news stories), but only one "Pole Administrator". The reason being that we decided to only let only one person at a time have access to the poles, which means less people to do something wrong.

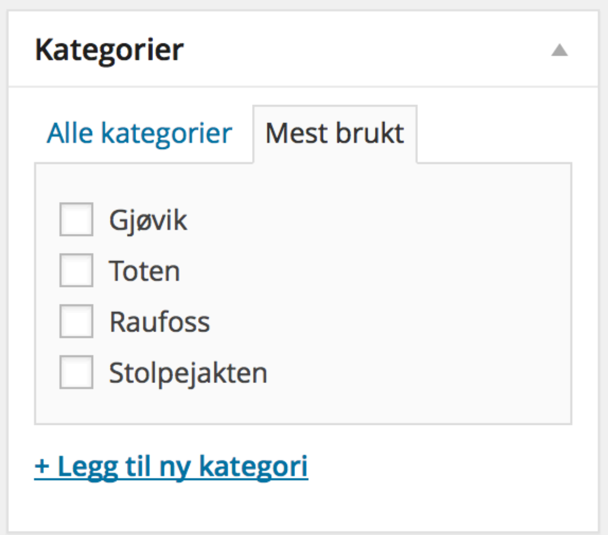


Figure 25: News Story Categories

By combining the different user levels, our own sig_admins table and the "Restrict Categories" plugin we separate the users from the administrators, and the area administrator from an area author. Since we use the "Restrict Categories" plugin, we can allow Authors to only post news stories in their own category (Figure 25). A Category, in this context, is equal to an Area.

Administrator functionality

A website administrator have to be able to do CRUD options for poles in their area. We had to create a Wordpress Page which only an administrator could access. The purpose of the page is to display the area's poles. Since each area will have its own administrator, we had to create a functionality which loads correct the area for each separate administrator. Figure 26 represents how it looks on the website for the Gjøvik administrator .

Stolper Gjøvik

ID	NAVN	LAT	LONG	QR	DIFF	
1	Totens Sparebank	60.795243	10.690124	001BE	1	Slett
2	Toten Treningssenter	60.794141	10.687504	002BC	1	Slett
3	Gamletorvet	60.795127	10.687375	003BA	1	Slett
4	Nordbohus	60.784225	10.699193	004AG	1	Slett
5	Kiwi	60.785285	10.695526	005BK	1	Slett
6	Sentrum Installasjon	60.783901	10.690174	006BR	1	Slett
7	Gjøvik Økonomitjenester	60.784827	10.692313	007BN	1	Slett

Figure 26: Pole List on Website

We arrange the poles in a table, equal to most of the content we arrange on the website. Code Example 5.6 elaborates the implementation in a Wordpress Page. Wordpress has a method for accessing the logged in user's ID from the database. We used this ID to check in our sig_admins table after which area

Code Example 5.6 Website - Administrator Functionality

```
<?php
    <!-- Executes if the user is administrator ->
    <?php if (sig_is_admin() === true) { ?>

        <!-- Create table with all the poles in area -->
        <table width="100%" border="1">
        <tr>
        <th>ID</th>
        <th>NAVN</th>
        <th>LAT</th>
        <th>LONG</th>
        <th>QR</th>
        <th>DIFF</th>

    <?php
    echo "<th></th></tr>";

    <!-- Gets the poles in area where user is admin.
    Displays each pole as row in table. -->
     global $current_user;
     get_currentuserinfo();

    $result = DB::get_poles_for_admin($current_user->ID)
    foreach ($result as $row) { ?>

      <!-- Prints pole data ?>

        <form action="" method="post"><tr>
        <td width="5%"><?php echo $row['pole_id'];; ?> </td>
        <input type="hidden" id="pole_id" name="pole_id"
        value="<?php echo $row['pole_id']; ?>">

        <?php
        echo '<td width=55%>'. $row['pole_name'] . '</td>';
        echo '<td width=15%>'. $row['pole_latitude'] . '</td>';
        echo '<td width=15%>'. $row['pole_longitude'] . '</td>';
        echo '<td width=10%>'. $row['pole_qr_code'] . '</td>';
        echo '<td width=5%>'. $row['pole_difficulty'] . '</td>';
        echo '<td width=5%><input type=submit
                    name=submit value=Slett></td></tr>';
        </form>
    <?php
    }
} else {
    echo "Du har ikke tilgang til denne siden!";
}
?>
```

User functionality

The feature list contained more features we had to implement, such as family members. We had two options when deciding on how to implement this feature. One alternative was to do it the same fashion as Netflix. Netflix asks *"Who?s watching?"*, then you can select an account member. Which for us would translate into *"Who is searching for poles?"*. The issue we had with this, is that it had to be one device available for each account member. This was contradictory to what Gjø-Vard Orienteering requested. Because the issue last summer was that not all family members searching for poles, had their own device (such as grandparents and young children). Therefor we went with the second option.

Legg til familiemedlemmer

Navn på medlem	Fødselsdato (DD-MM-YYYY)

Legg til

Figure 27: Family Members Functionality

When a user has registered an account, one can add family members on the website. The user becomes "parent" of the registered family member(s). Figure 27 presents how it looks on the website. Which means when a user goes searching for poles, the "parent" can select the family member(s) who are with them on a pole search. This ensures that the requirement from Gjø-Vard Orienteering is implemented as requested. A user can now include family members in the pole search, and only use one device to get all of the visits registered for each family member. Since Gjø-Vard Orienteering wanted to include as many users as possible, without technology prevent user participation.

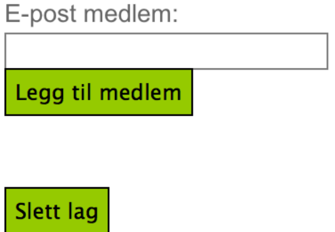
Gjø-Vard Orienteering might host some competitions such as "most active work place" or "most active school class", therefor they requested team functionality. They also wanted to have a limit of maximum members of a team. The reason behind it was it would be easier for them to calculate scores and statistics. The requested size was max four members.

We first considered the group functionality included in Buddypress, which allows users to create groups. The only main difference between a group and a team, in this context, is the name. The issue with Buddypress's group functionality, was that maximum members in a group was unsupported. The group functionally could be altered in such a fashion that it only allowed a user to member of one group. This was inadequate to meet the requirements set by Gjø-Vard Orienteering.

Instead we decided to create our own team functionality. Figure 28 is an example on how a team is composed in the team administrator's page.

Team AndMark

Navn	E-post	Stolper	
Administrator	anders@andmark.no	2	Fjern
Anders Hagebakken	anders@hagebakken.no	11	Fjern
Markus Brovold	markus.brovold@gmail.com	12	Fjern



Slett lag

Figure 28: Team Functionality

The team creator becomes the administrator of that team. An administrator has the option to add and remove a team member. While the team member has the option to leave their current team. We also decided to add a column which displays the team member's number of visited poles. Which in a competition might be useful for teams to see who is lagging behind so the team can ensure maximum effort. Our implementation of teams satisfied the requirements set by Gjø-Vard Orienteering.

Finally we had to implement the feature which allows a user to register their visits on the website. The initial implementation consisted of a table which contained a textfield for each pole in a given area below each other. It turned out to be inconvenient, because one had to scroll immensely to reach the submit button at the bottom.

To resolve this inconvenient issue we decided to re-design the layout. We started out with the same table as the base. Then we created a for loop, which generates a new row in the table whenever the loop reaches five columns. The row contains five textfields with a corresponding pole ID (Figure 20). This implementation ensures that the user always have to write the correct pole QR code in the corresponding pole ID textfield.

Code Example 5.7 Website - Register Poles Table

```
1  <form action="" method="post">
2  <?php $result = DB::get_poles($user_id); ?>
3  <table border="1"><tr>
4
5  <?php for ($i = $init; $i <= count($result); $i++) { ?>
6  <td><label><?php echo $i; ?></label>
7
8  <input type="text" id="pole_qr" size="6"
9   name="<?php echo "pole_qr_codes[".$i."][".$i."]"; ?>" value="">
10 </td>
11
12 <!-- Break row at 5 columns -->
13 <?php if (($i%5) === 0) {  echo "</tr>"; } };?>
14 <input type="submit" name="submit" value="Register">
15 </form>
16 </table>
```

Code Example 5.7 represents our implementation simplified. We implemented a check if the content of *pole_qr_codes[][]* matches a pole in the database, then store the visit if the submitted combination is correct. Otherwise we present an error message to the user, informing the user that the submitted data was incorrect.

5.3. Android Application

5.3.1 User Interface

When creating the main layout of the application, we had a couple of alternatives for the in-app navigation. We could either create a home screen with buttons arranged in a GridLayout (Figure 29), a list with buttons (Figure 30), or by adding icons to the ActionBar (Figure 31). The advantage of using a home screen with buttons (either in a GridLayout or in a simple list of buttons), is that each button gets more space for text. This way it is easier to explain in detail what action a click would initiate. When using the ActionBar, only a icon will be visible to the user. The advantages of using the ActionBar is that the user would not have to go back to the main screen to change activity. If the user for instance wants to go directly from the map to the settings, the user can press the settings-icon on the ActionBar.

Because Android has standardized all common buttons. The Official Android Developer Design Guide states that "Pictures are faster than words", and enchants developers to use the ActionBar instead of buttons for changing Activities. [10]. They further state that an applications core functionality should always be available from the ActionBar. Therefore, we choose to add icons for MainActivity, MapActivity, UserActivity and UserSettingsActivity on the ActionBar. In Figure 31 the application is in MainActivity, showing the icons for Map, User and Settings in the action bar.

Furthermore, Google suggests to make the application stand out from others using your brands color in the ActionBar. Note the difference in Figure 30 and 31. The green color (Hex triplet color 99CC00), also appears in the Launcher Icon and in a splash screen shown when the application is launching.

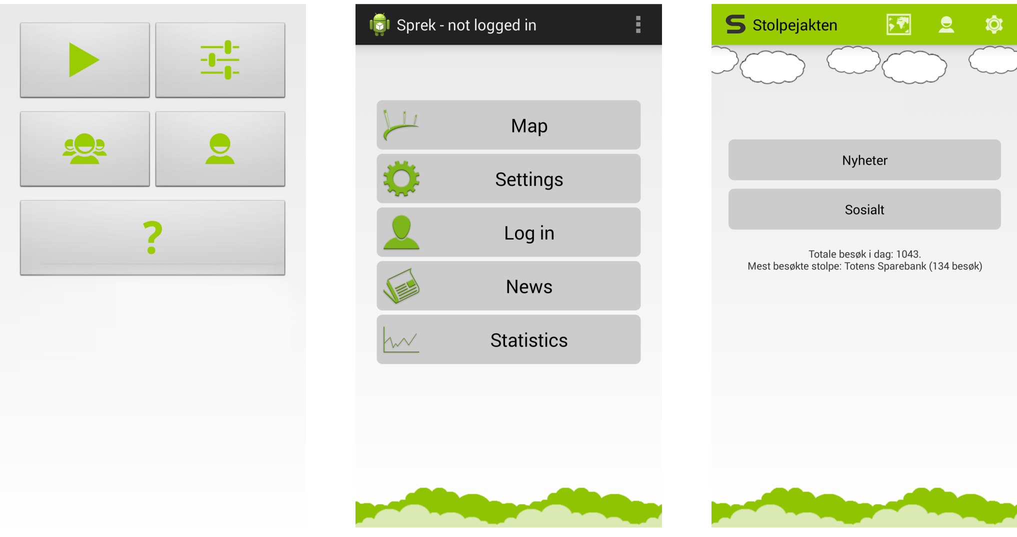


Figure 29: Grid Menu

Figure 30: List Menu

Figure 31: ActionBar Menu

5.3.2 User management

The core functionality of the application, such as visiting poles and viewing statistics, depends on the user being logged in. When deciding how to implement the log in-feature, there was a couple of options and matters to consider. When logging in, the users submits a username and a password. Our task is to ensure that the data is handled properly to minimize the risk of leaking user-passwords. There are four steps in the login-process. The first is the user typing the credentials into the form in the application. The second is the application sending the credentials to to server for verification, the third is the server verifying and processing it, and the fourth is the server telling the application wether it is successful or not.

The first step is done straight-forward, by creating a layout which contains EditText-fields for username and password. The password-field is declared as a password-field, making the input censored with asterisks for each character.

The second step is done through the FetchDataTask from Chapter 5.3.7, which transfers the data through a HTTPS-connection to a php-file on the server.

The third step is the server receiving the data from the HTTPS-connection. The processing on the server is explained in Code Example 5.8.

The fourth step is explained in Code Example 5.9. The "Url.LOGIN" is the link to the php-file on the server (Code Example 5.8).

Code Example 5.8 Login.php

```
<?php
  header('Content-Type: application/json; charset=utf-8', true,200);
  // Include the DB class containing prepared statements
  require_once("./pdo.php");

  // Include the config file to make direct use of the database.
  require_once("./wp-config.php");
  // Include the PasswordHass class to check if the passwords are matching.
  include_once("./wp-includes/class-phpass.php");

  // Get arguments from application
  $username = $_POST['username'];
  $password = $_POST['password'];

  // Get user for username
  $user = DB::get_user($username);

  // Encode and decode json_array to access as array
  $json_user = json_encode($user);
  $json_decoded_user = json_decode($json_user, true);

  // Get relevant data from query result array
  foreach ($json_decoded_user as $arr) {
    // Get the hashed password and the user id from the database
    $password_hashed = $arr['user_pass'];
    $user_id = $arr['ID'];
  }

  // Use WordPress and PHPASS to create a hash of the typed password
  $wp_hasher = new PasswordHash(8, TRUE);

  // Verifies the hashes, returns '1' if the typed password is correct, else '0'
  $logged_in = ($wp_hasher->CheckPassword($password, $password_hashed));

  if($logged_in == true) {
    // Send the userid back to application
    print json_encode($user);
  } else {
    // Send userid = 0 bak to application. Not logged in.
    $jsonString = '[{ "ID" : "0" }]';
    print $jsonString;
  }
?>
```

Code Example 5.9 Login Method from Android Application

```
private void login() {

  String username = m_UsernameEditText.getText().toString();
  String password = m_PasswordEditText.getText().toString();

  // Only try to sign in if credentials are given
  if (!username.matches("") && !password.matches("")) {

    // Create ASyncTask to execute PHP on server:
    FetchDataTask connection = new FetchDataTask(LoginActivity.this);

    // Add attributes
    connection.addValuePair(new BasicNameValuePair("username", username));
    connection.addValuePair(new BasicNameValuePair("password", password));

    //Execute query to verify credentials
    String result;
    try {
      result = connection.execute(Url.LOGIN).get();
      // Get the object from the result-string:
      JSONArray jArray = new JSONArray(result);
      JSONObject jsonData = jArray.getJSONObject(0);

      int userId = jsonData.getInt("ID");
      if (userId != 0) {
        // Set the userId value of the singleton-instance of User.
        User.getInstance(getApplicationContext()).setValues(userId, this);
        finish(); // Disables possibility to go back to this activity
        startActivity(new Intent(LoginActivity.this, MainActivity.class));
      } else {
        Toast.makeText(this,
            getResources().getString(R.string.wrong_username_password),
    Toast.LENGTH_LONG).show();
      }

    } catch (InterruptedException e) {
    } catch (ExecutionException e) {
    } catch (JSONException e) {
    }

  } else {
    // Missing either username or password.
    Toast.makeText(this,
        getResources().getText(R.string.enter_values),
    Toast.LENGTH_SHORT).show();
  }
}
```

5.3.3 Map

Choosing map technology

We had a meeting with our supervisor before we started the project planning, where we discussed the different options for implementing the map. Our initial plan was simply to implement either OpenStreetMap or Google Maps, and our supervisor agreed. We found that these maps were adequate for displaying poles in the terrain to a user. The reason being that a normal user does not care whether or not the map is extremely detailed. Therefor we thought we could convince the customer that either OpenStreetMap or Google Maps were enough. Both of these services are simple to implement in Android. We wanted a simple map because we wanted focus on the other features of the application.

After we finished the project plan, started the next iteration. The last day of the iteration we had a status meeting with our customer. We showed them we had done so far and told them what we were doing the next sprint. We talked about the map and our ideas and thoughts, about the detail level of the map and the everyday user's ability to read maps. The customer was quite clear that they had to have an orienteering map, and nothing else. They wanted a link to orienteering. Because the customer is an orienteering club, we understand that they want an orienteering map, but we told them the everyday user (such as us) do not care.

The map's detail level were the customer's concern, not the user's. Since one of the main goals for the project is to get people off the sofa and outside and not teaching them orienteering, we did not see the importance of a very detailed map. We still wanted to please our customer and told them, a bit reluctantly, that we would fix it. They were pleased. We then got a beta map we could use to test with.

The map from the customer was created in a software called OCAD, which is used for drawing orienteering maps. Our challenge was to create a functional map with GPS location and functionality to add poles. We started looking for an Android library which supports OCAD, but we only found an OCAD application in the Google Play Store. This application is written by OCAD

themselves. This did not help us much. Luckily OCAD have the possibility to export to other formats, such as KMZ/KML (Overlay for Google Maps). We found a library called ArcGIS which could read simple KML files and add them as a ground overlay to a map. Since the file we got from the customer contained thousands of polygons and polylines, it would not be able to be processed on a normal handheld device anyway. The device would quickly run out of memory trying to create the map.

Then we looked at other OCAD export formats, such as EPS, PDF and Shape-files. After searching for libraries to handle geo-referenced PDF and EPS, we found a Norwegian library from Norkart which can use something called MBTiles. MBTiles is MapBox Tiles. We then found that MapBox has a software called TileMill. TileMill can read Shape-files and add them to a MapBox map. We then exported the OCAD file to four different shape files (points, area, lines and text) and opened them in TileMill. TileMill allowed us the style our own map using CSS-like syntax.

Code Example 5.10 TileMill CSS

```
1 #layer {
2     line-color: #0AF;
3     line-opacity: 0.5;
4     line-width: 2;
5 }
```

After trying and failing with creating our own styles, we found that the problem was that the layers are not layered properly. Every line were in one layer and all the points in one layer and so on. Therefor we could not style the map in orienteering colors. We even found a GitHub repository [11] which had done the coloring before Another problem was that the API from Norkart did not read the MBTiles properly. Neither the file we created nor an example MBTiles file we found online. Therefor we had to find some other solution.

The last thing we tried was to Export the OCAD to TIFF. Because we found that Google Maps API v2 for Android has the ability to add image tiles as overlay. By implementing the TileProvider and overriding a couple of functions, we had an application which could add tiles to Google Maps. We then had to find some software which could create the tiles for us. MapTiler was the software of choice.

We had to pay 175 NOK for a version that allowed us to control the maximum and minimum zoom levels and remove the watermark. After trying and testing MapTiler, we had success loading the orienteering map as overlays. The next thing we did was to set the Google Map type to NONE, which just shows a white and gray grid in background. By choosing this solution, we get all the options in Google Maps, but with our own map. This solution works great on both high-end and low-end devices.

The last thing we did was to add functionality to limit the viewable area to our overlay. Code Example 5.11 is our solution. We set LatLng bounds (BOUNDS) created from Northeast and Southwest coordinates of the map. Then we check if the current visible region is inside the bounds, if so, it does nothing. When a user moves the camera to a valid target inside our bounds, we simply move the camera to that region based on x/y coordinates.

Code Example 5.11 Visible Bounds

```
1  private void limitVisibleMapRegion() {
2
3      if (BOUNDS.contains(googleMap.getCameraPosition().target)) {
4        return;
5      }
6
7      double x = googleMap.getCameraPosition().target.longitude;
8      double y = googleMap.getCameraPosition().target.latitude;
9
10     double maxX = BOUNDS.northeast.longitude;
11     double maxY = BOUNDS.northeast.latitude;
12     double minX = BOUNDS.southwest.longitude;
13     double minY = BOUNDS.southwest.latitude;
14
15     if (x < minX) {
16       x = minX;
17     }
18     if (x > maxX) {
19       x = maxX;
20     }
21     if (y < minY) {
22       y = minY;
23     }
24     if (y > maxY) {
25       y = maxY;
26     }
27     googleMap.moveCamera(CameraUpdateFactory.new LatLng(new LatLng(y, x)));
28   }
```

Legend

After the customer showed us the paper map from last summer, we noticed that the map had a legend table. We thought that this was a cool and nice-to-have feature. From the customer we got a single image with all of the symbols. We implemented this in a simple scrollable view, and it looked quite bad, since it was a single image. We then decided to contact the person that created last year's map brochure to get a hold of the separate images. We got sixty eight images named after the symbol it represents. The challenge was to get these images with the corresponding text into a ListView.

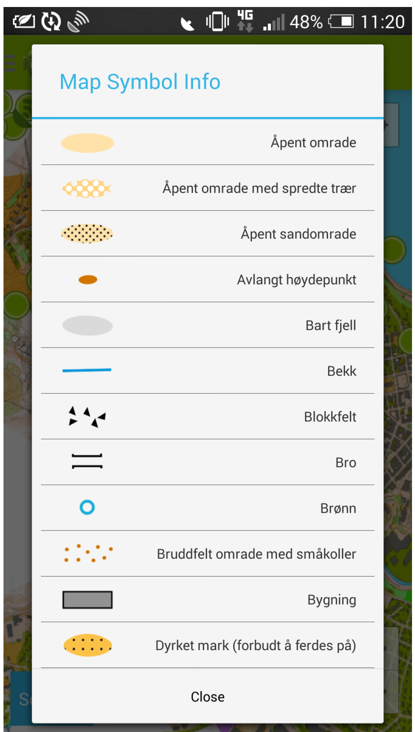


Figure 32: Legend

The first thing we did was to create a custom layout for a row. This row contains an ImageView and a TextView located next to each other. The finished product can be seen in in figure 32. Code Example 5.12 shows how we created a pre-defined array with drawable resource IDs, and a pre-defined string-array populated with pre-defined strings from strings.xml. This allows us to easily translate the legend to other languages.

Code Example 5.12 Legend

```
<resources>
<!-- Array of drawable resources -->
    <array name="info_images">
        <item>@drawable/info_open_area</item>
        <item>@drawable/info_open_area_with_scattered_trees</item>
        <!-- 66 more items ... -->
   </array>

    <!--Array of string resources -->
    <string-array name="info_images_text">
        <item>@string/info_open_area</item>
        <item>@string/info_open_area_with_scattered_trees</item>
        <!-- 66 more items ... -->
    </string-array>
</resources>

<!-- Then we have the corresponding strings defined in strings -->
<string name="info_open_area">Open area</string>
<string name="info_open_area_with_scattered_trees">Open area with
scattered trees</string>
...
```

Pole altitude

After a status meeting with the Customer and Supervisor, we discussed the features of the application. A requested feature was to get the altitude (meters above sea level) of the pole. After searching the Internet for solutions, we found the Google Elevation API. One can simply provide a location using latitude and longitude and get a calculated altitude in JSON format. The downside of this API, is the daily quota. Every time a user downloads new poles, the API is accessed. Which means the daily quota will get exceeded quite fast. To ensure that the quota was never exceeded, we had to store it in the server database before the users could use the application. We created a script which loads all the pole's locations from the database, then create a long query which we send as a request to the API (Code Example 5.13).

All the altitudes are stored into the poles table in a separate column. Which means when we download the poles in the application, we get the altitude along with everything else. Regarding the credibility of the altitudes from the API, we have Mjøsa as a reference point. Which we know has the altitude of 123 meters above sea level [12]. There is a pole located by the bank of the lake, and the altitude from the API is 123 meters above sea level. The screenshot (Figure 33) of the application shows the accuracy of the for mentioned pole.

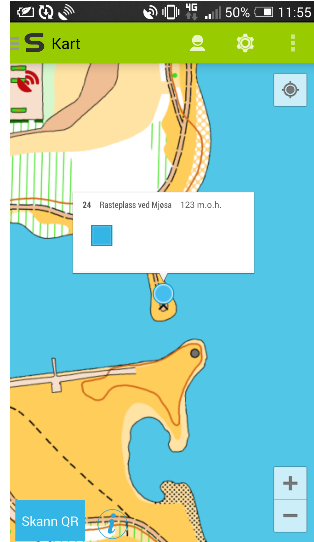


Figure 33: Pole Altitude Example

Code Example 5.13 Pole Altitude

```
<?php
  $all_poles = DB::get_poles();

  // Create the start of the request with the URL
  $url = "https://maps.googleapis.com/maps/api/elevation/json?locations=";
  $request = $url;
  $comma = ",";
  $seperator = "%7C";
  $end = "&sensor=false&key=API_KEY";

  $counter = 0;

  $pole_ids = array();
  // Loop through all the poles
  foreach ($all_poles as $pole) {

    // Stores the pole ids in an array for later use
    pole_ids[] = $pole['pole_id'];

    // Append latitude, then comma, then longitude.
    $request .= $pole['pole_latitude'];
    $request .= $comma;
    $request .= $pole['pole_longitude'];

    // If the counter is at the last request, add the separator.
    if ($counter < count($all_poles)-1) {
      $request .= $seperator;
    }
    $counter++;
  }

  // Add the last part of the request
  $request .= $end;

  // Get JSON data from the request result
  $jsonData = json_decode(file_get_contents($request), true);

  $qrsCounter = 0;

  // Adds each altitude to the pole database
  // using elements in the pole ids array
  foreach($jsonData['results'] as $key => $eval) {
    $ev = round($eval['elevation'], 0);
    $id = $pole_ids[$qrsCounter++];
    DB::update_altitude($id, $ev);
  }
?>
```

5.3.4 Sensor Usage

Camera

Since the application needs to scan the QR codes located on the poles, we had to implement a scanner library. In a previous project, we had some success implementing the “ZBar Scanner” library. This library must be started using “startActivityForResult”, then when the library is done scanning, it returns the result to the overridden function called “onActivityResult”, in the activity which launched the scanner library using the “startActivityForResult”. We had some issues with this, because the library did not open / release the camera properly, so it caused to application to crash. Therefor we discarded this library for this project.

Another option was the ZXing library. We found that this library was too advanced for our purpose, and we felt it was like cracking a nut with a sledgehammer. We needed a simpler library that just read a simple QR code. While looking for alternatives to ZBar, we found that the developer had stopped maintaining the project, and had moved to another. This project was called Bar Code Scanner We tested this library with great success. It opens and releases the camera properly, and we have yet to experience the application crashing because of it.

GPS

We decided to implement a graphical strength indicator in the map layout. The reason behind it was whether or not a user could trust the GPS location. At least now, the user know how good the signal is and can trust the GPS thereafter. The Figure (34) shows the three different symbols we used to illustrate the accuracy. Green is good, orange is medium and red is weak.

Figure 34: GPS Accuracy Symbols

Code Example 5.14 explains the simple algorithm we used to display the accuracy. We created a simple class that implements the LocationListener (native Android). The rest is explain the the code comments.

Code Example 5.14 GPS Accuracy Indicator

```
private class GPSLocationIndicator implements LocationListener {
    // Stored the accuracy
    private int oldAccuracy;
    private static final int GOOD_SIGNAL = 10;
    private static final int WEAK_SIGNAL = 20;
}

// Everytime the 'users GPS location changes, this method is called.
public void onLocationChanged(Location location) {

// Get the accuracy in meters, rounds to closest meter.
int newAccuracy = Math.round(lastLocation.getAccuracy());

    // If the old accurcy is different from the new location, update the accuracy.
    if (oldAccuracy != newAccuracy) {
    oldAccuracy = newAccuracy;

    // 10 meters or less accuracy — Good signal
    if (oldAccuracy <= GOOD_SIGNAL) {
      m_GPSIndicator.setImageResource(R.drawable.gps_good);
    }

    // Between 10 and 20 meters accuracy — Medium signal
    if (oldAccuracy > GOOD_SIGNAL && oldAccuracy <= WEAK_SIGNAL) {
      m_GPSIndicator.setImageResource(R.drawable.gps_medium);
    }

    // Over 20 meters accuracy — Weak signal
    if (oldAccuracy > WEAK_SIGNAL) {
    m_GPSIndicator.setImageResource(R.drawable.gps_weak);
    }
  }
}
```

5.3.5 User Data Storage

SQLite

One of many challenges with the application is that it needs to handle a user not having Internet access. Because of this we need to store most of the data on the device. We decided to go with SQLite, because it is very similar to MySQL which we are familiar with and Android has great support for SQLite. The other option would be to use a proprietary flat file to read and write from. The disadvantages with a flat file is that one needs to load the entire file each time we need one particular file. Using a SQLite allows us to preform a query to a table to get exactly what we want. With a flat file one needs to get all the file content, then selecting data from that content. It is also much simpler to delete and update a row in a SQLite table, rather than loading a file, removing / update that line, re-writing the entire file and storing it to the device.

A major functionality of the application is to register a pole visit after you scan the code. Hence we need to support pole visits if a user has no Internet access. We divided to go with a SQLite table which stores the pole_qr_code (which is unique in the SIG_POLES table), and a timestamp when the user scanned the QR code. We also added a column which has a uploaded flag, which gets set to '1' if it successfully got uploaded, and '0' if not. Then the next time a user scans a code, it will try to upload every visit in the table which has the '0' uploaded flag. The flow chart (Figure 35) shows the order of execution.

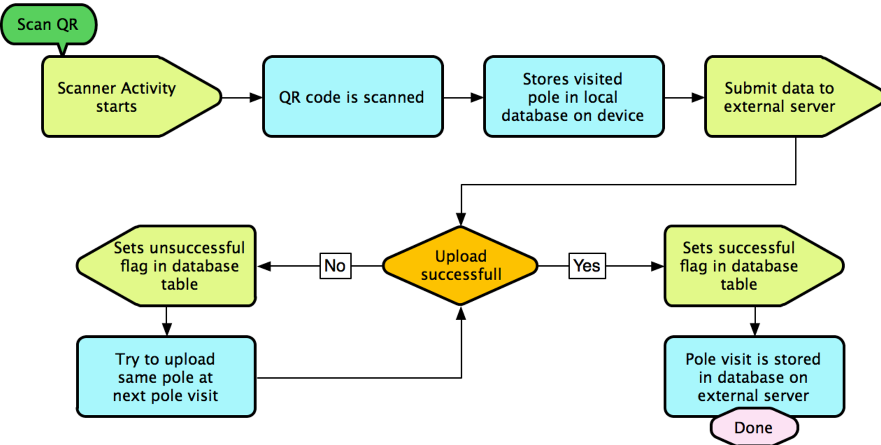


Figure 35: Pole Visit Submission Flowchart

Code Example 5.15 shows a minimal example on how we implemented offline storage of visits programmatically.

Code Example 5.15 Visit Poles

```
1  // Gets the text from the QR scanner
2  public void handleResult(Result rawResult) {
3  String qr = rawResult.getText();
4
5  // Adds the visited 'poles QR code to the database
6  db.visitPole(qr);
7
8  // The database tries to upload the visits from the table
9  // If the result string length is 0, it was not uploaded.
10 // Any other string length means it was uploaded.
11 // User feedback is given thereafter.
12 String result = db.uploadPoles();
13 }
14
15 public String uploadPoles() {
16 // The uploadPoles function tries to parse the result to an integer,
17 // because we return the visit_id from the database.
18 // The uploaded flag is set to "1
19 String result ="" ;
20 try {
21 result = uploadVisitsTask.get();
22 int number = Integer.parseInt(result);
23
24 // If the parseInt throws a NumberFormatException,
25 // we know it was not uploaded, and we return a empty result string
26 // Then we uploaded flag is set to "0
27
28 } catch (NumberFormatException e) {
29       result = "";
30 }
31
32 return result;
33 }
```

We also had to store all the pole data in a SQLite table locally on the device. By using our simple abstraction layer, we preform simple request tasks which downloads the data as JSON, then we parse the JSON data on the device. Code Example 5.15 shows our create table query. This query is the same for the SQLite database on the device and the MySQL database on the server.

Code Example 5.16 Poles Table in SQLite

```
1 private static final String CREATE_POLES_TABLE = "CREATE TABLE sig_poles “ +
2 ( pole_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
3     "area_id INTEGER," +
4     "pole_name TEXT," +
5     "pole_latitude DOULBE," +
6     "pole_longitude DOUBLE," +
7     "pole_altitude INT, " +
8     "pole_qr_code," +
9     "pole_difficulty )";
```

Another feature of the application is the pole content. Some of the poles are sponsored by local business, and some poles are placed on a historical location. Therefor the poles have some content attached to them. We store them on the MySQL database server, then when a user download the poles at start-up, it will fetch the pole content as well. The create table query is listed in Code Example 5.17.

Code Example 5.17 Pole Content Table in SQLite

```
1  private static final String CREATE_POLE_CONTENT_TABLE = "CREATE TABLE " +
2  " pole_contents ( pole_qr_code TEXT, " +
3  " pole_content TEXT )";
```

We also implemented a “Download poles” preference button in the User settings, which allows a user to always have the newest poles, because there will be submitted new poles during the summer.

SharedPreferences

We use shared preferences in the application to store preferences and data about the user, so that the application remembers important settings if it is killed and restarted. All the preferences from UserSettingsActivity are stored here, as well as the singleton instance of User. When a change is made to settings or the user-instance (for instance if the user changes home-area in UserSettingsActivity), the preferences is overwritten. In Code Example 5.18, methods for saving to and loading from SharedPreferences is shown.

Code Example 5.18 Usage of SharedPreferences in User.class

```
/*SAVE TO PREFERENCES*/
private void savePreferences() {
  // Open sharedpreferences for editing:
  SharedPreferences.Editor editor = sharedPreferences.edit();

  /* Put all data from this instance of User into the editor
   * (uppercase arguments are refs to static Strings,
   * used to uniquely identify a preference) */
  editor.putInt(USER_ID, m_Id);
  editor.putString(USER_NAME, m_Name);
  editor.putInt(USER_HOME_AREA, m_HomeArea);
  editor.putInt(USER_UNSUBMITTED, m_UnsubmittedPoles);

  if(m_FamilyMembers.isEmpty()) { // If no registered members, put 0
    editor.putInt(USER_FAMILY_SIZE, 0);

  } else { // Number of members > 0

    editor.putInt(USER_FAMILY_SIZE, m_FamilyMembers.size());
    User familyMember;
    for (int i = 0; i < m_FamilyMembers.size(); i++) {

      // Get User-objects from the ArrayList <User> m_FamilyMembers.
      familyMember = m_FamilyMembers.get(i);
      // Put family member-data into the editor
      editor.putInt    (USER_FAMILY_MEMBER + i + USER_ID,   familyMember.getId());
      editor.putString(USER_FAMILY_MEMBER + i + USER_NAME, familyMember.getName());
    }
  } editor.commit(); } // Commit changes to preferences

/* LOAD FROM PREFERENCES */
private void loadFromPreferences() {
  m_Id     = sharedPreferences.getInt(USER_ID, 0);
  m_Name   = sharedPreferences.getString(USER_NAME, "");
  m_HomeArea = sharedPreferences.getInt(USER_HOME_AREA, 0);
  m_UnsubmittedPoles = sharedPreferences.getInt(USER_UNSUBMITTED, 0);

  for (int i = 0; i < sharedPreferences.getInt(USER_FAMILY_SIZE, 0); i++ ) {
    m_FamilyMembers.add(new User(
          sharedPreferences.getInt(USER_FAMILY_MEMBER + i + USER_ID, 0),
    sharedPreferences.getString(USER_FAMILY_MEMBER + i + USER_NAME, null)));
} }
```

5.3.6 Performance and Optimization

Using the ViewHolder Pattern

ViewHolder might not be a renowned design pattern by Gang of Four, but it is quite essential in increasing performance in Android applications where lists are used. Android has a view feature called ListView, which allows us to create lists of objects. We are using ListView in our MapActivity to display every pole, and news stories in our NewsActivity. We experienced quite slow performance in both ListViews during development. At times it even made the application to crash, even on high-end devices. Most Google search results for "poor performance in listview", guided us to the ViewHolder pattern. ViewHolder radically increases performance in a listview. By creating a simple class in our custom list adapter class. The example below shows how we implemented a ViewHolder class in NewsActivity's list adapter.

Code Example 5.19 ViewHolder

```
private class ViewHolderItem {

    public TextView title;
    public TextView date;
    public TextView excerpt;

    public ViewHolderItem(View view) {

      title = (TextView) view.findViewById(R.id.news_title);
      date = (TextView) view.findViewById(R.id.news_date);
      excerpt = (TextView) view.findViewById(R.id.excerpt);
    }
}
```

By using public members, we can set and get the values without using setter and getter methods. The security of using private members are not a concern, since the class are private inside of a private custom list adapter class. We use a public constructor which finds each row view element. In this case, we are using three TextViews to display the title, date and excerpt of a news story. This means every row in the list consists of these three views.

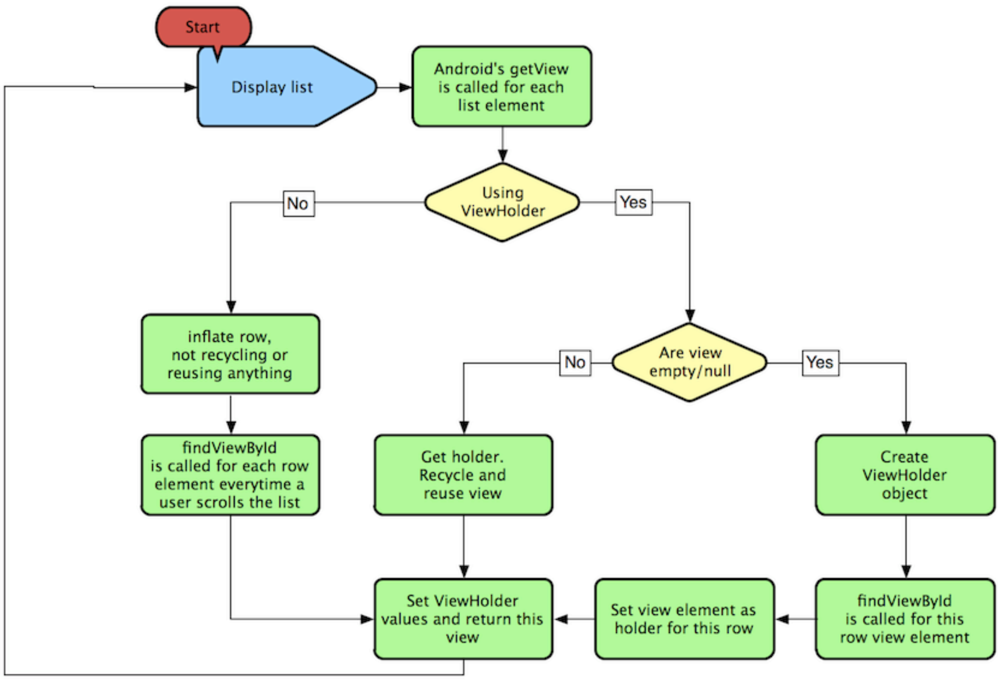


Figure 36: ViewHolder Pattern example

The flowchart (Figure 36) is a course illustration of how the list adapter with and without the ViewHolder pattern. The getView method looks like this:

public View getView(int position, View convertView, ViewGroup parent);

This function has to be implemented in order to implement a custom adapter. It takes three parameters: position in the ListView, convertView (in this case our custom row layout) and the parent, which is the View parent.

When the list is displayed, and the getView method is called for each element in a row. The code below shows a simplified example of how our getView method looks like.

Code Example 5.20 ViewHolder getView

```
public View getView(int position, View convertView, ViewGroup parent) {

ViewHolderItem viewHolder = null;

// If we have not seen this row element before we have to inflate it:
if(convertView == null) {
    LayoutInflater inflater = (LayoutInflater)getContext()
                    .getSystemService(Context.LAYOUT_INFLATER_SERVICE);

    // Inflate from XML layout
    convertView = inflater.inflate(R.layout.news_row, parent, false);

   // Create a new ViewHolder item
    viewHolder = new ViewHolderItem();

      // Stores our holder
     convertView.setTag(viewHolder);
} else {
// This view exists from before, reuse it. Get the holder.
viewHolder = (ViewHolderItem) convertView.getTag();
}
// Here we can set the text of the different textViews in our holder:
viewHolder.title.setText“(News ”title);

// The the row view is returned
return convertView;
}
```

We start by check if a convertView is null, if so, we have to inflate it from our layout XML file. Then we create a new ViewHolderItem, which calls the constructor that calls the findViewById method for each View in our custom XML layout (news_row parameter in the inflate method). Then we store the holder to that convertView element.

Because we set the holder of a convertView to be our viewHolder, we do not have to inflate the row and call the findViewById method again and again every time a user scrolls through the list. Which is a redundant process, because nothing has changed in our TextViews. Instead of inflating the row each time, we inflate the element once and reuse it. After we reuse it, we can easily set the text, for instance of the newsTitle member to the reused view. By implementing this simple pattern, we made sure the ListViews worked very well both on high-end and low-end devices. Which means no lagging and no freezing during scrolling.

5.3.7 Network Communication

ASyncTask

The Android Developer guide suggests to use ASyncTask to perform network operations. We implemented an ASyncTask (Code Example 5.21) as a generalized class to be able to use it to perform different tasks. Because this ASyncTask also will transfer the password from the device to the server, we choose to use HttpPost instead of HttpGet, as HttpPost supports using HTTPS. This encrypts the data to avoid man-in-the-middle attacks.

Code Example 5.21 ASyncTask

```
1  public class FetchDataTask extends AsyncTask<String, Integer, String> {
2
3    @Override
4    protected String doInBackground(String... params) {
5
6      HttpClient httpclient = new DefaultHttpClient();
7      HttpPost httppost = new HttpPost(params[0]);
8      url = params[0];
9
10     try {
11
12       httppost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
13       HttpResponse response = httpclient.execute(httppost);
14       jsonResult = inputStreamToString(response.getEntity().getContent());
15
16     } catch (ClientProtocolException e) {
17       return "";
18     } catch (UnsupportedEncodingException e) {
19       return "";
20     } catch (IOException e) {
21       return "";
22     }
23
24     return jsonResult;
25   }
```

We can instantiate this class from wherever we to download data. Most of the internet operations in the application is to connect to the database abstraction layer, which consists of several PHP-files on our server. The PHP-scripts returns a JSON-formatted array from a databasequery. If the objective is to for instance download a users visits, the query needs to uniquely identify the user. Because many of the operations needs one or more parameters, we created a function

to add parameters to the query.

Code Example 5.22 Add Parameter

```
1 // The list of parameters to send along with the PHP.
2 private List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>();
3
4 // Public method allow adding parameters from outside of FetchDataTask—class
5 public void addValuePair(BasicNameValuePair vp) {
6
7     nameValuePairs.add(vp);
8
9   }
```

The complete call to use the FetchDataTask to get the URL to the standing of the user is showed in Code Example 5.23

Code Example 5.23 FetchDataTask Usage

```
1 FetchDataTask connection = new FetchDataTask(UserActivity.this);
2     connection.addValuePair(new BasicNameValuePair("user_id",
3                 Integer.toString(User.getInstance(getApplicationContext()).getId())));
4
5     String result = null;
6
7     try {
8       result = connection.execute(Url.GET_STANDING).get().trim();
9     } catch (InterruptedException e) {
10      e.printStackTrace();
11    } catch (ExecutionException e) {
12      e.printStackTrace();
13    }
```

IntentService

Another way to do work in Android is by extending the IntentService-class. IntentService is used to execute tasks aside from the main thread. Like ASyncTask, IntentService also executes work asynchronously [13]. The main most important difference between ASyncTask and IntentService, is the execution. Where the ASyncTask freezes the UI, IntentService does not. In some operations, freezing the UI until the execution is finished is good, like when the user executes an operation that always should finish before the user can proceed. We use this approach for instance when a user visits a pole. When completing the operation is not crucial, we use IntentService. In this setting, not crucial means that aborting an operation will not have impact on any other part of the system. An example from our application is when the

user opens UserActivity. In this activity, the user should be presented with various statistics. The global leaderboard, which shows the users who have visited the most poles, is downloaded using an IntentService. This allows the user to navigate inside the activity while waiting for the IntentService to finish. When the IntentService is finished, the ListView which holds the leaderboard is updated with the downloaded data. In Code Example 5.25 we look at how we extended IntentService in HighScoreDownloadService, and in Code Example 5.24 we look at how this is being used in UserActivity.

Code Example 5.24 IntentService Usage in UserActivity

```
1  /* To create a HighScoreDownloadTask, we use an intent filter,  to make sure
2   we can receive the intent when it is finished.  This is done in the OnResume,
3    which is called when the UserActivity is brought to the foreground*/
4   @Override
5    protected void onResume() {
6      super.onResume();
7      registerReceiver(broadcastReceiver,
8        new IntentFilter(HighScoreDownloadService.TAG));
9      startService(new Intent(this, HighScoreDownloadService.class));
10   }
11
12 /* Unregister the receiver if the Activity no longer is in the foreground*/
13 @Override
14   protected void onPause() {
15     super.onPause();
16     try { unregisterReceiver(broadcastReceiver);
17     } catch (IllegalArgumentException ignored) { }
18   }
19
20 /* The broadcastReceiver in the call to registerReceiver above is
21 instantiated in the class: */
22 private BroadcastReceiver broadcastReceiver = new BroadcastReceiver() {
23   // When a broadcast is received, this method is called:
24   @Override
25   public void onReceive(Context context, Intent intent) {
26     Bundle bundle = intent.getExtras();
27     if (bundle != null) {
28       int result = bundle.getInt(HighScoreDownloadService.RESULT);
29
30       // If RESULT from the Extra is OK:
31       if (result == Activity.RESULT_OK) {
32         // Get the result-string from the Extra:
33         String topString = bundle.getString(HighScoreDownloadService.DOWNLOADED_DATA);
34         unregisterReceiver(broadcastReceiver);
35         // Use a custom parser to create an ArrayList from the result-string:
36         topList = parseJSONResult(topString);
37
38         // Use a custom ArrayAdapter to create list-items, and add them to a ListView:
39         CustomArrayAdapter adapter = new CustomArrayAdapter(context, topList);
40         statisticsListView.setAdapter(adapter);
41         adapter.notifyDataSetChanged();
42 } } } };
```

Code Example 5.25 IntentService Usage in UserActivity

```
// The public class that extends IntentService:
public class HighScoreDownloadService extends IntentService {

  public static final String TAG = "HighScoreDownloadService";
  public static final String RESULT = "result";
  public static final String DOWNLOADED_DATA = "downloaded_data";
  private List<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>();
  int result;

  public HighScoreDownloadService() { super("HighScoreDownloadService"); }

  // This method is called when startService is called in an Activity.
  @Override
  protected void onHandleIntent(Intent intent) {
    if (intent != null) {

      HttpClient httpClient = new DefaultHttpClient();
      HttpPost httpPost = new HttpPost(Url.GET_TOP_TEN);
      String downloadResult = null;

      nameValuePairs.add(new BasicNameValuePair("user_id",
              Integer.toString(User.getInstance(getApplicationContext()).getId())));
      try {
        HttpResponse response = null;
        httpPost.setEntity(new UrlEncodedFormEntity(nameValuePairs));
        response = httpClient.execute(httpPost);
        downloadResult = inputStreamToString(response.getEntity().getContent());
        result = Activity.RESULT_OK;
      } catch (IOException e) {
        result = Activity.RESULT_CANCELED;
      }
      publishResults(result, downloadResult);
    }
  }

  // Converts inputstream to string
  private String inputStreamToString(InputStream is) {/*Removed in example*/ }

  // Send a broadcast with the results
  private void publishResults(int result, String downloaded) {

    Intent intent = new Intent(TAG);
    intent.putExtra(RESULT, result);
    intent.putExtra(DOWNLOADED_DATA, downloaded);
    sendBroadcast(intent);
  }

}
```

5.4. iOS Application

5.4.1 Organizing the development

We never planned on doing the iOS application, because we did not know anything about iOS development. During a status meeting with the customer on the 21st of March (Meeting D.3), they discussed internally what they should do regarding the iOS application. Since we were not doing it, they said they had to purchase the application for a company. Because our main focus of the project has always been on pleasing the Customer, we had a quick group discussion. We agreed, and told them that we would try to create an application for iOS with limited functionality, and see how far we got. The customer was pleased to hear this, and we scheduled a status meeting two weeks later to discuss the iOS application progress. After the meeting (Meeting D.3) the group agreed that one should focus on finishing the Android application, and one focus on the iOS version. Since Anders is a registered iOS developer, we agreed that Markus should focus on finishing the Android application, while Anders develops the iOS version.

Even though we "split" the work of the applications between us, it was important to still to reach the deadline we initially set ourselves in the project planning phase. Therefor we had to come up with a limited scope based on the system requirements we created for the Android application. The first thing we did was to select which requirements were a must in the iOS application. Then we created a small feature list based on the one we made for Android. Even though Android and iOS are two different platforms, we knew that the main functionality should be the same. We quickly decided that the main functionality should be the following:

ID	Name <action> <result> <object>
1	Display map in application to a application user
2	Display QR Scanner in application to a application user
3	Store pole visits on device for a application user
4	Upload pole visits from device to database for a application user
5	Display pole list in map activity to a application user
6	Handle login information on device for a application user

Everything that was not listed in the miniature Feature List were considered a bonus. We knew that time was short, since we only had two weeks before we were to present what we were able to do with the iOS version.

Organizing the work with the iOS application regarding tools was quite easy. One has to use Apple's XCode as IDE to develop native iOS. XCode has a Git repository client attached to it which sets up a local Git repository in the project folder. Therefor it was no need to create another repository on Bitbucket, but only use the Issue tracker in the Android repository. XCode also comes with a great iOS simulator, which supports iPhone 4, 5 and iPad devices. Of course the simulator has some limitations, such as the lack of GPS and camera. Luckily we have access to an iPhone 5 and an iPad mini for testing.

The next thing we had to do, was to go through the guidelines for the App Store [14]. Submitted applications to the App Store has to follow certain criteria before they are released, which we had to consider while developing the application. Since Apple rejects applications with a lot of branding and advertisement, we knew that we could not integrate any of the sponsor's logos in the application.

We also had to focus on creating a stable and working application, since applications with clear bugs and crashes will not be approved. This was not a major concern, since we already wanted to create a functioning application.

5.4.2 Application design

After understanding more and more about how iOS development works, we started out setting up the initial project. Because we already had the system architecture and design in place for Android, we knew which classes to implement. We also knew how to structure the application thereafter. We needed to store the data, display the data and manipulate the data. Which is exactly how we initially set up the three-layer architecture (Figure 9). A similar pattern to Three-tier, is Model-View-Controller, which is much used in iOS development [15]. We decided to structure our app accordingly.

The application project was divided into three different groups, Model, View and Controller (Figure 37). The Model, or the data group was set up to contain all our data classes, which initially was the User and the Pole classes. We then added the controllers, such as a database handler class. This class was created to handle most the communication between the View and the Model (database), since the majority of our data would be stored in the SQLite database. At last we added the known view classes to the view group, for instance the Map and the User views. This ensured us a simple, but efficient structure in our Xcode project.

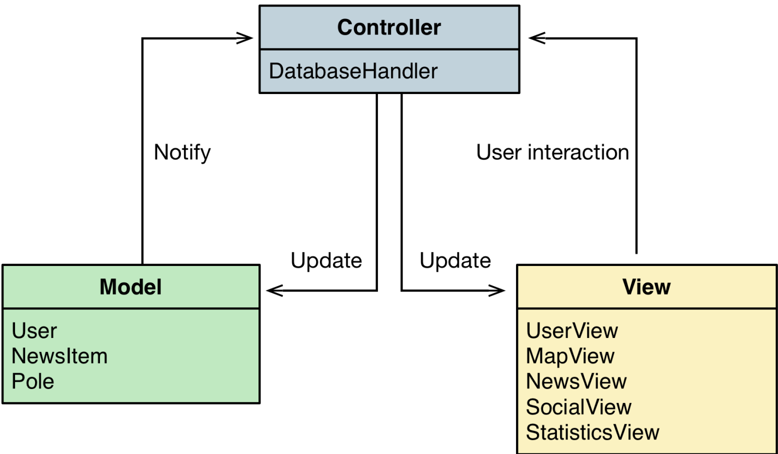


Figure 37: Model View Controller Application Structure

5.4.3 Map

After we set up the initial project and familiarized us with the App Store guidelines. Then we started testing different functionality of iOS, getting familiar with Objective-C (programming language for iOS) and Cocoa Touch (GUI framework for iOS). The most important feature was to display the map with the corresponding poles. We then again had to figure out what options there was for displaying the map with poles in our map view. The first idea we had was to do the same thing we did with the Android version, where we used Google Maps and add the tiles as an overlay. The second option was to use Apple's own MapKit then adding the tiles as an overlay. As a third option, we discussed using Open Street Map, which also supports tile overlays. We tried using Google Maps, but Apple's MapKit was much simpler to integrate into our iOS project. There was no need for API-keys and additional libraries, we only had to include the MapKit framework. One concern, which we also shared with the Android Application, was offline support. We had to discard both Google Maps and MapKit because of limited offline support.

After conducting more research, we found MapBox SDK for iOS. Since we already were familiar with MapBox (TileMill and mbtiles), we found this interesting. The MapBox SDK had support for mbtiles, which meant we could use MapTiler to create the mbtiles. MapTiler was also used to create the tiles for the Android application. MapBox SDK with mbtiles meant that we can have our own map in the application available offline. Another great feature is that the visible area in the application is only our own map. We did not have to limit any visible area, which we had to do with Google Maps in the Android application.

Code Example 5.26 shows how we load mbtiles to our application. The map loads different map based on each user's home area. For instance, if a user has homeAreaID set to 1, *gjovik* maps get loaded. Which means that adding a new if-statement is the only thing that has to be added to include a new map for and new area.

Code Example 5.26 Loading mbtiles

```
1
2 RMMapboxSource *tileSource = [[RMMapboxSource alloc] initWithMapID:@"gjovik"];
3 RMMapView *mapView = [[RMMapView alloc] initWithFrame:self.view.bounds
4                                           andTilesource:tileSource];
```

The next step after integrating our own map, was to download the poles from the database and store it on the device. Since we already used SQLite for Android, we felt this was a natural choice for iOS as well, because we can implement the same logic and the same database queries. We started out with a simple and self-written database handler which did not support features like transactions, which lead to an unstable application. Insert queries were executed our of order and caused crashes, which made us re-structure our DatabaseHandler class. We re-wrote some of the code to use transactions, which lead to a more stable application, but we still had a few bugs. Statements were still happening out-of-order which led to the application crashing.

We went online for tips on how SQLite could support out-of-order transactions and multi-threading. Most of the online tutorials recommended the FMDatabase framework. FMDatabase handled all of our issues and had a query "queue" system. We re-wrote the DatabaseHandler one final time including FMDatabase instead. This time the application functioned nicely, but still with a few bugs. After reading more about FMDB's "does and dont's" and by implementing the correct methods, the application worked like a charm.

After downloading and storing the poles, the next step was to add poles(markers) with different colors based on the pole difficulty to the map. MapBox has support for custom markers, which allowed us to create different markers with different colors. The only thing we had to implement was two methods.

Every item which is added to the map, is called a layer. A layer can be everything from an polygon to a marker. For instance, we implemented a for-loop, which iterates through the all poles and adds them as layers to the map (Code Example 5.27).

Code Example 5.27 Add Poles to Map

```
1
2 // Creates an immutable (read-only) array with the poles from the
3 //database in a selected area.
4 NSArray* poles = [databaseHandler getAllPolesFromDB];
5
6 // Iterates trough every pole in the array
7 for (Pole* pole in poles) {
8
9 // Gets the latitude and longitude of a pole, then creates a
10 // coordinate to where the pole should be placed
11   CLLocationCoordinate2D coord = CLLocationCoordinate2DMake(pole.lat,
12                                                             pole.lng);
13
14     // Creates an annotation layer (will soon be turned into a marker)
15     // based on the coordinates, and the pole name as title.
16     RMAnnotation *annotation = [[RMAnnotation alloc]
17     initWithMapView:mapView coordinate:coord andTitle:pole.name];
18
19     // This annotationType is used to identify the unique marker,
20     // and will be used in the tapOnCalloutAccessoryControl method
21     annotation.annotationType = pole.qrCode;
22
23     // Then reads the pole difficulty, and adds another identifier
24     // "userInfo", which will be used in the layerForAnnotation method
25     if ([pole.difficulty isEqual: @"1"]) {
26       annotation.userInfo = @"green";
27     }
28
29     if ([pole.difficulty isEqual: @"2"]) {
30       annotation.userInfo = @"blue";
31     }
32
33     if ([pole.difficulty isEqual: @"3"]) {
34       annotation.userInfo = @"red";
35     }
36
37     if ([pole.difficulty isEqual: @"4"]) {
38       annotation.userInfo = @"black";
39     }
40
41     // This will trigger the layerForAnnotation method
42     [mapView addAnnotation:annotation];
43
44   }
```

The method *addAnnotation* adds a pole to the map, which fires the *layerForAnnotation* method. Our *layerForAnnotation* implementation is in Code Example 5.28. We also wanted to implement a pole list similar to the one

in Android, where a user can select a pole and the map zooms to that pole. The issue we had with this functionality was that the information view (callout) did not appear. After reading the documentation we discovered the *selectAnnotation* method, which launches the callout when a pole is selected in the pole list. For the callout to work, we had to set *marker.canShowCallout = YES*.

Code Example 5.28 Create Marker Layer

```
1
2  - (RMMapLayer *)mapView:(RMMapView *)mapView
3    layerForAnnotation:(RMAnnotation *)annotation {
4
5  // Creates a marker object.
6    RMMarker *marker;
7
8    // Here we use the userInfo is accessed to create a marker
9    // with different colors based on the difficulty.
10   if ([annotation.userInfo isEqualToString:@"green"]) {
11     marker = [[RMMarker alloc] initWithMapboxMarkerImage:@"marker"
12     tintColor:[UIColor greenColor]];
13   }
14
15   if ([annotation.userInfo isEqualToString:@"blue"]) {
16     marker = [[RMMarker alloc] initWithMapboxMarkerImage:@"marker"
17     tintColor:[UIColor blueColor]];
18   }
19
20   if ([annotation.userInfo isEqualToString:@"red"]) {
21     marker = [[RMMarker alloc] initWithMapboxMarkerImage:@"marker"
22     tintColor:[UIColor redColor]];
23   }
24
25   if ([annotation.userInfo isEqualToString:@"black"]) {
26     marker = [[RMMarker alloc] initWithMapboxMarkerImage:@"marker"
27     tintColor:[UIColor blackColor]];
28   }
29
30   // Then adds a "info button" to the view when you click a marker.
31    marker.rightCalloutAccessoryView = [UIButton buttonWithType:
32                                              UIButtonTypeInfoLight];
33
34   // We display the title / pole name and the info button.
35   // When a marker is clicked, the tapOnCalloutAccessoryControl
36   // method is fired.
37   marker.canShowCallout = YES;
38
39   // Returns marker overlay.
40   return marker;
41 }
```

When a user clicks the marker on the map, the following method will get fired:

*- (void)tapOnCalloutAccessoryControl:(UIControl *)control forAnnotation: (RMAnnotation *)annotation onMap:(RMMapView *)map*

This method displays the pole information such as the name, altitude and content (if any). We use the *annotation.annotationType* (on our case is the QR code) to identify the pole, so the corresponding altitude and content is displayed for the correct pole. For instance the method in example 5.29, queries the SQLite *POLES* table to get the altitude for the pole with the corresponding QR code by accessing *annotation.annotationType*.

Code Example 5.29 Get Pole Altitude from Database

```

// Retrieve pole altitude
NSString* poleAltitutde = [databaseHandler
                             getPoleAltitude:annotation.annotationType];

// Implementation:
- (NSString*) getPoleAltitude: (NSString*)poleQR {

  // Open the database to interact with it
  [database open];

  // Create select query based on the QR code
  NSString *querySQL = [NSString stringWithFormat:
  @"SELECT ALT FROM POLES WHERE QRCODE=\"%@\"", poleQR];

// Executes query and the result is returned.
  FMResultSet *results = [database executeQuery:querySQL];

  NSString* altitude;

  // If a result is found, get the string for column with name "ALT"
  if([results next]) {
     altitude = [results stringForColumn:@"ALT"];
  }

  // Close the database so it can be accessed with other methods
  [database close];

  // return the altitude
  return altitude;
}
```

Regarding the marker look, the MapBox SDK supplies different images in their Maki icon set [16]. The following line of code shows how to set a marker with a Mapbox marker image:

RMMarker marker = [[RMMarker alloc]
initWithMapboxMarkerImage:@"marker"
tintColor:[UIColor redColor]];

We decided to use the "marker" icon, because it is it is similar to the marker icon in many other maps, such as MapKit and Google maps. All this code and graphics results in the final map view in Figure 38.

Figure 38: Final Map View

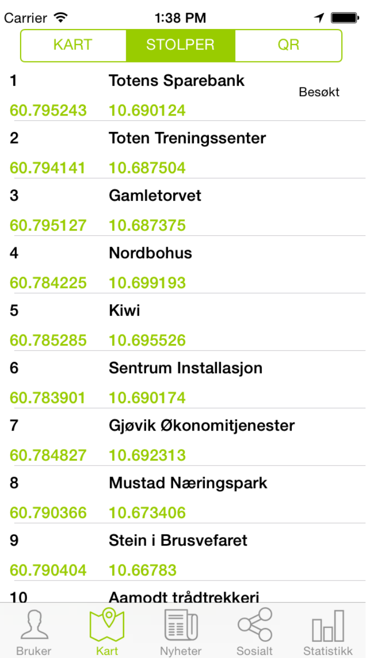


Figure 39: Final Pole List View

The finished map view contains three different subviews, which are called segments. The first segment contains the map. Next segment contains the list of poles in the selected home area (final pole list view in Figure 39). The final segment contains the QR scanner (elaborated in Section 5.4.5).

5.4.4 User data

Before we could get started with the QR scanner item from the Feature List, we had to implement the user functionality. Which meant creating a login view, fetch the user information from the database and storing it. iOS has an equivalent to Android's SharedPreferences, which is NSUserDefaults. By using this mechanism, we can store the user information so the user do not have to login each time the application starts. Since it is only one user logged in at a time, we felt it was unnecessary to create a database table to hold just a single user row, which is why we went with NSUserDefaults instead.

After hard-coding user-test data to get familiar with NSUserDefaults, the login view was created and the login connection to the database was successful. We downloaded the user data in the same fashion as we did on Android, and stored them in NSUserDefaults. The next step was to create a database table for storing the user's visits, both from when a pole QR code is scanned, and when they are downloaded from the web server. The method for uploading the user visits, which has yet to be uploaded, are in Code Example 5.30. It is a lot of iterating and conversion between data types, but it ensures that the result is a valid JSON string, which can be processed on the web server.

Code Example 5.30 Upload User Visits

```
// Open database for selection
[database open];

// Create a mutable (read-write) array to store the results
// from the visits table.
NSMutableArray * objects = [[NSMutableArray alloc] init];

// Selects the pole ID, QR code and the time the pole was visited,
// where the uploaded flag is 0/NO
NSString *querySQL = @"SELECT ID, QRCODE, VISITTIME "
"FROM VISITS LEFT JOIN POLES USING(QRCODE) WHERE UPLOADED = 0";

// Executes the query and returns the result
FMResultSet *results = [database executeQuery:querySQL];

// Iterates if any results
while([results next]) {

  // Create a mutable array to store each row of result
  NSMutableArray* temp = [[NSMutableArray alloc] init];

  // Gets the ID, VISITTIME and QRCODE columns and
  // adds them to the temp array
  temp[0] = [results stringForColumn:@"ID"];
  temp[1] = [results stringForColumn:@"VISITTIME"];
  temp[2] = [results stringForColumn:@"QRCODE"];

  // Then sets the uploaded flag to 1/YES
  [database executeUpdate:@"UPDATE VISITS SET
  UPLOADED ='1' WHERE QRCODE = ?", temp[2]];

  // Adds temp array to the array of results
  [objects addObject:temp];
}

// closes the database so it can be used in other methods
[results close];

// Then create a dictionary from that array which will be turned
// into a valid JSON string to be uploaded to the database.
NSMutableArray *visits = [[NSMutableArray alloc] init];

// Iterates through the visit arrays, adds them to the dictionary.
for (NSArray* t in objects) {

  NSMutableDictionary *jsonDict = [[NSMutableDictionary alloc] init];

  [jsonDict setObject:t[0] forKey:@"pole\_id"];
  [jsonDict setObject:[[User getInstance] userID] forKey:@"user\_id"];
  [jsonDict setObject:t[1] forKey:@"visit\_time"];
  [visits jsonDict];
}

// Creates a JSON data object from the directory of objects
NSData* jsonData = [NSJSONSerialization dataWithJSONObject:visits
options:NSJSONWritingPrettyPrinted error:&error];


// Creates the final JSON string from the valid JSON data
NSString* visited = [[NSString alloc]initWithData:jsonData
                     encoding:NSUTF8StringEncoding];

// Database connection is then made and the results uploaded ...
```

Figure 40 illustrates how the final user view looks. We have a profile picture which the user can provide by using the website. Next we have the number of visited poles-counter, which sums up all the visits in all the areas. Below the counter, we have single area counters which shows how many you have visited in that particular area. The menu has three menu items. First item is the user's latest visits in a scrollable list. Second item is the top ten standing overall. Third item is the team activity, which members have taken which poles with corresponding date.

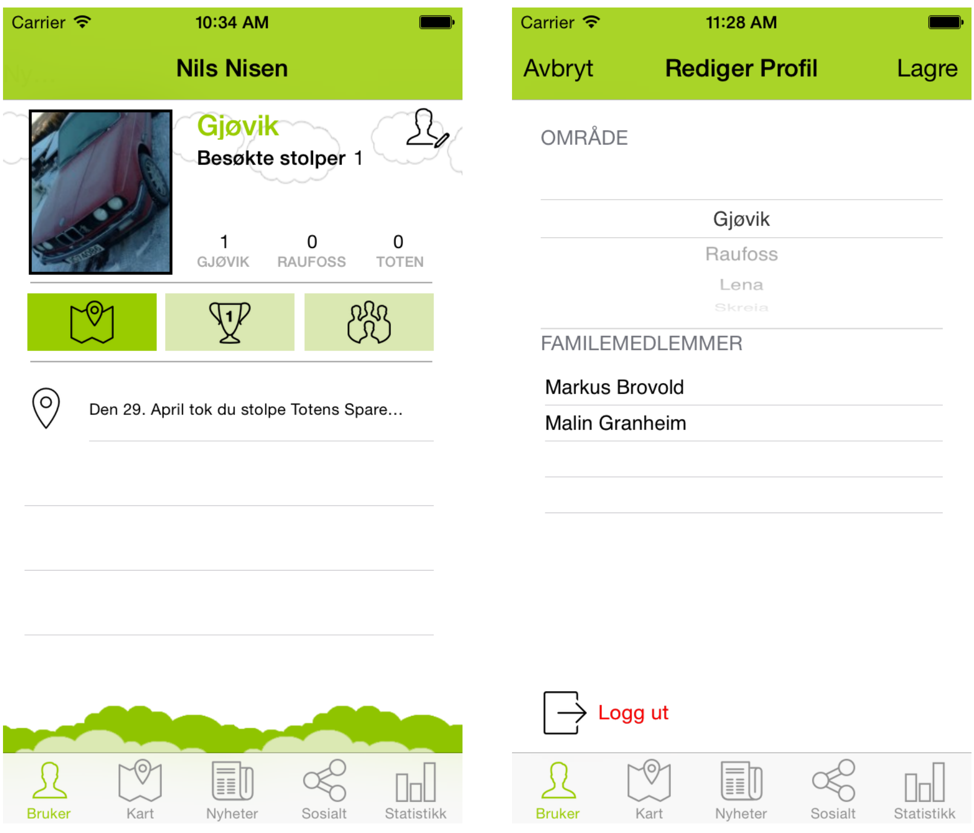


Figure 40: Final User View

Figure 41: User Settings View

The top right item of Figure 40 is the "Edit Profile" button. In the Profile settings the user can switch home area and include family members in their pole search. The Figure 41 shows how the UI looks. We have the option to store ("Lagre") this changes, but also discard ("Avbryt"). Which means unless a user presses the store button in the settings menu, it does not save the changes. To implement the cancel changes feature, we had to store the original data whenever a user presses the "Edit Profile" button. Code Example 5.31 shows a simplified version

of the implementation. When a user changes something, the value is stored in the database and/or the NSUserDefaults. We only restore the original data if the user presses cancel. Which means when a user changes something, and then presses "Lagre", nothing really happens, because the value changes is already stored.

Code Example 5.31 Save/Cancel User Setting

```
1  // In the .h (interface) file
2  @interface SettingsViewController {
3    NSArray* restoreSelectedFamilyMembers;
4    NSString* restoreSelectedHomeAreaID;
5  }
6
7  // In the m. (implementation file)
8
9  // This method gets fired when a user presses the "Edit Profile" button.
10 // Which stores all the original data in the class variables.
11 - (void) storeOriginalData {
12
13   // Stores the home area
14   restoreSelectedHomeAreaID = [[NSUserDefaults standardUserDefaults]
15                                objectForKey:HOME_AREA_ID];
16
17   // Stores the selected family members
18   restoreSelectedFamilyMembers = [[DatabaseHandler getInstance]
19                                   getSelectedFamilyMembers];
20 }
21
22
23 // This method gets fired if the user presses "Cancel".
24 - (void)restoreOriginalData:(UIBarButtonItem*)sender {
25
26   // Restores the home area
27   [[NSUserDefaults standardUserDefaults] setObject:restoreSelectedHomeAreaID
28                                          forKey:HOME_AREA_ID];
29
30   // Restores all the selected family members
31   for (NSString* member in restoreSelectedFamilyMembers) {
32
33     [[DatabaseHandler getInstance] updateFamilyIncludedInTrip:member :@"YES"];
34   }
35
36   // If any changes done in the selected family table, restore to original state.
37   for (NSString* member in [[DatabaseHandler getInstance] getFamilyMembers]) {
38
39     if (![restoreSelectedFamilyMembers containsObject:member]) {
40
41       [[DatabaseHandler getInstance] updateFamilyIncludedInTrip:member :@"NO"];
42     }
43   }
44 }
```

5.4.5 QR scanner and pole visits

Since the database table and logic all ready was in place in the Android application, we wrote similar logic and queries for Objective-C. The next step was to find a stable QR scanner solution. Since iOS 7, Apple added QR reader functionality to their AVFoundation framework. After following a tutorial, we had successfully implemented our own scanner without any external libraries. Next we created our own QR codes to test with. We then successfully read the code, stored it in our local database as a visited pole.

Since storing visits locally on the device was not enough, we had implement upload functionality to our database as well. Therefor we decided to do it in a similar fashion as we did on the Android version. The iOS implementation (simplified) is listed in Code Example 5.32. We validate the scanned QR Code with the QR codes stored in the device's local SQLite database. Then we either submit the result (if it is valid), or present the user with a suitable error message.

Code Example 5.32 Scan QR Code

```
1
2  // Gets checks if scanned object is a valid QR code in local database
3  NSString* qrCode = [[[DatabaseHandler getInstance]
4                         getSinglePole:[scanObject stringValue]] m_PoleQRCode];
5
6  // If valid QR code
7  if (qrCode) {
8
9    // Add pole to visited table using QR Code
10   [db visitPole:qrCode];
11
12   // Inform the user that the visit went well
13   [result setText: @"Stolpen ble lagt inn!"];
14
15   // Stop the QR Code reader, and
16   [self stopReading];
17
18   // Tell the database to upload the visits to server
19   [[DatabaseHandler getInstance] visitPoles];
20
21 // Else not valid QR code
22 } else {
23
24 // Inform the user
25 [result setText: @"Ugyldig QR-kode!"];
26
27 // Re-restart the QR code reader
28 [self startReading];
29 }
```

5.4.6 Supported devices

The application is created for iPhone 3GSS (which is required if we want our application to be accepted by Apple), 4, 5/5s and iPad (both original and mini). We decided on doing an universal application, because it is not hard to change an iPhone Storyboard to an iPad Storyboard. By making a copy of the excising Storyboard and altering two lines of code, we get a Storyboard for iPad. Then we had to alter the UI elements to fit the iPad screen (explained in Code Example 5.33). This "hack" came from a StackOverflow post ([17]).

Code Example 5.33 Convert iPhone Storyboard to iPad Storyboard

```

<!-- Change -->
targetRuntime="iOS.CocoaTouch"
<!-- to -->
targetRuntime="iOS.CocoaTouch.iPad"

<!-- Then change -->
<simulatedScreenMetrics key="destination" type="retina4"/>
<!-- to -->
<simulatedScreenMetrics key="destination"/>
```

The target iOS version is 7.0 since the majority of iOS users are now using iOS 7. According to Apple ([18]), 87% of all Apple devices are using iOS 7. Equal to the Android application, the functionality should not suffer at the expense of supportability. We have included some functionality which is unsupported in iOS 6 and below, such as the QR scanner functionality from AVFoundation.

5.4.7 Graphics

Graphics on iOS requires two images, one for normal screens, and another for retina display. It is quite simple to implement these images, we simply have to add one image for standard resolution and one for retina. The retina must have twice the resolution as the normal image. All you have to do is, to add "@2x" before the file extension for a retina image. For instance "user.png" for normal screens, and "user@2x.png" for retina display. To add the image to our UI, we simply have to call:

UIImage userImage = [UIImage imageWithName:"user"];*
Then iOS loads the correct image for the corresponding screen resolution.

Figure 42: Normal Display

Figure 43: Retina Display

All of the icons used in the application is provided by Icons 8 [19] and is free to use. The set comes in five different sizes for most icons. For instance when we want a button with icon size 50x50 pixels, we had to include the 50x50 icon to support normal screen and the 100x100 pixel to support retina display. Both resolutions is provided by the Icons 8 pack (Example icons in Figure 44).

Figure 44: Icons 8 Example Icons

5.5. Deployment

The deadline from Gjø-Vard Orienteering were the launch date of the project, which was the 10th of May 2014. We finished the applications and submitted both to the application stores about a week before the launch. The iOS application had some review time, and was luckily approved just in time on May 9th. Regarding the Android application, there was no review time since Google Play allows application submission without a review. To prepare for the launch, we had to move the entire system from our own private server which we have used for testing, to the server Gjø-Vard Orienteering had acquired.

This operation to move the the Abstraction Layer was painless. Since we set up an class to handle the host information, we only had to change the username and password to match the new server. We also initially set up the complete database with all the tables on our own server. PhpMyAdmin allows export of a database, which worked perfectly. We only had to login to the new server and import the old tables into the new database using phpMyAdmin. Both the export and import operation with assurance testing took approximately 10 minutes.

The next step was to change the URLs in both of the applications. It turned out to cause a lot of havoc, because we did not start out correctly during the development phase. We were foolish enough to hard-code the URLs inside different methods in the Android and iOS application. This is not good practice. To solve this once and for all, incase we had to move it to another server. We created a constants class which holds all of the URLs (Code Example 5.34). This solution is what we should have implemented initially. We have certainly learned from this experience. It would have saved us a lot of time with the deployment, if the URLs class was implemented originally. We could easily just altered the DOMAIN string to the correct domain, and upload the files to the new server.

Code Example 5.34 URL Class

```
1  public class Url {
2    // Domain URL
3    private static final String DOMAIN = "http://stolpejakten.no/";
4
5
6    // File URL
7    public static final String GET_AREAS = DOMAIN + "abstractionlayer/get_areas.php";
8    public static final String GET_POLES = DOMAIN + "abstractionlayer/get_poles.php";
9  }
```

The last step and the most time consuming was to move the website to the new server. One can not move Wordpress files to a new server and new domain without having to deal with a few issues. The first thing we had to do was to update the .htaccess-file. To alter this file we had to go into the Wordpress Dashboard and update the "Permalinks" setting, which updates the .htaccess-file. Permalinks is a permanent link (URL), which means it never changes by it self, and is often used on blog sites such as Wordpress. An example is the permalink for the different leaderboards:

http://www.stolpejakten.no/topplister-for-ulike-omrader/.

If we did not complete this step, none of our pages with work. After the permalinks was in place, we had to add two lines in the wp-config file:

update_option('siteurl', 'http://stolpejakten.no');
update_option('home', 'http://stolpejakten.no');

Then we had to load the website once, go back into the wp-config and remove the two previously inserted lines. After this was taken care of, all we had to do was to update the website URL inside the Wordpress Dashboard. This entire process was not the most time consuming, but the process of figuring out why website did not function properly. Such as the website redirects to the old page and figuring out why and how to correct it. The migration took about half a day's work.

The fun part is that Gjø-Vard Orienteering at first wanted us to use *stolpejakt.no*. Then they told us to use *stolpejakten.no* instead. Which meant we had to repeat the entire process once more time to get it to the correct server and domain. At least this time, we knew exactly how to do it, so it luckily took us about half an hour.

When the website and database were up and running at the correct server, under the correct domain, we allowed some of the members of Gjø-Vard Orienteering to test the system. After the customer had tested the system, some eager users already downloaded the applications and registered on the website on the the 9th of May. Since the system was up and running a few days before launch we concluded that the deployment, even though we had to do it twice, was a success.

6. Testing and Quality Assurance

6.1. Unit testing

The first thing we started with was the requirements, then we finished the design. After the design was finished, we turned the design into code. For each new feature we completed, we had to test the feature in the system. The problem with the manual testing we executed is that it is ineffective. If this was a paid assignment, it would have been cheaper for customer if we had automated unit tests, since developer time is expensive [20].

We wanted to implement automated tests in the development phase. Which for us meant to write the test before we write the code. It turned out that this was easier said than done. We were not strict enough to implement unit tests in the development phase. The idea was to implement both end-to-end and single method unit testing. Unit testing is simply put, the testing of functionality and parts (units) of the code. This testing practice allows a developer to check if the system works as excepted, without having to do any manual testing.

Unit tests would help us check if our code logic is correct, and it helps us as developers to write effective code that is easy to test [21]. Unit tests would therefor ensure a more efficient development phase, if we knew how to properly implement it. This was the main reason why we did not implement unit testing. We did not know to execute it. Instead we agreed that we would focus on the development (write code), and create an application and website which works well.

Unit testing would also help us regarding scaleability. If we were to add new maps, more areas, more poles and more content, we could have used the automated test cases to check that scaling the database did not affect the current working system. Because we do not have any automated tests, we had to check manually if new parts of the system functioned properly. To simplify our development phase, what we could have done is to implement a few simple end-to-end tests. Figure 45 illustrates how the flow of our potential end-to-end unit test could function.

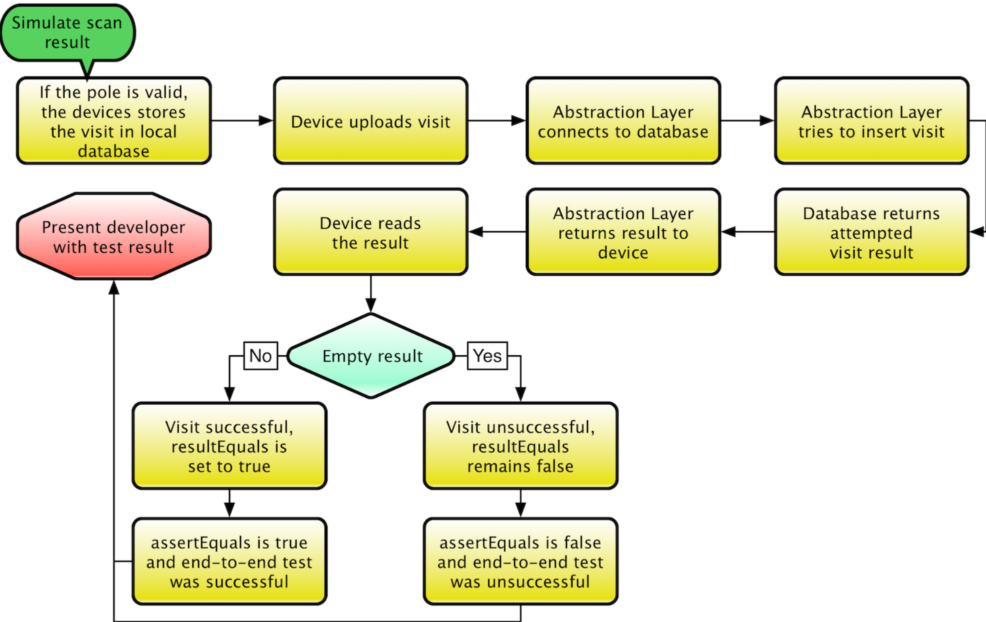


Figure 45: End-to-End Example Test

The end-to-end test could have been used to ensure that the communication functions between the three different layers (elaborated in Chapter 4) of the system. Code Example 6.1 elaborates how we could have implemented a simple test case in Android, to test a complex part of the system. The same test logic could also been added to the iOS application.

Code Example 6.1 Register Visit Test from Application to Server

```
String poleQr = "ASDF";
databaseHandler.visitPole(poleQr);

// Database abstraction layer returns the row ID of the insert
String result = databaseHandler.uploadVisits();

String assertResult = "0";

boolean resultEquals = false;


// If result is not empty, upload was successful.
if (!assertResult .equals(result)) {
    resultEquals = true;
}

// If true, test is OK.
assertTrue(resultEquals);
```

Instead of implementing the for mentioned unit tests, we used a lot of Logger functionality in Android Studio, and in XCode. Loggers prints text to a console in the respective IDE. For instance when we were testing to see if the *String result = databaseHandler.uploadVisits();* functioned properly, we printed the content of the result as such: *Log.d("RESULT", result);*. By doing it in this manor, we had to check the console to check if the string was correct instead of testing it automatically.

6.2. Beta testing

Since we as the developers were the alpha testers, the customer became the beta testers. A week before the project started, we opened the website for the customer to register and test the functionality of the website. We also submitted the Android application to Google Play. Which allowed the customer to download and test the functionality of the application before launch. Then a day before launch, the iOS application got accepted to the App Store, which then allowed the customer to test the application as well. By including the customer a few days before the users, helped ensure that the system functions properly after launch.

There were a few bugs and some unexpected behavior. Which might have been solved earlier if we had implemented unit tests. There was a short period for the customer to test before the project launch, which meant a few hectic days of bug fixing and manual testing. Then again, this might have been avoided if we had implemented automated tests. What we did learn during the beta test, is that next project we are involved in will include automated tests.

Since we did not implement proper tests, we still wanted to stress-test the system. To stress-test the system we decided to let the users try the system over the launch weekend (10th and 11th of May 2014). Over the weekend we could use the feedback from the users to improve and ensure the quality of the system.

6.3. User feedback

Since the project started on the 10th of May, the user mass has increased rapidly each day. On the 14th of May the database contains 1474 members including family members (Figure 46), and 10146 registered pole visits (Figure 47).

```
select count(*) from sig_wp_users
```

+ Innstillinger

count(*)
1474

Figure 46: Number of Users in Database

```
select count(*) from sig_visits
```

+ Innstillinger

count(*)
10146

Figure 47: Number of Pole Visits in Database

Since we implemented a contact form on the website, and a feedback option in the Android Application, the users which have experienced difficulties has contacted us. If an issue received affected more than one user, we have corrected it, and re-submitted it to the respective store. We came across two major issues, one became of user feedback and another because our own field test (elaborated later in the chapter).

Two examples of issues from users regarding the Android application, was the offline storing of the pole visits and the QR scanner crashes with Sony devices. The offline storage the of visits, turned out to only upload the latest visit when the user decided to upload unsubmitted poles. This reason behind this issue, was an JSON Object placed on outside of the for-loop, and not inside. Which means that the visit got overwritten with the next visit, and the last visit remained. The next issue was the QR scanner crash on Sony devices. It turned out to be an issue with Xperia models and the autoFocus-functionality. To fix this issue, we had to create a separate activity (Figure 48), which is launched if a device is a Sony model (Code Example 6.2 explains the implementation).

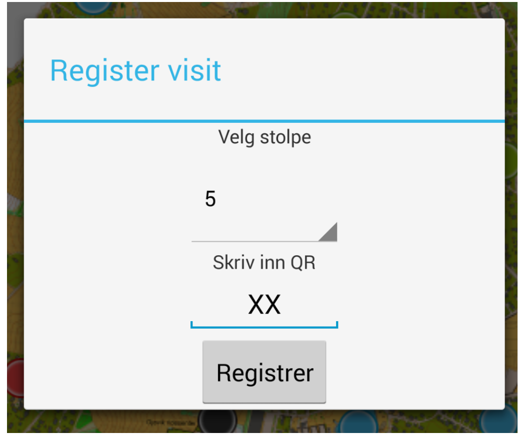


Figure 48: Manual Visit Activity

Code Example 6.2 QR Scanner Issue with Sony Devices

```
1  private void setupScanButton(){
2  // Get manufacturerfinal
3  String manufacturer = android.os.Build.MANUFACTURER;
4
5  // Set up onClick for button
6  Button scanButton = (Button)findViewById(R.id.scan_button);
7  scanButton.setOnClickListener(new View.OnClickListener(){
8
9       @Override
10      public void onClick(View v){
11
12      // if manufacturer is sony, start the manual activity
13        if(manufacturer.toLowerCase(Locale.ENGLISH).contains("sony")) {
14        startActivity(new Intent(MapActivity.this, ManualVisitActivity.class));
15
16     // if not sony, start normal scanner activity
17     } else {
18        startActivity(new Intent(getApplicationContext(), ScannerActivity.class));
19     }
20   }});
21 }
```

Two examples of iOS issues which were corrected because of user feedback was the non-scrollable registration form, and missing QR scan button on iPhone (3.5-inch screens, which is iPhone 4S and older). It turned out that the registration form was not completely visible on a 3.5-inch screen, and was non-scrollable. To solve this issue we had to move all the textfields higher up on the screen so they became visible on 3.5-inch screens.

It was our fault, because we did not check every part of the system manually on a simulator properly. The next issue was the missing QR scanner button on 3.5-inch screens.Users reported that the QR scanner "froze" after scanning a code, because the screen turned white, and nothing happened. One user sent us a screenshot of the issue (Figure 49).

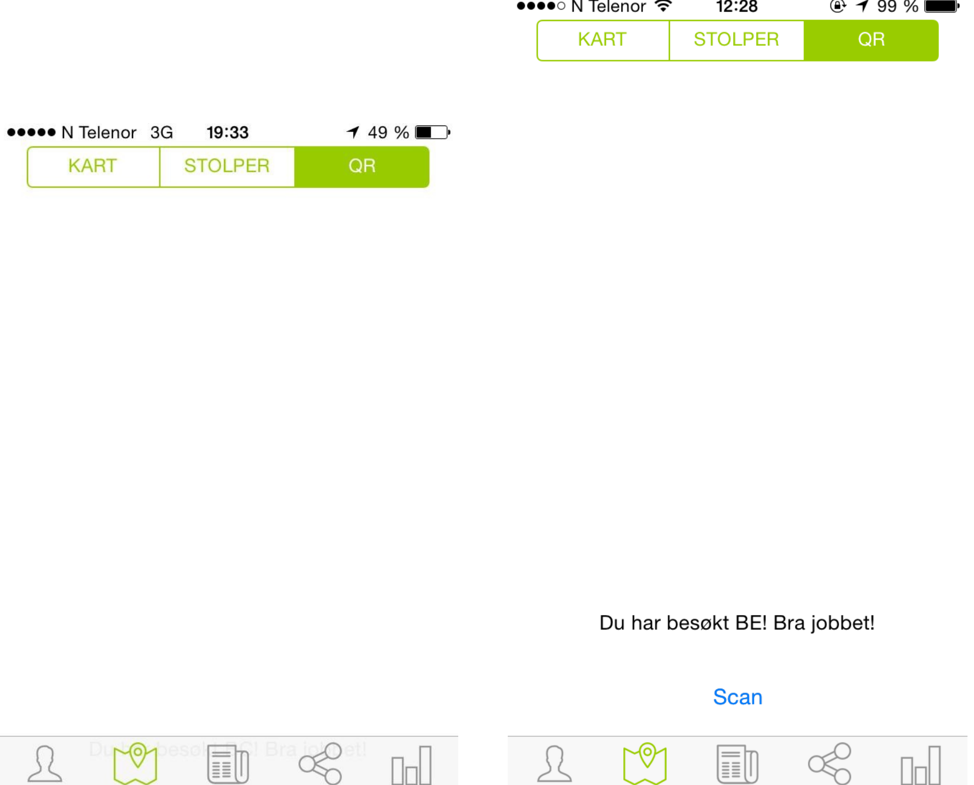


Figure 49: 3.5-inch Screen Issue with QR Scanner Button

Figure 50: Propper QR Scanner Button on 4-inch Screens

Figure 50 is a screenshot of a 4 inch screen, which displays the button properly. It did not turned out to be a frozen screen, but what happened on 3.5-inch screens, was that the result text and the Scan button popped up too far down on the screen. Which resulted in invisible button and text on 3.5-inch screens. These two issues was fixed because of user feedback.

6.4. Field test and Quality control

Incase there were no user feedback, we decided to go out field our selves to test the two applications as soon as the poles were in place. Figure 51 and 52 is pictures of us, out in the field testing the QR codes and the applications. We had with us three devices, one Android and one iPhone 5S with Internet connection, and one Android without Internet connection.

While we were testing ten poles in Gjøvik, it seemed that everything worked perfectly. After we were done, we controlled the visits in the database. Both the iPhone 5S and HTC One (both with Internet Connection) worked perfect. The Samsung S2 without Internet connection did not submit every unsubmitted pole as mentioned previously when it came back online, only the last visit. This is how we came across, and solved this issue.

Because of the field test we conducted and because we implemented the different feedback possibilities, such as contact forms, as well as using the feedback from the users in both App Store and Google Play, we have been able to correct the known kinks and bugs. Because of all the feedback and manual tests, were able to ensure the quality of the system.

A week after the launch, we have a functional system which is capable of handling a constant growth of user mass. Because the system is functional and running well a week after launch, it ensures us that the system is scaleable. Which means we can implement more areas, more poles and more users in the feature without any unexpected issues. Even though we did not implement automated tests, we have a successful and functional system which the users seem to enjoy. On the other hand, we have learned that if we implemented automated tests, we would have saved us a lot of time and frustration. We have certainly learned from our mistakes.

Figure 51: Markus in the Field

Figure 52: Anders in the Field

7. Conclusion

7.1. Assignment Evaluation and Results

Since we had worked with development of an application for the Sprek project in 2013, we somewhat knew in large terms what the customer wanted before choosing this project as our bachelor assignment. Other student groups had interviewed a lot of the users from 2013, which helped both us and the customer to determine what the main focus should be in the new application. Because of all this knowledge, productivity was high from the very beginning of the development phase. The customers initial plan was to pay for another development team to develop a independent solution parallel to ours. Because we got such a good start, they cancelled the parallel process. The fact that they trusted us with the future for the project gave us confidence in what we had accomplished.

Initially, we had decided that we would not make an application for iOS. In Meeting D.3), the customer discussed what they were going to do with the iOS application. Because we had most of the Android application and the entire back-end finished, we told them that we would try to make one and see how far we got. Over the easter holiday, the application got developed. At least to a stage which allowed for beta testing. Since most of the logic was in place in the Android application, all we had to do was to learn Objective-C enough to implement the same logic for iOS. After two weeks of development and testing, the application was ready to be presented to Gjø-Vard Orienteering. They were impressed and decided to go with our application, instead of acquiring it from someone else.

7.2. Group work evaluation

In the development process, we have kept each iterations deadline. After each iteration, we have had status meetings, where we have evaluated the previous iteration and planned the next. Both group members has been eager to satisfy the customers wishes. This has affected the writing process of the thesis. In retrospect, we could have set shorter limits for each of the iterations, and stopped developing at an earlier stage. In the final stage of the project period, we have had a lot of contact with users, which have occupied much of our time. The upside of this is that the product delivered to the customer is more stable, and in the end, all we wanted was happy users.

7.3. Further development and maintenance

Although the Android application is finished according to the requirement, there is always a new feature which could be implemented. As we see it, the most interesting feature to add is the possibility to download new maps without having to update the application.

First of all, since we have zero experience with iOS, there is a lot which can be done it improve the application. The iOS application could be updated to get all the features that the Android app has, such as the same level of offline support. Currently, the iOS application is missing the functionality to upload unsubmitted poles. It only stores the visits in on the device, but do not tell the user if visits are unsubmitted. To upload unsubmitted poles until the new upload is available, a user has to be online for the last pole visit of the day to get all of their unsubmitted poles uploaded.

Second, the application is not very robust in terms of networking. When the opportunity arrives, we will upgrade the networking part of the application to use AFNetworking. AFNetworking is a networking framework for iOS. This framework will greatly improve the network functionality of the application.

Finally, the UI could be upgraded. We might replace the storyboards with programmatically created UI instead in the feature. It is easier to deal with one file of code for different screen support, rather than the current three different storyboards. Which is something we did not figure out until after the application was done.

7.4. Conclusion

It has been an interesting project period. Gjø-Vard Orienteering has been both an interesting and challenging customer. It has been a great, challenging experience. In addition to the practical skills we have acquired, this project has let us take advantage of much of the theoretical knowledge from our three years from Gjøvik University College.

It has been great to work for such a comprehensive public health initiative, and the fast-growing user-mass at the last weeks of the project period has been a nice reward.

We are especially proud to have native applications for both Android and iOS. Both of the applications can and will be used as future reference. We are both very proud of the end result.

We will bring all of our new experiences which we have acquired during this project period into our newly founded company, AndMark Software Development DA.

The iOS application has been downloaded on 872 units, and the Android application has been downloaded on 538 units

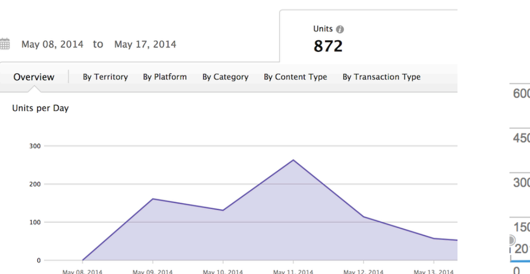


Figure 53: Total Downloads in App Store

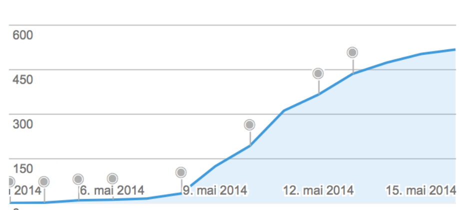


Figure 54: Total downloads in Google Play

Bibliography

[1] Schwaber, K. & Sutherland, J. 2011. The scrum guide - the definitive guide to scrum: The rules of the game. https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guide.pdf. (Visited Feb. 2014).

[2] Google. Unknown. Starting an activity, basics, android developer. http://developer.android.com/training/basics/activity-lifecycle/starting.html. (Visited Apr. 2014).

[3] Kniberg, H. 2007. *Scrum and XP from the Trenches*. C4Media.

[4] Goyal, S. 2007. Major seminar on feature driven development. http://csis.pace.edu/~marchese/CS616/Agile/FDD/fdd.pdf. (Visited Feb. 2014).

[5] Google. Unknown. Code style guidelines for contributors. http://source.android.com/source/code-style.html. (Visited Feb. 2014).

[6] Otgard, H. 2014. Det norske android-landskapet. http://beta.knowitlabs.no/android-telefoner-i-norge/. (Visited Jan. 2014).

[7] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. 1995. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[8] PHProup, T. Unknown. Prepared statements and stored procedures. http://no2.php.net/pdo.prepared-statements. (Visited May 2014).

[9] Bill, G. Unknown. 45 useful responsive web design tools and techniques. http://www.freshdesignweb.com/responsive-web-design-tools-and-techniques.html. (Visited May 2014).

[10] Google. Unknown. Iconography, design, android developers. http://developer.android.com/design/style/iconography.html. (Visited May 2014).

[11] geografa. 2011. Isom. https://github.com/geografa/ISOM. (Visited Feb. 2014).

[12] Wikipedia. Unknown. Mjøsa. http://no.wikipedia.org/wiki/Mj%C3%B8sa. (Visited May 2014).

[13] Google. 2014. Intentservice, reference, android developers. http://developer.android.com/reference/android/app/IntentService.html. (Visited Apr. 2014).

[14] Apple. Unknown. App store review guidelines. https://developer.apple.com/appstore/resources/approval/guidelines.html. (Visited Apr. 2014).

[15] Apple. Unknown. Model-view-controller. https://developer.apple.com/library/ios/documentation/general/conceptual/devpedia-cocoacore/MVC.html. (Visited Apr. 2014).

[16] MapBox. Unknown. Maki icon set. https://www.mapbox.com/maki/. (Visited May 2014).

[17] tharkay. Unknown. Converting storyboard from iphone to ipad. http://stackoverflow.com/a/8694985. (Visited Apr. 2014).

[18] Apple. Unknown. App store distribution. https://developer.apple.com/support/appstore. (Visited May 2014).

[19] Icons8. Unknown. Icon pack for ios 7. http://icons8.com/free-ios-7-icons-in-vector/. (Visited Apr. 2014).

[20] Vaaraniemi, S. 2003. The benefits of automated unit testing. http://www.codeproject.com/Articles/5404/The-benefits-of-automated-unit-testing. (Visited May 2014).

[21] McFarlin, T. 2012. The beginner's guide to unit testing: What is unit testing? http://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728. (Visited May 2014).

Appendix

A. Project agreement

PROJECT AGREEMENT

between Gjøvik University College (GUC) (education institution),

Arnfinn Pedersen (Gjø-vard O-lag)

(employer), and

Anders Hagebakken

Markus Brovold

(student(s))

The agreement specifies obligations of the contracting parties concerning the completion of the project and the rights to use the results that the project produces:

1. The student(s) shall complete the project in the period from 01.01.2014 to 19.05.2014.

   The students shall in this period follow a set schedule where GUC gives academic supervision. The employer contributes with project assistance as agreed upon at set times. The employer puts knowledge and materials at disposal necessary to complete the project. It is assumed that given problems in the project are adapted to a suitable level for the students' academic knowledge. It is the employer's duty to evaluate the project for free on enquiry from GUC.

2. The costs of completion of the project are covered as follows:
   - Employer covers completion of the project such as materials, phone/fax, travelling and necessary accommodation on places far from GUC. Students cover the expenses for printing and completion of the written assignment of the project.
   - The right of ownership to potential prototypes falls to those who have paid the components and materials and so on used to make the prototype. If it is necessary with larger or specific investments to complete the project, it has to be made an own agreement between parties about potential cost allocation and right of ownership.

3. GUC is no guarantor that what employer have ordered works after intentions, nor that the project will be completed. The project must be considered as an exam related assignment that will be evaluated by lecturer/supervisor and examiner. Nevertheless it is an obligation for the performer of the project to complete it according to specifications, function level and times as agreed upon.

4. The total assignment with drawings, models and apparatus as well as program listing, source codes and so on included as a part of or as an appendix to the assignment, is handed over as a copy to GUC who free of charge can use it in lessons and in research purpose. The assignment or appendix cannot be used by GUC for other purposes, and will not be handed over to an outsider without an agreement with the rest of the parties in this agreement. This applies as well to companies where employees at GUC and/or students have interests.

   Assignments with grade C or better are registered and placed at the school's library. An electronic project assignment without attachments will be placed on the library part of the school's website. This depends on that the students sign a separate agreement where they give the library rights to make their main project available both on print and on Internet (ck. The Copyright Act). Employer and supervisor accept this kind of disclosure when they sign this project agreement, and they must possibly give a written message to students and dean if they during the project period change view on this kind of disclosure.

B. Project Plan

Project plan

Sprek i Gjøvik

Markus Brovold

Anders Hagebakken

May 2, 2014

Table of Contents

1 Objectives and limitations **1**
1.1 Background . . . 1
1.2 Objectives . . . 1
1.3 Boundaries . . . 1

2 Scope **3**
2.1 Field of study . . . 3
2.2 Limitations . . . 3
2.3 Description . . . 3

3 Project organization **4**
3.1 Roles and responsibilities . . . 4
3.2 Group rules and routines . . . 4

4 Planning, follow-up and reporting **4**
4.1 Choosing Software development process . . . 4
4.2 Status meeting and points of decision plans . . . 5

5 Quality control **5**
5.1 Documentation and coding conventions . . . 5
5.2 Version Control / Source Code Management . . . 5

6 Risk analysis **6**

7 Plan of execution **7**
7.1 Gantt . . . 7
7.2 Work Breakdown Structure . . . 8

1 Objectives and limitations

1.1 Background

Sprek i Gjøvik was started by Gjø-Vard orienteering the summer of 2013. The main goals of the project is to get people out and about Gjøvik. The summer of 2013 had about thousand users. They wish to double that amount for this summer. This summer they will include other areas of the municipality, like Sørbyen and Biri. They will place markers around Gjøvik, with different difficulties, and the point being that the users should visit as many as possible during the summer.

During last semester we developed an Android application for the Sprek i Gjøvik project. We got a feel of what the project were about, and got to know our customer. Some of our fellow students also developed applications, and others did a thorough studies of the previous user experiences and feedback. We have access to their work, and will at least be using the user feedback in the development process in our application.

1.2 Objectives

1.2.1 Impact objectives

- Increased public health in Gjøvik.
- Increased navigational skills for the everyday user.

1.2.2 Outcome objectives

- Increase the user mass from summer 2013.
- Get more people to use the Android application.

1.2.3 Learning objectives

- Learn more about state of the art technologies used in both Android and web development, as well as the link between the two.
- Learn more about writing a proper thesis.

1.3 Boundaries

We have to stay within our time frame; January 13th - May 19th of this year. Our application have to be ready for out-in-the-field at least by the 3rd of May. The customer has to have an application

ready for deployment before the project goes live and the poles are placed at their locations.

Since we are not handling sensitive data (i.e. Visa card number and credit info), we really do not have to worry about any government laws. When a user trusts us with their email address, we have a responsibility to at least treat it with respect and not abuse their email in any way.

Because we are doing a bachelor assignment, the development is free. Hence we do not have a budget to worry about.

When it comes to technology, we do not have any clear boundaries from the customer. As long as we use free libraries and open software.

2 Scope

2.1 Field of study

During our bachelor assignment we will be studying Android Application development, different server side and map technologies, cross platform data interaction (Web and Android), database design and system architecture.

2.2 Limitations

Since we are developing for Android, we are limited to the native functionality that Android has to offer. Because we our application research does not start until middle of February, we are uncertain of what limitations we might encounter in the development process.

2.3 Description

The assignment will be split into several tasks, as the system required by the customer consists of several modules.

The first module is the server side, which shall be capable of supporting all the required functions in the later described mobile application. The server side should contain a database. The database should be designed in such a way that the product owner can manage it after the project period.

The second module of the assignment is the mobile application. The applications main purposes are to show the user the interactive map with the poles position, the ability to scan QR-codes from the poles and register which poles are visited by each user. The application should synchronize statistics with the database on the server. The application should be able to save data for later synchronization in cases where the user has no internet access.

The third module of the system is a web site. This web site should let users register and manage family accounts. It should also show various statistics. The web site should also contain a admin-module where the administrators can manage both poles and users. There should also be a possibility to register companies. This will be further specified in the research process in February.

3 Project organization

3.1 Roles and responsibilities

Customer
Gjø-Vard Orienteering club, represented by Bjørn Godager.
Mentor
Mariusz Nowostawski, Associate Professor at Gjøvik University College.
Group leader
Anders Hagebakken. Group leader has the overall responsibility to make sure we follow our time schedule as well as the main communication with both the customer and supervisor.
Chief Development Officer
Markus Brovold. The Chief Development Officer has the responsibility to verify the result of a sprint according to specifications. He is also responsible for keeping proper documentation at the required level at all time.

3.2 Group rules and routines

- Use version control - Submit working code!
- Follow standard Java/Android Coding conventions.
- Document code!
- If a disagreement should occur, supervisor and/or customer should be consulted in the decision-making.

4 Planning, follow-up and reporting

4.1 Choosing Software development process

Our group consists for two people. Because of the limited amount of available work force, we need a way to maximize productivity. From earlier projects we have experienced that a Scrum/Kanban hybrid works very well for our group. We find that a simple process with simple but effective tools are the best for us. From Scrum we will be using iterations (non-static length), and from Kanban we will be using the Work In Progress Limit handling our tasks. This way we can control what is being done by whom easily.

We will be using iteration even though Kanban do not have iteration, but the iteration length will be non-static. This fits our development process well, and is a great compromise between iterations and no iterations.

4.2 Status meeting and points of decision plans

Since we are working by an agile software development methodology, the process will be divided into shorter iterations ("sprints"). After each iteration there will be a short status meeting with our supervisor and customer. During these meetings we will discuss the past iteration and the next iteration.This will give us a foundation for further development. Then we can create our to-do list for that iteration. By including the customer in our development process, we can re-prioritize tasks if required in our backlog for the next iteration.

5 Quality control

5.1 Documentation and coding conventions

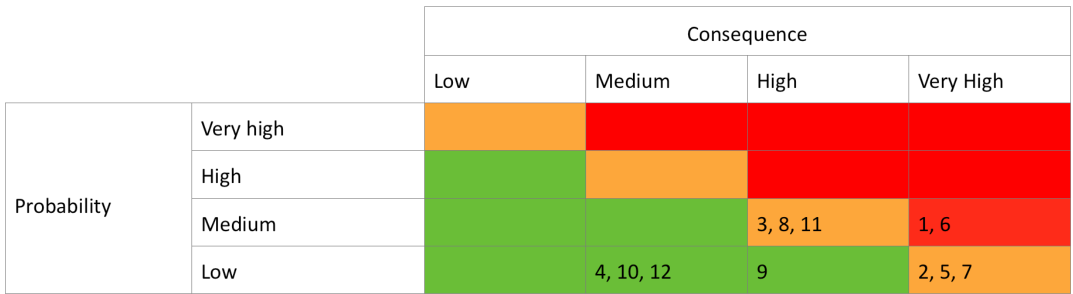
We will deliver the application to the customer for further development and maintenance. Therefore we have to provide documentation good enough for someone else to continue to work with the application. Code will be commented and documented during the development-process.

Regarding coding conventions and -style, we will follow the standard Java coding conventions while developing for Android. Since we will be using Android Studio for the development, the software formats the code correctly for us.

5.2 Version Control / Source Code Management

Source code for the application will be stored in a Git repository. The group is familiar with Git and how a distributed system works from previous projects. Bitbucket will be the service provider. Because the group members are accustomed with using Git through Bitbucket, this will let us focus more on the development and the report.

6 Risk analysis



		Consequence			
		Low	Medium	High	Very High
Probability	Very high				
	High				
	Medium			3, 8, 11	1, 6
	Low		4, 10, 12	9	2, 5, 7

Number	Risk	Probability	Consequence	Impact
1	Android Not Responding	Medium	Very high	High
2	Unavailable server	Low	Very high	Medium
3	Corrupt data	Medium	High	Medium
4	User data theft / leakage	Low	Medium	Low
5	Project group gets dissolved	Low	Very High	Medium
6	Serious disease	Medium	Very High	High
7	Hardcoded passwords	Low	Very High	Medium
8	Customer absence	Medium	High	Medium
9	Governmental constraints	Low	High	Low
10	Bitbucket unavailable	Low	Medium	Low
11	Not following time schedule	Medium	High	Medium
12	Incomplete product	Low	Medium	Low

Number	Follow-up
1	Test on several different devices. Follow good coding conventions.
2	Store user data locally on device, upload when server is available.
3	Have proper fail safe checks before user data is manipulated.
5	Be friends.
6	Stay healthy.
7	Make sure not to hardcode anything. Do code review frequently.
8	Establish different contact points within Gjø-Vard Orienteering.
11	Re-prioritize if needed. Discard trivial functionality for important functionality.

C. Correspondence with Service Provider regarding Shared SSL

D. Meetings

D.1. First meeting with Bjørn Godager - 13.01.2014

Date: **Jan 13. 2014**
Present at meeting:
Customer: Bjørn Godager
Students: Anders Hagebakken
Markus Brovold
Purpose of meeting:

1. Thoughts on web site
Nettside til prosjektet kan lages med wordpress. Nettside kan ha en admin-modul hvor administrator kan legge inn flere stolper. Nettsiden kan også vise besøksstatistikk. /
Web site is preferred to be WordPress or similar. Customer wants to be able to post news and access administrator-tasks such as adding poles to the map and getting user-statistics. Web site should also show visitor-per-pole-statistics to all users.
Highscore/hall of fame.

2. Application(s) platform
Bachelor project should focus mainly on Android-application.

3. Maps
Customer wants us to spend some time on researching what map-types will be best on Android, regarding both performance and detail level.

4. Database
Customer wants a easy-to-manage database, which the orienteering club can host when the project period is done. Customer wants documentation on how to use this after the project-period.

5. Competitive application
Study how Karlstad manages competitions and if possible; improve and apply.

D.2. Status meeting - 21.02.2014

Tema: Sprek i Gjøvik - bachelor 2014

Arrangør: AndMark v/ Anders Hagebakken og Markus Brovold
Inviterte: Arnfinn Pedersen, Bjørn Godager og Mariusz Nowostawski

Tilstede: Anders, Markus, Arnfinn, Bjørn
Ikke møtt: Mariusz

Referatet lister opp temaene vi diskuterte i kronologisk rekkefølge.

1 Avvikshåndtering

(med avvik menes her manglende stolpe eller ødelagt stolpe)

1.1 Hvem skal få ansvaret for å ordne manglende stolper eller feil stolpe-informasjon?

- Hvert område har kontakperson(er) som tar imot meldinger om stolper.
- Kontaktpersonene tar kontakt med områdeadministrator (se pkt. 3.2)

2 Hovedstruktur på websiden:

2.1 stolpejakt.no:

- Generell informasjon om prosjektet / generell informasjon om stolpejakt
- Informasjon/bruksanvisning for apper, manuell registrering osv.
- Kart med klikkbare områder, f.eks Gjøvik, Hamar, osv, hvor brukeren kan klikke på ønsket område og blir sendt dit.

2.2 De enkelte områdenes hjemmesider:

- På de spesielle områdene (Hamar/Gjøvik osv):
- Spesiell informasjon for området.
- Nyheter for området.

3 Administrering:

3.1 Super-administrator (Arnfinn eller Bjørn, evt Anders og Markus):

- Opprette under-administrator for de spesifikke områdene.
- Legge ut globale nyheter (nyheter for alle områder - på forsiden)

3.2 Underadministratorer:

- Legge inn stolper for sine områder
- Publisere nyheter for sitt område

Eksempel (“user story”):

Arnfinn gir “per erik fra toten” underadministrator-rettigheter.
“Per erik fra toten” legger inn stolper som står på Totenåsen.

“Per erik fra toten” tar kontakt med Arnfinn, fordi det skal arrangeres et stort stolpeløp 14. Juni på Toten. Arnfinn lager da en nyhet på stolpejakten.no slik at alle norges stolpejegere får med seg dette.

4 Kart:

4.1: Fordelene med offline kart er at vi kan style kartene, slik at vi får orienteringskart.

4.2 **Bjørn** undersøker videre eksportmuligheter av kart fra OCAD slik at vi kan bruke kartdata rett i appen, hhv. med GeoJSON-format.

4.3 Knut Olaf Sunde kan å koble SOSI-formatet til OCAD, utviklerteamet tar evt. kontakt med han.

4.4 Det er også et alternativ å ha kartdataene på en WMS-server (altså ferdig stylet med orientering-utseende), dette sjekker **Bjørn** videre og melder tilbake til utviklerteamet om.

D.3. Status meeting - 21.03.2014

Tema: Status på appen og nettsida
Inviterte: Anders, Markus, Arnfinn, Bjørn, Mariusz
Møtt: Anders, Markus, Arnfinn, Bjørn

1. Demonstrasjon av hjemmesiden og appen

1.2 Mangler:

1.2.1 Familie-konto (flere navn på samme epost)

1.2.2 Geo-referering av ocad-kart. Bjørn fikser dette asap.

2 Generere QR

https://www.the-qrcode-generator.com

3 Legge inn stolper

3.1 På nettsida: kan gjøres så fort nettstedet er flyttet

3.2 Via XML eller KML-filer: Kan laste opp en fil med info for flere stolper (for eksempel et "slipp" når det skal komme nye stolper)

4 iOS-app

4.1 Anders og Markus ser på muligheten for å sette sammen en app til iOS. Dette blir i første omgang bare en kartlegging for å avgjøre hvorvidt vi kan lage en app eller ikke.

D.4. Status meeting - 24.03.2014

Invited: Anders, Markus, Bjørn, Arnfinn, Mariusz
Present: All

1. Demo of applications (iOS and Android)
   - Bjørn and Arnfin will send updated pole difficulty-XML
   - Markus will update pole-content to show altitude from DB.
   - Logos in app (for companies) are not allowed in iOS-app.
   - Anders and Markus will take care of pole-numbers.

2. Demo of web site:
   - Make visited poles highlighted
   - Bjørn will create .PDF-maps for download
   - Bjørn will take care of TotenTroll-map (get it georeferenced)
   - Cultural information will be delivered later.
   - make the list on http://www.stolpejakt.no/mine-besokte-stolper/ clickable or hoverable to show cultural information. Also some information about altitude and difficulty to make it easier to decide which pole to visit.
   - Maybe take a look at labs.kartverket.no
   - Firmalogoer (til nettsiden) fikser Arnfinn, via Sondre. Disse må grupperes på område. Disse skal være klikkbare på nettsiden (Anders).

3. Additional wishes for app:

:Icon: Herb/AndMark -

Create a new icon with a pole (black on top, white and wood)

Note to self:

Manual pole registering is not checking admin-shit, and registering wrong display_name in sig_wp_users !!!

Thesis:
   - Write something about the teamwork and the.
   - Write about time used on developing
   - Always say why when a desicion is made.
   - Write a lot about cross platform development., but only in the imp-chapter.
   - Write about how we decided that iOS is actually doable.