University of Stavanger

1 Faculty of Science and Technology

BACHELOR`S THESIS

Study program/Specialization: Bachelor in engineering - computer	Spring semester, 2017 Open
Writer: Alexander Wiig Sørensen Espen Stuvik Vier	Alexander W. Sørensen Espen S. Vier ..... (Writer's signature)
Faculty supervisor: Erlend Tøssebro External supervisor(s):	
Thesis title: Behaviour Mapper – a tablet application for behaviour mapping	
Credits (ECTS): 20	
Key words: Behaviour mapping Tablet iPad Swift	Pages: 58 (incl. 1 attached + code) Stavanger, 14.05.2017

2017

Behaviour Mapper – a tablet application for behaviour mapping

ALEXANDER WIIG SØRENSEN
ESPEN STUVIK VIER

Summary

The bachelor thesis' main goal was to create a tablet based application for use in behavioural mapping. This application should allow the user to register participants and serve as a replacement to using pen and paper.

The assignment was presented by the University of Stavanger. The group was free to decide how the task should be solved, what platform to build the application for and what functionality it would have.

The application was made with a focus on speed of registration and ease of use. At the same time the infrastructure was designed to easily allow further development and implementation of new functionality.

Table of Contents

2 Introduction

2.1 Description of Assignment

The assignment was to make an application for tablets to be used by students studying city planning. This application should allow them to register people's behaviour on a map. There were no restrictions or requirements in how this should be solved.

Suggested functionality:

- Before starting registration, the user creates a list of symbols (”legend”) to be registered. To be able to differentiate between factors such as gender, whether the person is with a child/dog/other animal, whether they are walking/running/biking, etc. For this, the application should have a library of icons for the user to choose between.
- During registration, the user could press on a point at the screen to select position. Then the user selects one of the symbols created to choose what is being registered at that point.
- It should be possible to edit registrations after they have been registered. As an example, to edit the time of registration, to allow registration of two persons simultaneously.
- Option to create notes.
- Option to display data on a map, both during registration and afterwards. During registration, the entries should be shown in a way that keeps the map clean and visible. A problem with the paper based solutions is how cluttered they become during registration.

2.2 Goal

The goal was to make an application for tablets that should be able to replace pen and paper when doing behavioural mapping. For this application to work as a full substitute to pen and paper, it needed to perform just as well in several key areas. It needed to be easy and intuitive to use. Registration of people's behaviour needed to be very fast. The different entries and their categories also needed to be visually distinct from each other. At the same time, the registrations needed to be precise.

2.3 Motivation

Now, virtually all behaviour mapping is done using pen and paper. Despite their widespread use for this activity, pen and paper has several limitations. Registering a person with pen is time consuming. Especially when different persons are to be registered with visually unique and easily discernible icons. The paper map can quickly get cluttered as more people are registered on it. There also isn't a quick, easy way to register the precise time for each entry, which reduces the precision of the registrations.

When the registration process is finished, the data needs to be imported to different digital tools for further work and analyses. It can be difficult to accurately read the registrations written on paper, as many entries can be placed close together, have symbols that are hard to differentiate or perhaps have been written fast and imprecisely. Especially difficult is it to read entries written by other people.

These are all issues that can potentially be fixed by a tablet based application.

3 Background

This chapter is intended to give an overview of terms and relevant concepts that can be required to best understand the intent behind the application and how it's designed.

3.1 Behavioural Mapping

Behavioural mapping is a systematic and non-intrusive method to map peoples conduct in relation to the properties of a given physical area. The user registers on map and eventual tables where individuals are within a specified area. It's often appropriate to also note their activity level, presumed age, gender, ethnicity and in what degree they interact with other people in the area. Behaviour mapping might be used in areas like schools, city centres, hospitals and malls.[1]

3.1.1 Purpose

In every man-made area, there can exist a disconnect between the intention of the design and how people use it. Behaviour mapping can be used to expose how it is used. This information can then be used to plan eventual improvements to the area, improve the design of similar areas in the future or to explain that a newly designed or redesigned area supports the behaviour it was intended for. The area to be mapped is usually printed on a paper, where the mapping can be drawn with a pen (Figure 3-1).

Figure 3-1: Behaviour mapping on paper[2]

[1] (Clyne, 2017)
[2] (Reinertsen, 2017)

3.1.2 Mapping

Before the mapping process begins, it needs to established what parameters are important to the research. The parameters are different from area to area, and depend on the purpose of the research. In one case, it might be relevant to know the time of each observation, the gender of the observed or what they direction they are moving – and not only where they were observed. Different observations should be marked differently, to be able to distinguish them later when the data is to be analysed. Each type of observation might have their own symbols and colours.

The mapping is to be done non-intrusive, so that the observer themselves not affect the behaviour one wishes to document. Because of this, it is usually done from a distance, and often in public spaces, where consent isn't required from the people being observed. Each participant is observed until some predetermined criteria is reached. This could be that the participant has stopped, arrived at the area, left the area or moved a given distance. When this happens, the participant is marked on the map, and the observer is ready to observe the next participant.

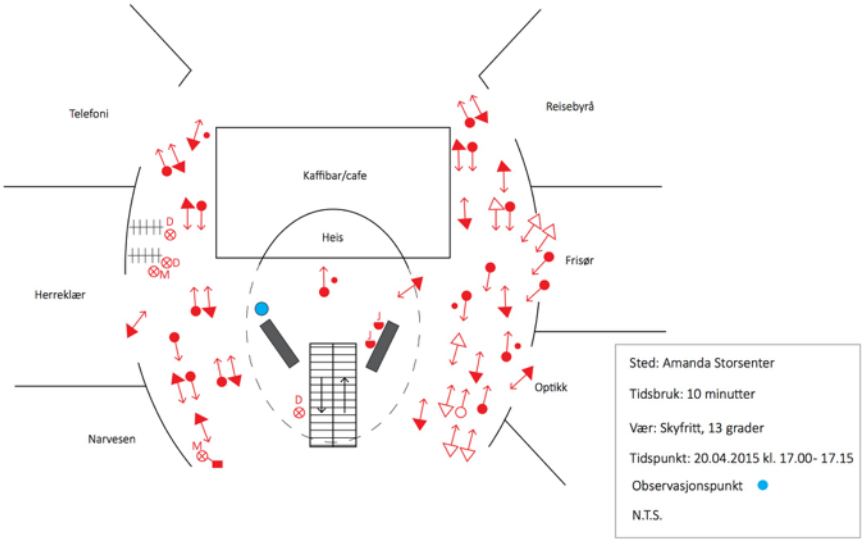

Figure 3-2: Graphical illustration of behavioural mapping[1]

[1] (Madsen, 2015)

3.1.3 Analysis

To view the results in an aggregate, is a common starting point for analysis. A quick overview of what are the most trafficked and which are under-utilized. This can be translated to a graphical representation of the mapping result (Figure 3-2). Very often it is useful to know more than a participant's position and movement. It can be good know how many of this type of participant is observed. For example, it could be useful to know how many where biking on a given time, and not only where they were biking. Further work is often done with concrete numerical data. This is often used to create tables and graphs to present the findings (Figure 3-3).

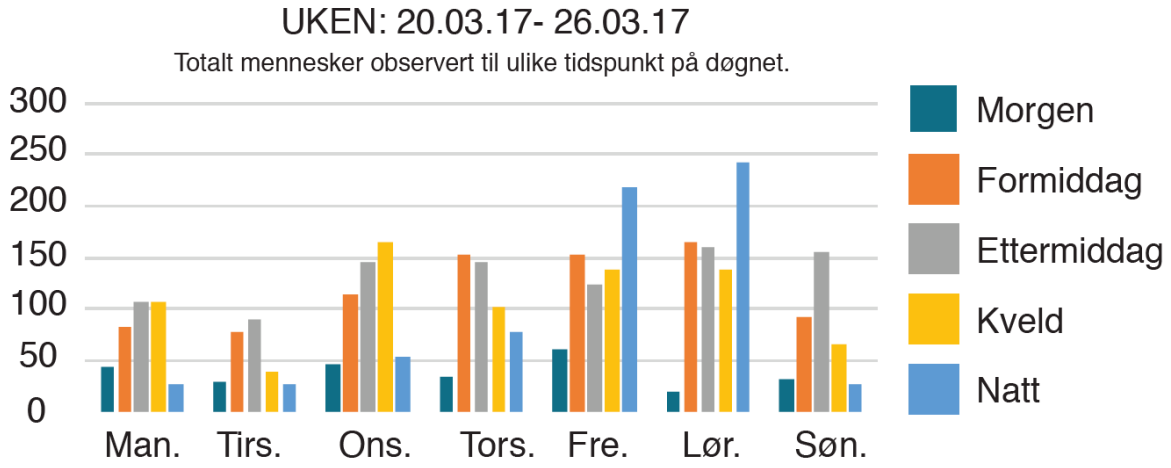

Figure 3-3: Graph of data generated from Behaviour Mapper[1]

[1] (Reinertsen, 2017)

3.2 Platform

The application is developed for the tablet family iPad, using the operating system iOS. This chapter gives basic description of the structure of iOS-based applications, and the programming language the application is made in: Swift.

3.2.1 iOS Applications

When an iOS application starts, the *UIApplicationMain* function initializes several key objects. At the core of every iOS application, is *UIApplication*, whose job is to facilitate the interactions between the system and other objects in the application (Figure 3-4).

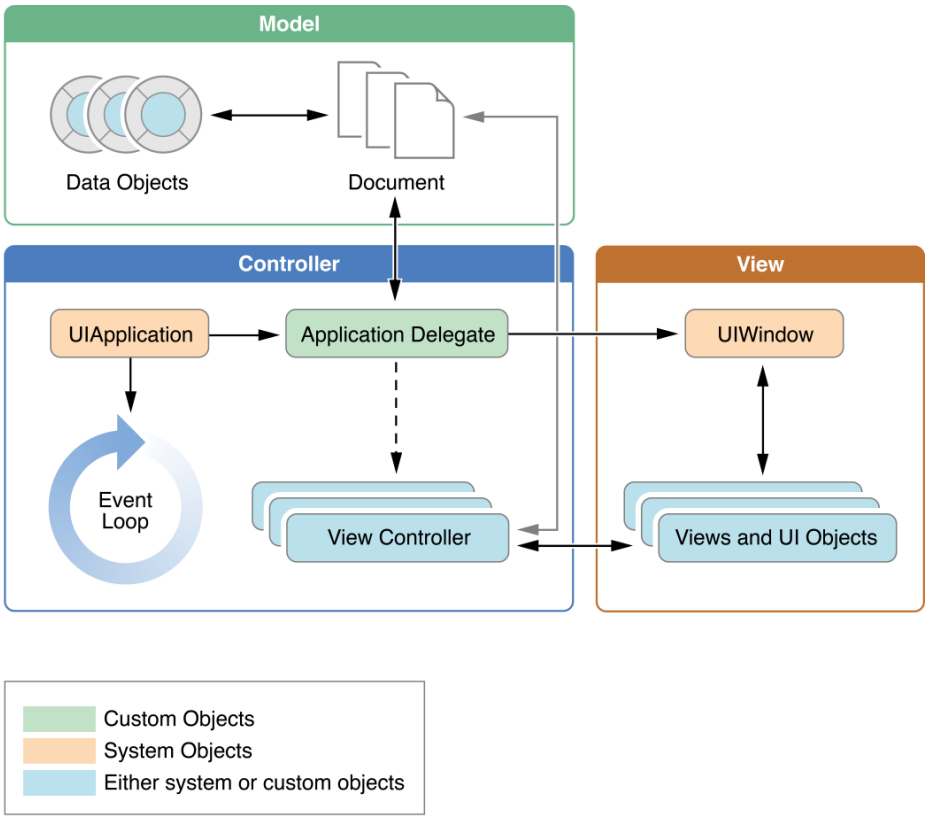

Figure 3-4: Key objects in an iOS application[1]

Object types

Description of the roles of different objects in an iOS application:

- **UIApplication:** Controls the *event loop* sequence and other high-level behaviour. It also notifies some application transitions and events to its delegate, which is a custom object.
- **Application Delegate:** The core of the custom code. This works with UIApplication to initialize the application and select other events. This is the only object guaranteed to exist in all applications, and is often used to initialize an applications data structure.

[1] (Apple Inc., 2017)

- **Documents and data model:** Data model objects store the applications content and is specific to each application. What these contain will depend on the purpose of the application.
- **View objects, control objects**: A view is an object that draws content in a rectangular area and responds to events occurring within that same area. Controls are a special type of view responsible for implementing common user interface objects like buttons, text fields and switches.
- **View controller:** *View controllers* control the presentation of the applications content on the screen. A view controller controls a single view, and its subviews. When presented, the view controller will make its views visible by installing them in the applications window.
- **UIWindow**: A *UIWindow* coordinates the presentation of one or more views on a screen. To change the content in an application, a view controller is used to change what views are visible in the corresponding window.

Activity states

An application can be in one of several different activity states (Figure 3-5). The operating system moves the application from one state to another in response to actions occurring in the system. These actions could be the user pressing the home button of the device, receiving a call or another action that causes an interrupt.

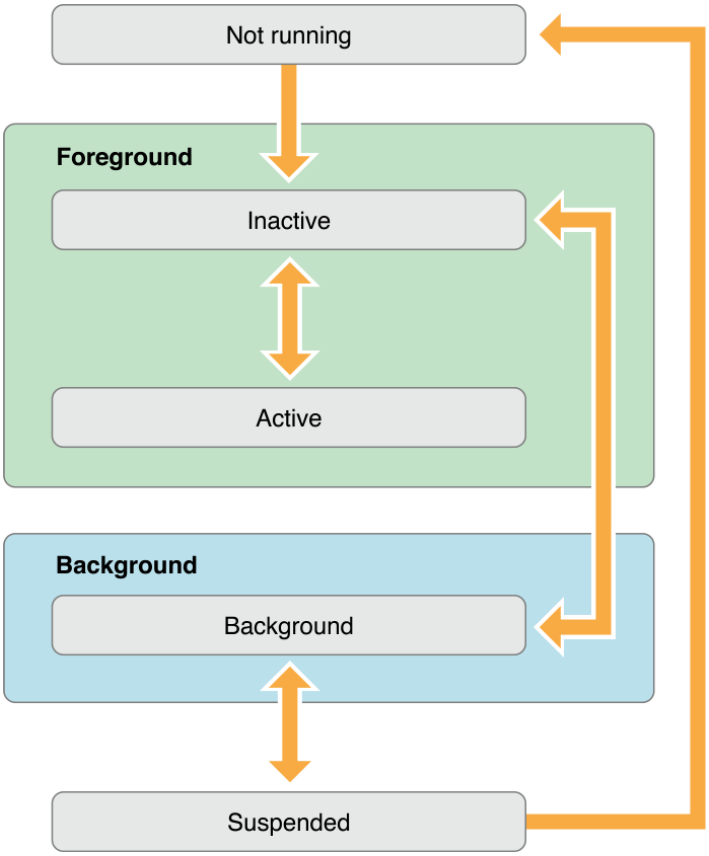

Figure 3-5: Activity states of an iOS application (Apple Inc., 2017)

These are the different activity state an application can be in:

- **Not running:** The application hasn't been started, or has been terminated by the system.
- **Inactive:** The application is running in the foreground, but isn't currently receiving any events. An application usually only has this state momentarily, when it is switching to another state.
- **Active:** The application is running in the foreground and receiving events. This is the normal state for applications running in the foreground.
- **Background:** The application is in the background and running code. Most applications are in this state only briefly when they are about to be suspended. Applications asking for extra running time can stay in this state for a longer duration.
- **Suspended:** Application is in the background and isn't running code. The system puts applications in this state automatically without notifying them beforehand. When suspended, an application will still be in the system memory, but without running any code.

3.2.2 Swift

Swift is a general-purpose programming language developed by Apple, designed for developing software for their hardware platforms. Swift was developed with the goal of replacing Objective C, which was the former dominant programming language for developing for Apple devices. Swift is designed to work with Apple's Cocoa and Cocoa Touch frameworks, and most of Objective C code written for Apple devices.

Swift 1.0 was launched 02.06.14 under a proprietary licence. At the launch of Swift 2.2, it was changed to Apache License 2.0, a tolerant open source licence.

Swift is focused on security and performance. The compiler is designed in a way that it tries to catch runtime errors during compilation.

Syntax

The syntax of Swift looks different from languages with a so-called C style syntax. Swift uses two distinct ways of declaring variables. One for constant, immutable variables and one for mutable variables. Swift also uses type deduction, meaning it infers what data type a variable has, often removing the need to specify it (Code 3-1).

Functions need to be declared with the word "func", and return type after function name and input parameters. Void functions are declared by omitting the return type. To call a function, it is often required to name the in parameters in the function call. This is necessary because there can exist functions with similar names, where the input values are the defining difference.

```
// Mutable variable with type deduction
var mutable = 10

// Immutable variable with type deduction
let immutable = 20

// Mutable vaiable with explicit type
var explicitType: Int32 = 30

// Immutable varible with type deduction from function
// NOTE: Named input parameters
let f = addFloat(a: 2.0, b: 3.0)

/* Function declaration
func functionName(inParam1: type, inParam2: type, ...) -> returnType {
 ...
}
*/
func addFloat(a: Float, b: Float) -> Float {
    return a + b
}

// Function without return type. Similar to void, in Java/C
func printMsg(msg: String) {
    print(msg)
}
```

Code 3-1: Declaration of variables and functions

Classes are defined in the same way as in languages like Java. Custom in Swift is to declare private variables with names starting with an underscore. This is usually accessed by a secondary variable with the same name – but without the underscore – and its getters and setters (Code 3-2).

```
// Simple class definition
class MyClass {
    // Private variable
    private var _myVariable: Int = 0
    // Computed getter and setter
    var myVariable: Int {
        get { return _myVariable }
        set { _myVariable = newValue }
    }
}
```

Code 3-2: Private variable with getter and setter

Unlike languages which uses ordinary member methods as getters and setters, Swift has what's called "computed properties". These doesn't store any information, and therefore has no impact of the size of the class instance. Computed properties add a getter and, optionally, a setter to set and retrieve other values indirectly (Code 3-3). These can contain computations to generate the new or retrieved value.

```
class MyClass {
    private var _myVariable: Int = 0
    // Computed property, getter and setter
    var myVariable: Int {
        set {
            if (newValue % 2) == 0 {
                _myVariable = newValue / 2
            } else {
                _myVariable = newValue * 3 + 1
            }
        }
        get { return _myVariable }
    }
}
```

Code 3-3: Private variable with computed getter and setter

Swift also has built-in security. New objects can't be defined as nil, as this will cause the compiler to report an error. There are however, still situations when nil is necessary. For this, Swift uses so-called *optionals*, where variables are explicitly defined with a question mark at the end. These variables are called optionals to indicate that it is currently not known whether the variable has a value or not. They need to be unwrapped before use. Unwrapping can be forced by setting an exclamation mark at the end of the variable whose value is retrieved. It can also be done using double question marks as a ternary operator, checking if it has a value or not, and responding accordingly (Code 3-4). A common, more advanced method is to unwrap an optional with an "else if" loop.

```
class MyClass {
    var myVariable: Int?
}

var c = MyClass()
// c.myVariable is nil, and 10 is printed
print(c.myVariable ?? 10)
c.myVariable = 99
// c.myVariable is now 99, and 99 is printed
print(c.myVariable ?? 10)
// tempVal is only given a value if c.myVariable has a value
if let tempVal = c.myVariable {
       print(tempcVal)
}
```

Code 3-4: Unwrapping an optional

3.2.3 Cocoa Touch

Cocoa Touch is a UI application development environment for building applications for iOS devices, developed by Apple, it includes the Foundation framework and UIKit framework. (Apple Inc., 2015) (Apple Inc., 2017)

Foundation

The Foundation framework provides an implementation for the root class *NSObject* which defines basic behavior for all Objective-C objects in the

Cocoa Touch environment. It also provides wrapper classes for basic types and primitives, such as strings, numbers and collections (arrays, dictionaries etc.) as well as functionality like file management and XML processing.
In Objective-C and later versions (<= 2) of Swift, Foundation functions and classes are prefixed with *NS*, for example *NSDate, NSString and NSDictionary,* however with in Swift 3, the *NS* prefix was dropped to comply with Swift naming conventions, for compatibility reasons prefixed classes and functions are still available, but it is recommended to drop the prefix in new projects.

UIKit
The UIKit framework is used for developing user interfaces for applications, and implements functionality and classes for UI specific behavior, such as text processing, drawing, image- and event-handling. It also implements UI elements like buttons, sliders and text fields.
As with Foundation, classes and functions that have the *NS* prefix are available from Swift 3 without the *NS* prefix.

3.2.4 CocoaPods

CocoaPods[1] is a dependency manager for Swift and Objective-C, specifically geared towards Xcode[2] projects, but it does also integrate with JetBrains' AppCode. The CocoaPods project is open-source, under the MIT License, originally developed by Eloy Durán and Fabio Pelosin, and while not being directly affiliated with nor sponsored by Apple, it has become the de facto standard dependency manager for Xcode and IOS projects. The project is still under active development and receives many contributions by the community.[3]

Podfile
CocoaPods manages dependencies through a project specific configuration file, called a *Podfile*, where the user a target project and *pod*(library) using a Podfile domain specific language. The pods themselves can be retrieved from multiple different sources, usually from Git repositories, however Subversion, Mercurial, HTTP and local project sources are also supported (Code 3-5).

```
1  source 'https://github.com/CocoaPods/Specs.git'
2  target 'Behavorial Mapper' do
3    pod 'GoogleMaps'
4    pod 'GooglePlaces'
5  end
```

Code 3-5: Podfile from Behaviour Mapper

[1] (CocoaPods, 2017)
[2] Apple's IDE for creating iOS and macOS applications
[3] (Unknown, 2017)

The *source* line specifies repository containing *Podspecs*, these are JSON files containing metadata about the different pods registered with the spec repository. The one in Code 3-6 is the official CocoaPods spec repository, if you wanted to use private/closed-source pods you would simply add a new *source* line with the link to the spec repository where those pods are registered (Code 3-6).

The *target* is the name of the project you wish to link the pods with, you may have multiple separate targets if your Xcode workspace contains multiple projects, if your project contains multiple bundles (similar to Java packages) you may also nest targets for specific bundles.

```
source 'https://github.com/CocoaPods/Specs.git'
# Spec repository for private pods
source 'url-to-repository'

target 'Behavorial Mapper' do
  pod 'GoogleMaps'
  pod 'GooglePlaces'
  # Add pod dependency for a specific bundle
  target 'Behavorial Mapper Tests' do
    pod 'PodName'
  end
end

# Separate project
target 'AnotherProject' do
  pod 'PodName'
end
```

Code 3-6: More complex Podfile with multiple spec sources, nested and separate targets

You may also specify specific version for a pod, either with just a version number, which specifies that your projects depend on that version only, or you can use logic operators. CocoaPods also supports what they call an "optimistic operator", which will let you specify dependencies of pod versions up to, but not including the next most significant version. If no version is specified, CocoaPods will use the latest version available (Code 3-7).

```
source 'https://github.com/CocoaPods/Specs.git'
target 'YourProject' do
  pod 'PodA', '0.9' # Depends on PodA version 0.9 and only 0.9
  pod 'PodB', '<= 1.0'
  pod 'PodC', '> 0.4'
  pod 'PodD', '~> 1.2.3' # Depends on PodD versions 1.2.3 and up to but not including 1.3.0
  pod 'PodE', '~> 1.2' # Depends on PodE versions 1.2 and up to but not including 2.0
  pod 'PodF', '~> 0' # Depends on PodF versions 0 and up, same as dropping the version specifier
end
```

Code 3-7: Example of pods with version requirements

3.2.5 Google Maps & Google Places

Google launched the Google Maps API[1] in June 2015, allowing developers to integrate the map service in their software.

The GoogleMaps pod-file[2] is the pod that supplies the API for displaying and interaction with the map itself, while the GooglePlaces pod adds additional functionality to let you add location searching. To use the API, the GoogleMaps and optionally GooglePlaces pod-file must be linked to your project[3] and an API key is also required, which can easily be acquired from the Google Developers webpage.[4]

API

The Google Maps and Google Places APIs are well documented and easy to use. Once you have your API key, and set up the workspace, register your key with the API, set up a *GMSCameraPosition*, generate a view from that camera position and assign it to your view (Code 3-8). Note that the Google Maps API key and the Google Places API key refers to the same key.[5]

```
1  import UIKit
2  import GoogleMaps
3
4  class YourViewController: UIViewController {
5    private var _myView = GMSMapView()
6
7    override func loadView() {
8      // Register your API key with the GoogleMaps API
9      GMSServices.provideAPIKey("your-api-key")
10     // and optionally GooglePlaces
11     GMSPlacesClient.provideAPIKey("your-api-key")
12
13     // Create a GMSCameraPosition that tells the map to display the
14     // coordinate 58.937595, 5.694519 (UIS) at zoom level 3
15     let camera = GMSCameraPosition.camera(withLatitude: 58.937595, longitude: 5.694519, zoom: 3.0)
16     // Create a view from the camera
17     let mapView = GMSMapView.map(withFrame: CGRect.zero, camera: camera)
18     // Set your view to the resulting Google Maps view
19     _myView = mapView
20   }
21 }
```

Code 3-8: A bare bones Google Maps API view setup

---

[1] Application Programming Interface
[2] See CocoaPods
[3] See Code 3-8: A bare bones Google Maps API view setup
[4] (Google Developers, 2017)
[5] (Google Developers, 2017)

The Google Places API, as it is used in the Behaviour Mapper project, uses Place Autocomplete, which is an autocomplete service in the Google Places API, that returns place predictions based on user input.
Instantiating the autocomplete view controller is done through a single function call, and it is important to remember to assign the delegates. Here "self" is used, meaning that the class this function belongs to, also implements the required delegate functions (Code 3-9).

```
158       @IBAction func searchLocation(_ sender: Any) {
159           let autoCompleteController = GMSAutocompleteViewController()
160           autoCompleteController.delegate = self
161
162           self.present(autoCompleteController, animated: true, completion: nil)
163       }
```

Code 3-9: Set up an IBAction to show the autocomplete view

To use the autocomplete service, the *GMSAutocompleteViewControllerDelegate* protocol must be implemented. The *GMSAutocompleteViewControllerDelegate* protocol contains three required functions that must be implemented. One which is called when the user selects a place from the autocomplete predictions (Code 3-10), one which handles user exiting from the autocomplete view without selection (Code 3-11) and one which handles errors (Code 3-12).

```
140     func viewController(_ viewController: GMSAutocompleteViewController, didAutocompleteWith place: GMSPlace) {
141         _camera = GMSCameraPosition.camera(withLatitude: place.coordinate.latitude,
142                                            longitude: place.coordinate.longitude,
143                                            zoom: 15)
144         self._mapView.camera = _camera
145         viewController.dismiss(animated: true, completion: nil)
146     }
```

Code 3-10: Handling user selection from autocomplete

```
153       func wasCancelled(_ viewController: GMSAutocompleteViewController) {
154           self.dismiss(animated: true, completion: nil)
155       }
```

Code 3-11: User exiting autocomplete view without place selection

```
148     func viewController(_ viewController: GMSAutocompleteViewController, didFailAutocompleteWithError error: Error) {
149         viewController.dismiss(animated: true, completion: nil)
150         print("Error: \(error)")
151     }
```

Code 3-12: Autocomplete view error handling

3.3 Design Patterns

In software development, design patterns are general, reusable solutions to common problems in a given context. It's not a finished design that can be directly applied to a program, but rather a description or template for how a problem can be solved. In practice, they work as a guide for how to write code that is both easy to understand and easy to reuse. It also contributes to create code that avoids hard coupling between elements. That means code where individual parts more easily can be replaced, without having to make big changes in the other parts of the application. In this assignment, several design patterns were used. The most significant ones will be explained in this chapter.

3.3.1 Model-View-Controller

A very common design pattern for applications with user interfaces is Model-View-Controller – MVC (Figure 3-6). It splits an application into three connected parts to separate internal representation of information from the way it is presented to and used by the applications user. MVC splits these elements to allow efficient code reuse and parallel development (Figure 3-7).

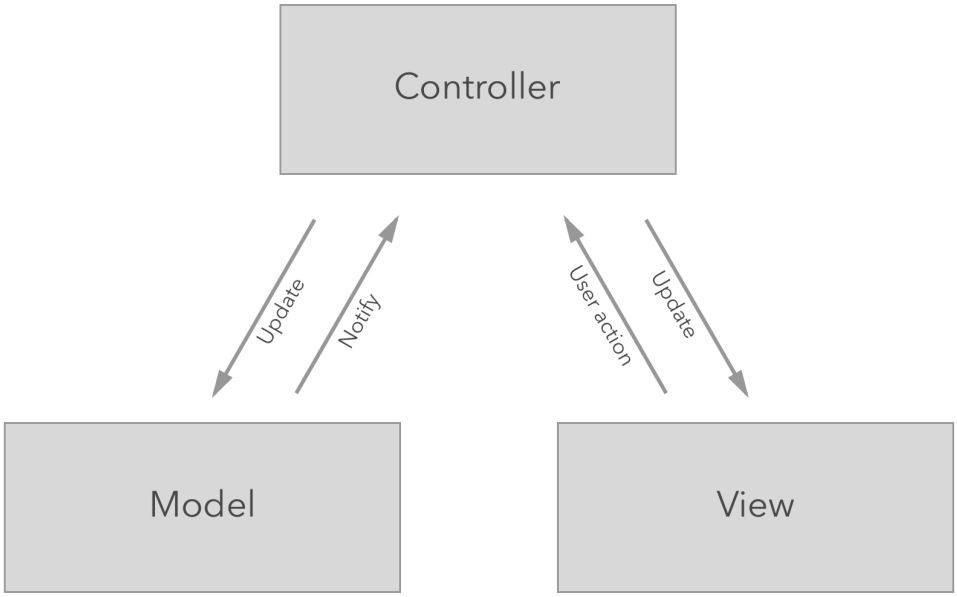

Figure 3-6: The model-view-controller relationship

Traditionally, the 3 elements have the following roles:

- Model: Contains data and corresponding rules and logic.
- View: a visual representation of information.
- Controller: Module which accepts user inputs and sends instructions to model and view objects in response to these.

In general, this means that all information for the user is displayed in a view object. This information is prepared by a controller, which retrieves relevant data from a model object, before sending it to the view. The information shown might just be a small part of the data in the model object. When a user performs an action of the view– like pressing the screen – it will notify the responsible controller. The controller then notifies a model which will make the necessary changes. This could result in the information shown in the view changing, or maybe the view now displays another part of the data from the model.

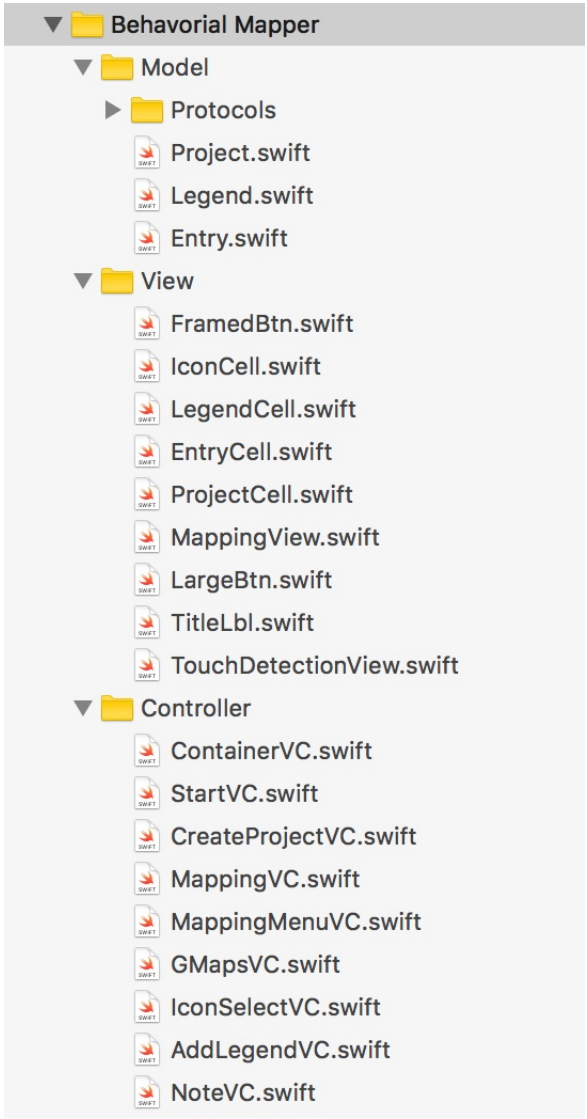

Figure 3-7: File structure based on the MVC pattern

An advantage of this structure is that it allows for much code reuse. The same – or a similar – view can be used in different parts of the application, to show different data. This is possible because a view only determines how data should be shown, irrelevant of what the data is. Another possibility is having a model being presented by different views. Some information might be displayed differently depending on where in the application, the user

currently is. At one place, it might make sense for a detailed view of the information, and a more concise, compact view somewhere else in the user interface.

MVC also allows developers to work in parallel on different parts of an application without affecting or blocking each other. A group of developers might be split into back-end and front-end development. The back-end team can work on a model object without needing the user interface being completed. At the same time the front-end team can design user interface elements without needing a finished model object to work with.

3.3.2 Decorator

The decorator pattern is a dynamic way to behaviour and responsibility to existing objects without changing the structure of these. This is an alternative to subclassing where you modify a class by wrapping it in another object. In Swift there are 2 typical implementations of this pattern: extension and delegation.

Extension

Writing an extension is a method for creating new functionality in an existing class or structure, without creating a separate subclass of it. It is even possible to create extensions for code without having access to it. This means one can add functions to classes that are part of iOS' own API (Code 3-13). It is not however, possible to overwrite existing functionality with an extension.

```
extension UIColor {
    class func fromHex(hex: Int) -> UIColor {
        let red: CGFloat = CGFloat((hex >> 16) & 0xFF) / 255.0
        let green: CGFloat = CGFloat((hex >> 8) & 0xFF) / 255.0
        let blue: CGFloat = CGFloat(hex & 0xFF) / 255.0
        return UIColor.init(red: red, green: green, blue: blue, alpha: 1.0)
    }
}
```

Code 3-13: An extension of the UIKit class UIColor (Extensions.swift)

New functions added with an extension during compilation, can be called like normal functions of the class that has been extended (Code 3-14).

```
static let backgroundPrimary = UIColor.fromHex(hex: 0x303030)
```

Code 3-14: Calling a function added in extension (Style.swift)

Delegation

Delegation is a mechanism to make an object act on behalf of, or in coordination with, another object. Using delegation allows separation between objects with different roles in an application. This way avoids hard coupling between elements that ideally are separate.

In Swift, delegation works by an object having a defined protocol of functions. This is in practice a list of function signatures – an interface, without any restrictions in how they are implemented (Code 3-15). Another object can then define these functions and later function as a delegate for the former object (Code 3-16). This former object can then call these objects through its delegate. When called, the delegate will perform the functions per its own implementation.

```
protocol MenuContainerDelegate {
    func closeMenu()
    func exitProject()
}
```

Code 3-15: Delegation protocol (MappingMenuVC.swift)

A regular area of use is user interface elements. They often use delegates to forward user actions. A controller object might be set as the delegate, letting it perform functions based on calls from the user interface element. This also lets it pass variables to its delegate.

```
extension ContainerVC: MenuContainerDelegate {
    func closeMenu() {
        hideMenu()
    }

    func exitProject() {
        killMenu()
        loadVC(ofType: .start)
        presentVC(ofType: .start)
    }
}
```

Code 3-16: Implementation of delegate functions (ContainerVC.swift)

4 Construction

This chapter will describe the design of the application. It is divided into two subchapters. One describing the presentation layer, the main view controllers and how these function. The second chapter will detail the structure of the workspace. It will describe how files, classes and methods are organized and explain the implementation of data storage.

4.1 Application Structure

The application consists of 4 main view controllers. These are called *ContainerVC*, *StartVC*, *CreateProjectVC* and *MappingVC*. *ContainerVC* is a hidden view controller which serves as the backbone of the application. *ContainerVC* is always running in the background and will continually control which of the other 3 view controllers that's currently being presented. *ContainerVC* can load, terminate and present the required view controllers, as required by the application state or user interaction (Figure 4-1).

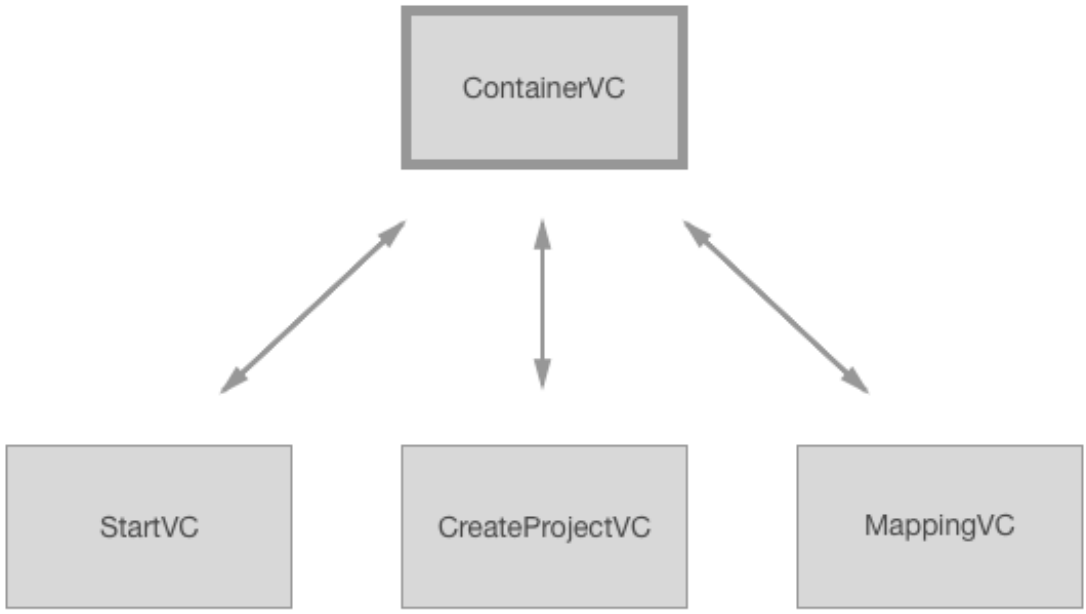

Figure 4-1: View controller hierarchy

All communication between *ContainerVC* and the child view controllers is handled through protocols and delegate methods. When the application is about to transition to a new view controller, *ContainerVC* creates a new instance of that specific view controller and sets itself as the delegate at the same time (Code 4-1). It will always function as the delegate for the other 3 view controllers. Then it presents the newly created view controller and terminates the previous one, saving memory (Code 4-2).

```
func loadVC(ofType: VCType) {
    switch ofType {
    case .start:
        if (startVC == nil) {
            startVC = UIStoryboard.startVC()
            startVC?.delegate = self
        }
```

Code 4-2: Loading a view controller (ContainerVC.swift)

```
func deloadVC() {
   switch currentVC {
       case .start:
          self.startVC!.view.removeFromSuperview()
          self.startVC = nil
       }
```

Code 4-1: Terminating a view controller (ContainerVC.swift)

This structure avoids hard coupling between the different view controllers, which results in a more scalable application. This makes it easy to add new view controllers, accessing them from several situations or using one for different purposes.

4.1.1 Start Screen

When the application is launched, *ContainerVC* will load and present *StartVC*, which acts as the starting screen (Figure 4-2). This is where the user can choose to start a new project, load an existing one, create a new one based on a stored project or delete any of the stored projects. These are the main functions of this screen, and are all easily available as large buttons with descriptions, at the lower left area.

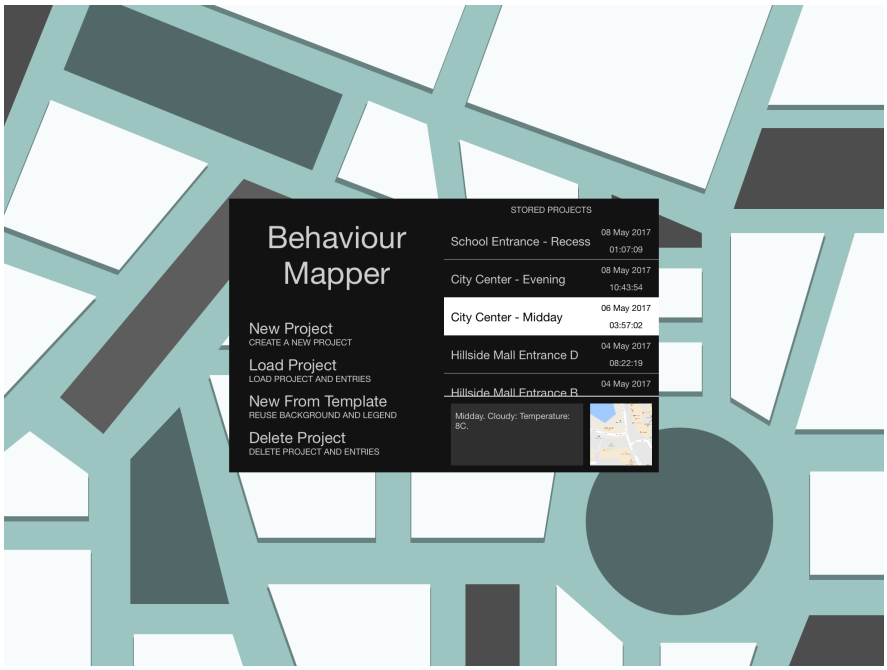

Figure 4-2: Start screen

When loaded, the view controller will check if there are any previously stored projects. If so, it will load all the stored *Project*[1] objects and display them in a table view list to the right (Code 4-3). Each cell in the list displays the project's name and the time it was last edited. Tapping on any one of them will display the chosen project's background image and project notes at the lower part of the screen, making it easy to discern which project is currently selected.

```
func initStoredProjectsTableView() {
    storedProjectsTableView.dataSource = self
    storedProjectsTableView.delegate = self
    _storedProjects = getProjectFiles()
    if _storedProjects == nil {
        self._storedProjects = [String]()
    }
}
```

Code 4-3: Populating the table view with stored projects (StartVC.swift)

If the user presses a button that isn't "Create New Project", *StartVC* will check if a project is currently selected and display warning message if it isn't.

The four available actions:

- **Create New Project:** The *ContainerVC* method *createNewProject()* is called, and *ContainerVC* switches from *StartVC* to *CreateProjectVC*, where the user can create a new project.
- **Load Project:** The *ContainerVC* method *loadProject(fromProject: Project)* is called. This method gets passed the loaded Project object as an argument. ContainerVC then loads MappingVC, which is assigned the loaded project. MappingVC then populates its views with the data in the Project object: entries, background, legends and notes.
- **New From Template:** The *ContainerVC* method *loadFromTemplate(fromProject: Project)* is called (Code 4-4). *ContainerVC* then loads *CreateProjectVC*, and assigns it the loaded Project object. *CreateProjectVC* is then presented, with the background and legends from the Project object already present. Allowing the user to create a new project based on a previous location. Useful if they want to gather data at different times.
- **Delete Project:** The selected project and all contained data will be deleted. *StartVC* will then clear the image and note from the bottom part of the screen, before reloading the table view.

```
@IBAction func newFromTemplatePressed(_ sender: Any) {
    if let loadedProject = Project(projectName: _selectedProject) {
       delegate?.loadFromTemplate(fromProject: loadedProject)
    } else {
       displayMessage(title: NO_PROJECT_SELECTED_TITLE, message:
NO_PROJECT_SELECTED_MSG, self: self)
    }
}
```

Code 4-4: Call delegate method to create project from template (StartVC.swift)

[1] See Project under Model

4.1.2 Create Project Screen

When the user chooses to create a new project, either from scratch or based on a previous one, ContainerVC will load and present *CreateProjectVC* (Figure 4-3). This is where the user configures the new project before starting the mapping process. The project's title, background, notes and legends, are all configured in *CreateProjectVC*. At the bottom, there are two buttons, Cancel and Create. The first one cancels the project and returns the user to *StartVC*, and the other gathers all entered data and creates a new *Project* object, before transitioning to *MappingVC*.

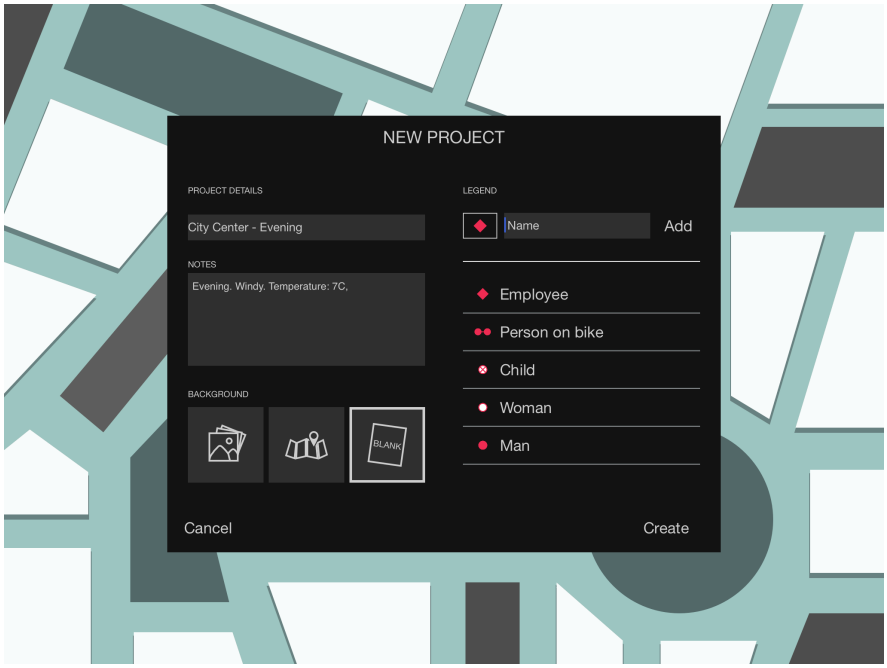

Figure 4-3: Create project screen

Name and notes:

The project's name is entered in a *UITextField* object, and the notes are entered in a *UITextView* object. As the name is required data, *CreateProjectVC* will check if there is a one entered, before a project can be created.

Legend:
The right side of the menu is dedicated to the creation of symbols for the project. The top part is where the user creates new symbols. It consists of a *UIButton* showing the currently selected icon, a *UITextField* for the name, and another button for creating a new symbol with the selected icon and entered name. Hitting the icon button will initialize the view controller *IconSelectVC*, which is then presented modally over the current view controller (Figure 4-4).

Figure 4-4: Icon selection view controller

IconSelectVC is a subclass of the view controller type *UICollectionViewController*. A collection view is useful for displaying data in a grid pattern. In this case, it is a grid of icons that can be selected for the symbol. When one is selected, *IconSelectVC* will check to see what class the presenting view controller is. As *IconSelectVC* is called by two different view controllers in the application, it reacts different depending on which it currently being presented by (Code 4-5). After having determined that the presenting view controller is CreateProjectVC, it calls the function *enterLegendIcon(iconId: Int)* and passes in the index of the selected icon as the argument.

```
override func collectionView(_ collectionView: UICollectionView, didSelectItemAt indexPath:
IndexPath) {
    if let vc = self.presentingViewController as? CreateProjectVC {
        vc.enterLegendIcon(iconId: indexPath.row)
    } else if let vc = self.presentingViewController as? AddLegendVC {
        vc.enterLegendIcon(iconId: indexPath.row)
    }
    dismiss(animated: true, completion: nil)
}
```

Code 4-5: Detecting which view controller is presenting (IconSelectVC.swift)

Hitting the "Add" button then creates a new Legend object, which it adds to an array. The table view below will then be updated and show all the objects in the array. If the user wishes to delete one of the Legend objects, it can be done by sliding the table view cell to the left, and hitting the "Delete" option which will slide into view (Figure 4-5).

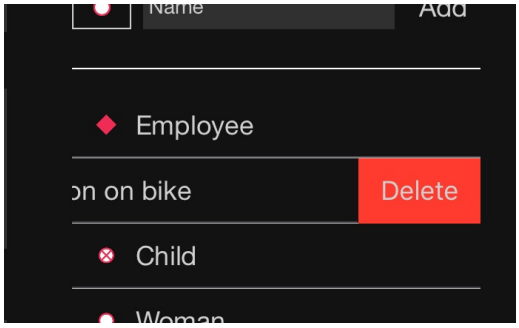

Figure 4-5: Delete action on Legend cell

Background:
There are 3 options for choosing what background to use in the mapping process. The default option is a blank, white background. Another option is to upload a custom image. The last option is to create a background using Google Maps. These are all illustrated as individual buttons on the lower left side: *loadPictureButton*, *createMapButton*, *blankBackgroundButton*. The button corresponding to the current selection, is highlighted with a bright border (Figure 4-6).

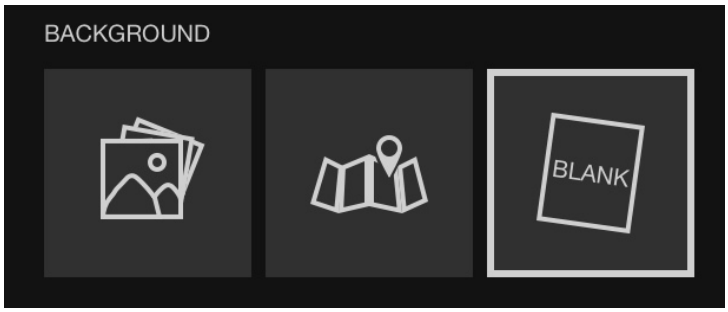

Figure 4-6: Buttons for selecting background

Pressing *loadPictureButton* will initialize a *UIImagePickerController*. The image picker is a part of the Cocoa Touch framework, used for accessing image files stored outside of the application.

Pressing *createMapButton* will initialize the custom view controller GMapsVC, which is then presented over *CreateProjectVC*, taking up the entire screen (Figure 4-7). GMapsVC uses the Google Maps SDK to display an interactive map. Google Maps SDK is an external library installed with a *CocoaPod* file created by Google themselves.[1]

[1] See Google Maps & Google Places

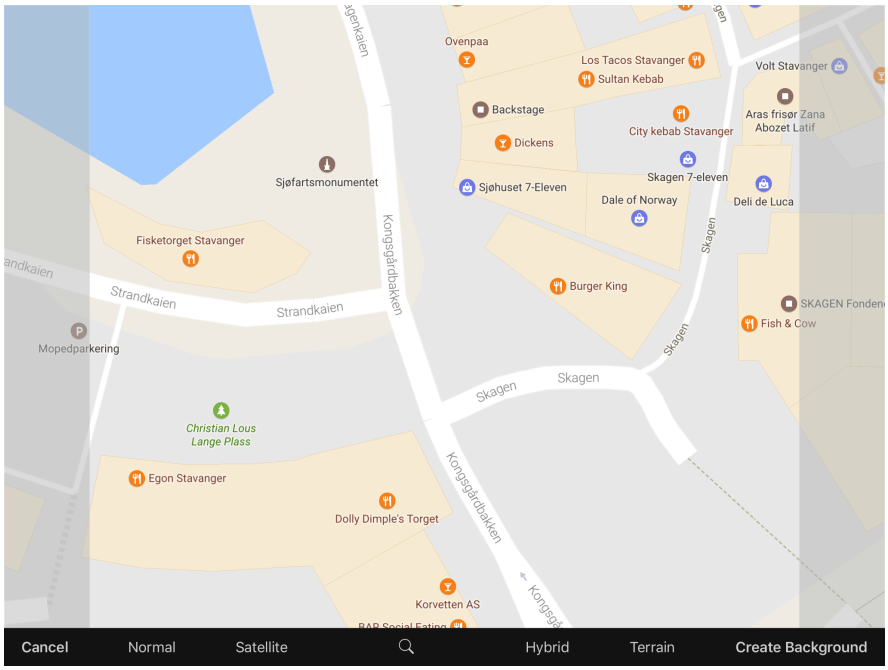

Figure 4-7: Google Maps screen

Upon loading, *GMapsVC* calls the function *GMSMapView.map(withFrame: CGRect, camera: CMSCameraPosition),* which returns a view of the interactive map. This is inserted in the main view of the view controller. The camera variable determines the position of the map. When initialized, it's given the position of the University of Stavanger, making that the starting position of the map view (Code 4-6).

```
private var _camera = GMSCameraPosition.camera(withLatitude: 58.938100,
longitude: 5.693730, zoom: 15)
```

Code 4-6: Initializing Google Maps view with starting position (GMapsVC.swift)

The bottom row consists of toolbar with buttons. One cancels the process, dismissing the view controller and returning the user to *CreateProjectVC*. There are 4 buttons to select what type of map to use: Normal, satellite, hybrid and terrain. Pressing one immediately updates the map view to reflect the chosen mode (Code 4-7).

```
func setMapType(withSender sender: UIBarButtonItem) {
    _mapView.mapType = GMSMapViewType(rawValue: UInt(sender.tag)) ?? GMSMapViewType.normal
}
```

Code 4-7: Handler for map type buttons (GMapsVC.swift)

Pressing "Create Background" will take a snapshot of the current area (Code 4-8). The toolbar gets hidden during the image creation. The *UIView* function *snapshot()* has been extended to take in an argument of *CGRect*. The extension lets it take a snapshot of only the chosen area of the view. This way it produces an image that has the same dimensions as the mapping view it

will be used in later.[1] The image gets passed to *CreateProjectVC* through a delegate function.
CreateProjectVC then runs a function to update the image buttons, setting the image as the icon for the chosen button, giving it a border, and resetting the two other buttons.

```
func takeScreenshot() {
    _toolBar.isHidden = true
    let image = self.view.snapshot(of: CGRect(x: 100, y: 0, width: 824, height: 768))
    _toolBar.isHidden = false
    delegate?.getScreenShot(image: image!)
    dismiss(animated: true, completion: nil)
}
```

Code 4-8: Creating an image from a specified area of a view (GMapsVC.swift)

4.1.3 Mapping Screen

The mapping screen is loaded if the user chooses to resume a previous project from *StartVC*, or completes the setup of a new one in *CreateProjectVC*. In either case, *ContainerVC* will load and present the *MappingVC* view controller, passing in the Project object as an argument (Figure 4-8). When the view loads, *MappingVC* sets the background to the mapping view and populates the legend array. At the same time, it checks if the Project object has any previously recorded entries. If so it runs through the array of entries, and draws each of them on the mapping view.

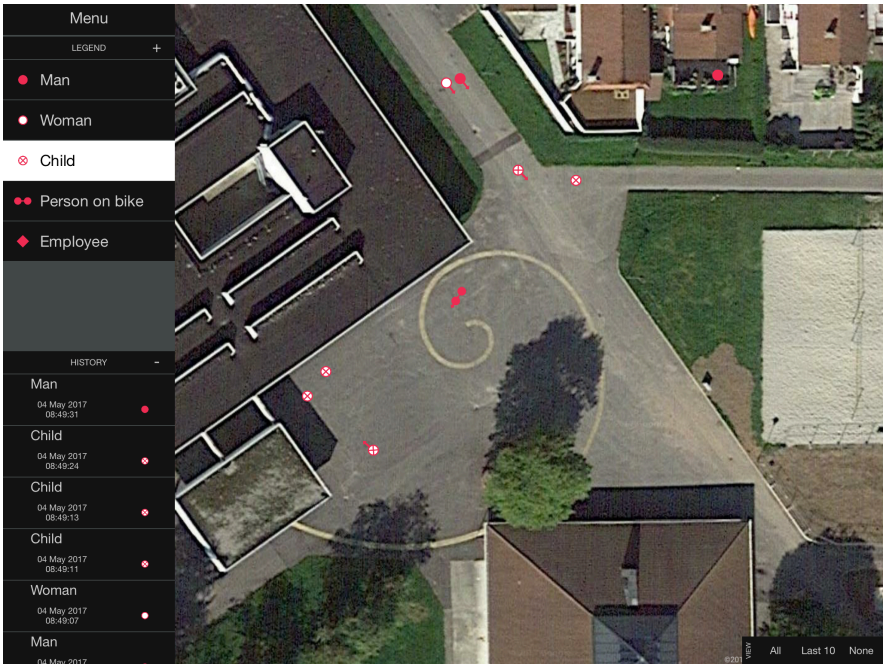

Figure 4-8: Mapping screen

[1] See Code 4-14: Extension for taking snapshot of only a selection of a view (Extensions.swift)

This screen consists of two main parts: The left side has a menu button at the very top, and two table views below it. These table views display the different symbols and recorded entries, respectively. The right side is the mapping view, which shows the chosen background and all entries. Touching the view generates a new entry where the user touched. Touching and releasing creates an entry icon on that spot, holding and dragging will additionally create an arrow pointing in the direction the user moved their finger. At the bottom right corner of the mapping view is a menu allowing the user to filter how many entries being displayed at any given time: The last 10, all entries or none of them. This allows the mapping view to be easily readable no matter how many entries are recorded.

Mapping View

The mapping view is a custom subclass of *UIView*. This subclass has overwritten the functions for detecting touches, and instead of handling them directly, does so with a delegate function located in *MappingVC*. These functions are for when the touch started, moved or ended. The first time a touch is detected, the function *mappingViewTouchBegan()* is called. It draws a new entry on the mapping view. If the same touch is continue and moves, *mappingViewTouchMoved()*, is triggered. That function determines whether the move is greater than a predetermined deadzone – to account for accidental movement. If so, the arrow is made visible, and continually rotated in the direction of the touch (Code 4-9). The final function, *mappingViewTouchEnded*(), is triggered when the touch ends. In other words when the user removes the finger from the view area. When that happens, the entry is added to the array of entries in the Project object. At the same time, the associated table view is reloaded to include this latest entry.

```
func mappingViewTouchMoved(sender: MappingView, touches: Set<UITouch>) {
    if(touchesMovedDeadZone == 0) {
        if(_arrowIcon.isHidden) {
            _arrowIcon.isHidden = false
        }
        if let touch = touches.first {
            let newPos = touch.location(in: mappingView)
            let mPoint = bearingPoint(point0: _centerPos, point1: newPos)
            _angleInDegrees = pointToDegrees(x: mPoint.x, y: mPoint.y)
            rotateImage(imageId: tagNumber, angleToRotate: _angleInDegrees)
        }
    } else {
        touchesMovedDeadZone -= 1
    }
}
```

Code 4-9: Delegate function to handle movement on mapping view (MappingVC.swift)

Each entry is drawn with a function called *createEntryIcon()*. This function is called both when loading a project with recorded entries, and each time the user creates a new one by touching the mapping view. It creates two separate *UIImageView* objects for each entry: One for the main icon, and one for the arrow, indicating direction. The function is passed coordinates, an angle in degrees, an integer to determine icon type, what view to draw in

and an id to tag the image views with. This id is used to manipulate the images later, either when deleting them or hiding them from the view. The angle determines whether or not the arrow should be visible, and if so what angle it is to be drawn at. The range of acceptable angles is -180 to 180. The default angle is 999, which is outside of the approved range, and indicates that the arrow should remain hidden (Code 4-10). The coordinates are relative to the mapping view, which has a dimension of 824 by 768 points. These simply tell where the centre of the drawn image view should be placed. The entry object will have icon and name corresponding to the type of symbol chosen in the legend table view.

```
func createEntryIcon(xPos: CGFloat, yPos: CGFloat, targetView: UIView, angleInDegrees:
CGFloat, tagId: Int, icon: Int) {
    let _centerIcon = UIImageView(frame: CGRect(x: xPos - (CENTER_ICON_SIZE/2), y: yPos -
(CENTER_ICON_SIZE/2), width: (CENTER_ICON_SIZE), height: CGFloat(CENTER_ICON_SIZE)))
    _centerIcon.image = UIImage(named: "entryIcon\(icon)")
    _arrowIcon = UIImageView(frame: CGRect(x: xPos - (ARROW_ICON_SIZE/2), y: yPos -
(ARROW_ICON_SIZE/2), width: (ARROW_ICON_SIZE), height: (ARROW_ICON_SIZE)))
    _arrowIcon.image = UIImage(named: "arrow")
    _centerIcon.tag = tagId
    _arrowIcon.tag = tagId + 1
    targetView.addSubview(_arrowIcon)
    mappingView.addSubview(_centerIcon)
    if (angleInDegrees == 999) {
        _arrowIcon.isHidden = true
    } else {
        _arrowIcon.isHidden = false
        rotateImage(imageId: _centerIcon.tag, angleToRotate: angleInDegrees)
    }
    targetView.addSubview(_arrowIcon)
    mappingView.addSubview(_centerIcon)
}
```

Code 4-10: Function for drawing an entry on the mapping view (MappingVC.swift)

Entry Filter Menu

Three buttons are formed in a row in the lower right corner of the mapping view (Figure 4-9). These buttons enable filtering of the visible entries. The user can choose to view all entries, only the last 10 or none of the recorded entries. The filtering is done simply by looping through all or the last 10 entries in the array, and setting them visible or hidden. It also checks their angle value, to determine if their arrow should change its visibility.

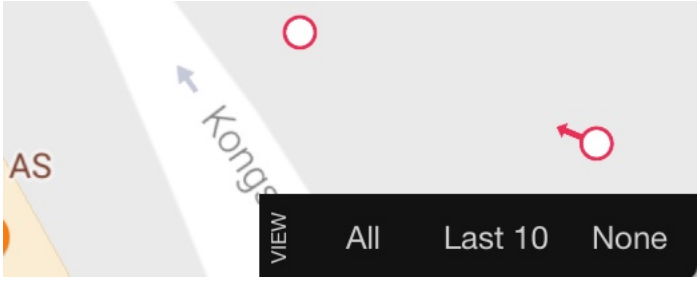

Figure 4-9: Filter buttons

Legend Table View
All the available entries are presented in table view on the left side of the screen. Each cell has both the name and icon of each symbol. By default, the top cell is highlighted, and the corresponding symbol is selected. Meaning that the next entry created on the mapping view, will have the same icon and name. Choosing a symbol is as simple as tapping on the desired cell. Doing so will highlight it, and set the variable *selectedLegend* to the same type.

The highlighting happens with a function in the *LegendCell* class, which is a custom subclass of *UITableViewCell*. During the loading of the table view, each cell is generated with the table view function *cellForRowAt*. It generates cells for each row in the table view. For each new cell, it checks if the index of the row it is currently populating, is equal to the array index of the attribute *selectedLegend*. If it is, it calls the *styleHightlight()* function on the new cell, changing its background and font color. Otherwise, it calls *styleNormal()* on the cell. Both functions belong to the *LegendCell* class.

At the top of the table view is a button with a "+" icon, for adding new symbols to the legend array. This allows the user to continually add new symbols without interrupting the mapping process. Pressing the button creates an instance of the *AddLegendVC* view controller, and presents it modally over the current view controller (Figure 4-10). This has the same legend fields found in *CreateProjectVC*: a text field for the name, a button for the symbols, and another button to add it to the array.

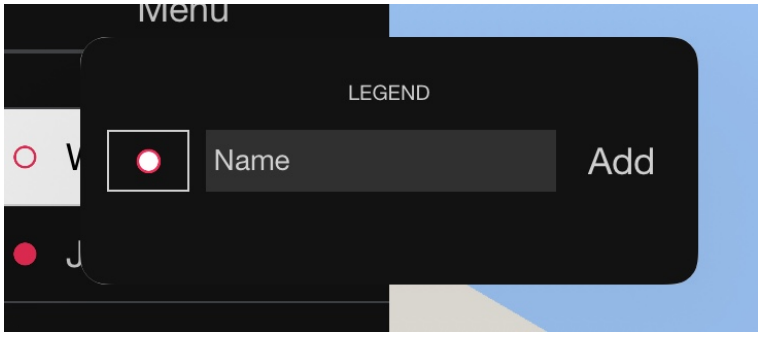

Figure 4-10: View controller for adding symbols

History Table View
Whenever an entry is created, a cell representing it is added to the history table view. Each cell contains the symbol name, time of entry and the icon of the entry. These are sorted by time created, with the newest on top. Tapping on a cell will generate an animation around the corresponding entry in the mapping view, making it easy to know which entry the cell belongs to (Figure 4-11).

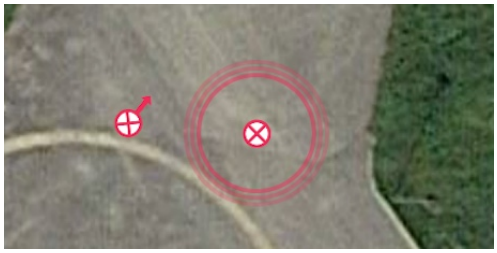

Figure 4-11: Snapshot of animation highlighting entry

These cells can also slide to the left, to reveal to actions: "Note" and "Delete". Pressing "Delete", simply deletes the cell and the corresponding entry in the entry array and the image view objects in the mapping view. Hitting "Note" will present the *NoteVC* view controller, where the user can enter a note for that entry. When loaded, *NoteVC* will display the already stored note, if one exists. If an entry has been given a note, its cell will have a small icon to indicate that (Figure 4-12). This icon is given to it with the *configureCell()* function located in the *EntryCell* class. The table view will call *configureCell()* on each cell it generates. The function makes the note icon visible if the note attribute of the Entry object contains any text.

Figure 4-12: Recorded entry with note

Menu

At the top left is a large button used to reveal a menu. The menu is a separate view controller – called MappingMenuVC – that is drawn beneath MappingVC. When the menu is loaded, ContainerVC moves MappingVC 210 points to the right, revealing the menu beneath it with a quick animation (Figure 4-13). This keeps most of the MappingVC still in view. At the same time a new instance of TouchDetecionView is placed on top of MappingVC. If pressed, it will call a delegate method in ContainerVC, which brings MappingVC back, and terminates the menu view controller.

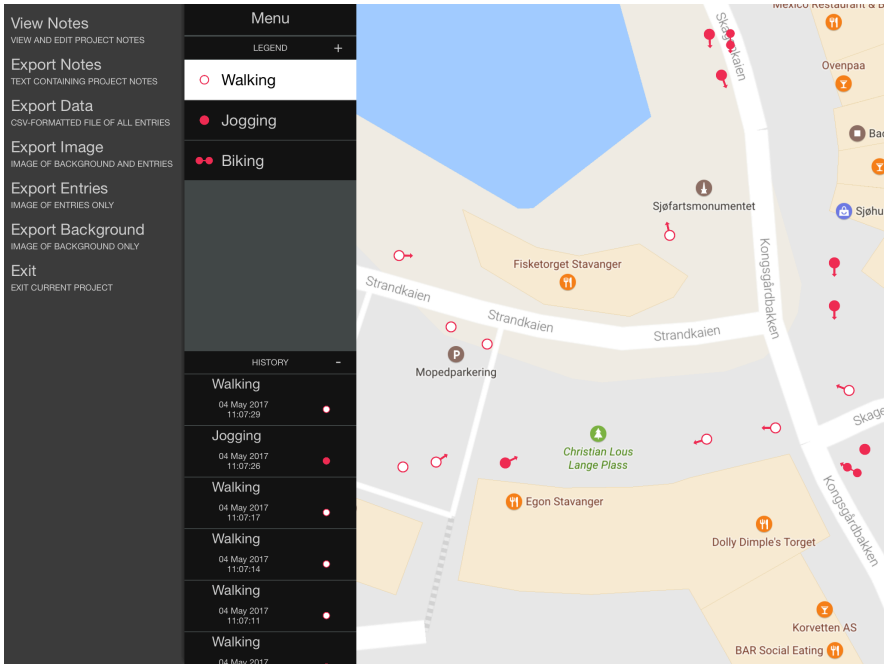

Figure 4-13: Menu shown behind mapping screen

The menu has the following actions:

- View Notes: Present *NoteVC* which lets the user edit the project's note.
- Export Notes: Uses an instance of *UIActivityController* to import the project note as text into a compatible application. This lets the user send it in an email, post it with a social media application, edit it in a document application etc.
- Export Data: Uses a custom function to generate a CSV file based on the recorded entries. It uses an instance of *UIDocumentController* to import the CSV file into a compatible application (Figure 4-14). Examples include sending it in an email, editing it in a spreadsheet application (Figure 4-15), storing it in Apple's iCloud service etc.
- Export Image: Takes a screenshot of the mapping view with all entries. Uses an instance of *UIActivityController* to import the image into a compatible application. Examples include sending it in an email, editing it in a photo editor etc.
- Export Entries: Same as "Export Image", except it hides the background and creates an image of just the entries on a white background.
- Export Background: Same as "Export Image", except it hides the entries and creates an image of just the background.
- Exit: Returns the user to *StartVC*.

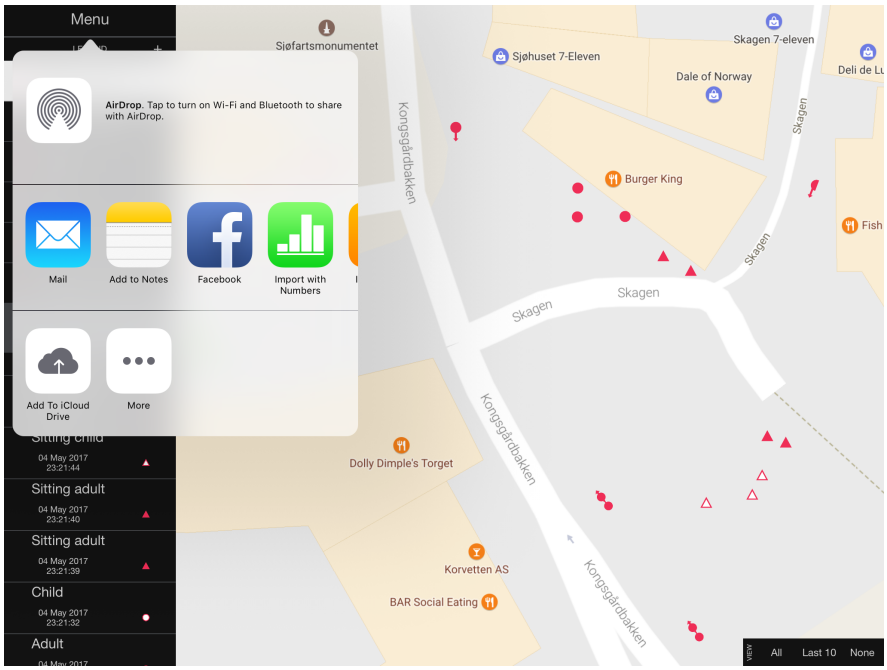

Figure 4-14: Dialogue window for data export

Time	Position	Angle in degrees	Note	Type	Icon
04/05/2017, 23:21	(688.0, 500.5)	999.0		Sitting adult	16
04/05/2017, 23:21	(709.5, 508.0)	999.0		Sitting adult	16
04/05/2017, 23:21	(682.0, 546.0)	999.0		Sitting child	15
04/05/2017, 23:21	(617.0, 578.0)	999.0		Sitting child	15
04/05/2017, 23:21	(670.5, 568.5)	999.0		Sitting child	15
04/05/2017, 23:21	(326.0, 142.0)	-90.0000025044782		Adult	1
04/05/2017, 23:21	(166.0, 187.0)	5.27389598979267	Stopped and talked on phone	Adult	1
04/05/2017, 23:21	(44.0, 290.0)	10.3663223910746		Person on bike	9
04/05/2017, 23:21	(605.0, 729.0)	125.469878373839		Person on bike	9
04/05/2017, 23:21	(499.0, 577.0)	129.05848708452		Person on bike	9
04/05/2017, 23:21	(743.0, 208.5)	-110.840917737803	Left the same way	Person running	6
04/05/2017, 23:21	(142.0, 315.0)	18.2324898949591		Person running	6
04/05/2017, 23:21	(467.5, 212.5)	999.0		Adult	1
04/05/2017, 23:21	(522.5, 245.5)	999.0		Adult	1
04/05/2017, 23:21	(467.0, 246.0)	999.0		Adult	1
04/05/2017, 23:21	(129.0, 405.0)	999.0		Child	0
04/05/2017, 23:21	(567.0, 291.5)	999.0		Sitting adult	16
04/05/2017, 23:21	(599.0, 308.5)	999.0		Sitting adult	16
04/05/2017, 23:21	(130.5, 207.0)	999.0		Sitting child	15
04/05/2017, 23:21	(92.5, 218.0)	999.0		Sitting adult	16

Figure 4-15: Data imported to spreadsheet software

4.2 Workspace Structure

The general structure of the workplace is designed after the Model-View-Controller design pattern. The classes are divided into folders named after that pattern: *Model*, *View* and *Controller*. A fourth folder, called *Util,* contains global functions, extensions and constants. This chapter will detail these folders and how data serialization is handled.

4.2.1 Model

The model directory contains all classes intended to hold and manipulate raw data. In addition to the attributes detailed below, each model class has a custom *init()* function, which takes a dictionary object as an argument. This is used to recreate objects when loading stored data. A subfolder called Protocols contains classes related to data serialization.[1]

Legend

The class Legend represents the symbols used for the entries in the mapping environment. Each Legend object contains a name for that symbol, and an integer indicating which icon belongs to it.

Entry

The class Entry represents a single recorded entry made during the mapping process. An Entry object contains a Legend variable – determining name and icon. In addition, there are the following variables:

- time: A *Date* object indicating when the entry was created.
- position: A *CGPoint* object including the relative coordinates of the Entry object's position in the mapping view.
- angleInDegrees: A *CGFloat* value representing the direction the entry is pointed. Its starting value is 999. If the user creates an entry with a direction, this is overwritten with a value between -180 and 180.
- note: A *String* value that defaults to an empty string. Can be overwritten with specific details about the entry.
- tagId: An *Int* value used to tag the *UIImageView* objects that represent the entry. This tag makes it possible to edit, hide or delete the view objects after their creation.

Project

The class Project holds all the data related to a mapping project. It has a Legend array, containing all the symbols used in the project, an Entry array for all the recorded entries, and in addition it holds the following variables:

- name: A *String* representing the project's name.
- created: A *Date* object for when it was created.
- lastSaved: A *Date* object for when it was last edited. Used for sorting the stored projects in *StartVC*.
- note: A String where the user can enter details about the project.

[1] See Data Serialization

- background: A String with the identifier of the project's background. Used for retrieving the stored background.
- backgroundType: An *Int* value to determine if the background is either created from *GMapsVC*, uploaded by the user, or simply a blank background.

4.2.2 View

The view folder contains classes of view objects. These are custom subclasses created to give that view object a special style or functionality.

- **FramedBtn, LargeBtn, TitleLbl:** These classes set styling attributes to give all instances of these classes the same, specific appearance, giving the application a uniform look. This also makes it much easier to change the style globally, and not having to change values of each individual instance.
- **IconCell, LegendCell, ProjectCell, EntryCell:** These all have a custom function *configureCell(),* which is called upon creation of each cell. The function determines their appearance and how to apply data to its contained view objects. Examples like giving the cell a particular icon image, or applying text to a contained label.
- **MappingView, TouchDetectionView:** Custom subclasses of *UIView* where the touch event functions have been overwritten. These now call delegate functions in *MappingVC* to handle the touch events for that view uniquely.

4.2.3 Controller

The controller folder contains all the view controllers. Some represent full screens of the applications, and others are used within other view controllers.

- **ContainerVC:** A non-visible view controller used to load, terminate and present the three main view controllers: *StartVC*, *CreateProjectVC* and *MappingVC*.
- **StartVC:** Contains everything related to the start screen where the user chooses create, delete or load a project.
- **CreateProjectVC:** Controls everything related to setting the details of the new project.
- **MappingVC:** Used for the actual mapping process.
- **MappingMenuVC:** Contains a stack of menu options that can be selected from the mapping screen.
- **GMapsVC:** Presented when the user chooses to create a background using Google Maps.

4.2.4 Util

The *util* (short for utility) folder is home for constants, functions and other features that are used globally in the application.

Constants

This class is a collection of constants used throughout the app. The reason for keeping them together is to make it easier to adjust these without having to find the – often numerous – places where they're implemented. Examples of constants are sizes of visual elements (Code 4-11), text strings that are communicated to the user and headers used when exporting the project to a csv file.

```
// MappingVC
let VIEW_ALL = 1
let VIEW_10 = 2
let VIEW_NONE = 3

let LEGEND_TABLEVIEW_CELL_HEIGHT = 47
let ENTRY_TABLEVIEW_CELL_HEIGHT = 60
let DEADZONE_START_VALUE = 5

let CENTER_ICON_SIZE: CGFloat = 35
let ARROW_ICON_SIZE: CGFloat = 35
let MENU_EXPAND_OFFSET: CGFloat = 210
let HIGHLIGHT CIRCLE INIT SIZE: CGFloat = 3
```

Code 4-11: Collection of sizes (Constants.swift)

Functions

The Functions class is a set of functions that are independent of either model classes or specific view controllers. These can be called by any class from anywhere in the application. Examples include functions for displaying alert messages, for formatting date objects or retrieving the background of a Project object (Code 4-12).

```
func getBackgroundImg(fromProject: Project) -> UIImage? {
    var bkgImage: UIImage?
    if fromProject.background == BACKGROUND_BLANK_STRING {
        bkgImage = getWhiteBackground(width: 2000, height: 2000)
    } else {
        let mapFile = FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask).first!
            .appendingPathComponent("/maps/\(fromProject.name)").appendingPathExtensio
n("map.png")
        let data = try? Data.init(contentsOf: mapFile)
        if let image = UIImage.init(data: data!) {
            bkgImage = image
        }
    }
    return bkgImage!
}
```

Code 4-12: Function for retrieving background image from Project object (Functions.swift)

Style

Includes a set variables of *UIColor* objects. These variables all belong to a specific type of visual object, which all have the same visual style (Code 4-13). Having these declared globally like this, make it very easy to edit them, giving the application a different visual style.

```
struct Style {
    static let iconPrimary = UIColor.fromHex(hex: 0xEF2D56) // Pink
    static let backgroundPrimary = UIColor.fromHex(hex: 0x121212) // Jet Black
    static let backgroundSecondary = UIColor.fromHex(hex: 0x45474A) // Medium grey
    static let backgroundTextField = UIColor.fromHex(hex: 0x303030) // Dark grey
    static let textPrimary = UIColor.fromHex(hex: 0xD0D0D0) // Semi-White
    static let textSecondary = UIColor.fromHex(hex: 0x000000) // Black
}
```

Code 4-13: Style struct with global colors (Style.swfit)

Extensions

Some Cocoa Touch classes have been extended, to give them more functionality. These are all gathered in the Extensions class. Examples include a function to create UIColor objects using the very common hexadecimal format, a snapshot function to take a screenshot of only a selection of a view (Code 4-14) and functions to initialize the common view controllers with only a simple function call.

```
extension UIView {
    func snapshot(of rect: CGRect? = nil) -> UIImage? {
        UIGraphicsBeginImageContextWithOptions(bounds.size, isOpaque, 3.0)
        drawHierarchy(in: bounds, afterScreenUpdates: true)
        let wholeImage = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()

        guard let image = wholeImage, let rect = rect else { return wholeImage }

        let scale = image.scale
        let scaledRect = CGRect(x: rect.origin.x * scale, y: rect.origin.y * scale,
width: rect.size.width * scale, height: rect.size.height * scale)
        guard let cgImage = image.cgImage?.cropping(to: scaledRect) else { return nil }
        return UIImage(cgImage: cgImage, scale: scale, orientation: .up)
    }
}
```

Code 4-14: Extension for taking snapshot of only a selection of a view (Extensions.swift)

4.2.5 Data Serialization

This project serializes its data in two different ways, JSON for storing project files and CSV for exporting data. Serialization of both JSON data and CSV data happens through Protocols and Extensions.

CSV

The CSV serializer protocol defines two computed properties[1], *_csvHeader* and *_csvBody*.

The *_csvHeader* is a string computed property, that when computed for any of the supported classes – with pre-defined constants for the *_csvHeader* string – use reflection. Using reflection is a way to get type information at runtime, to determine any other required header sections. When serializing an

[1] See Syntax in Swift

Entry object, which contains a *Legend*, the serializer will detect that *Legend* variable through reflection and append the necessary CSV header tags (Code 4-15).

```
for case let (_?, value) in Mirror(reflecting: self).children {
    switch value {
    case is Legend:
            header += "\(CSV_ENTRY_NAME);\(CSV_ENTRY_ICON);"
    case is Entry:
            header += "\(CSV_TIME);\(CSV_COORDINATES);\(CSV_ANGLE_IN_DEGREES);\(CSV_ENTRY_NOTE);"
    default:
        header += ""
    }
}
```

Code 4-15: Generating CSV header through reflection (CSVSerializable.swift)

The *_csvBody* is also a string computed property that will extract relevant data based on the class being serialized and append it to a string. As with the *_csvHeader* the *_csvBody* will use reflection to detect other model[1] classes and append any relevant data (Code 4-16).

```
for case let (_, value) in Mirror(reflecting: self).children {
    switch value {
    case let value as Legend:
        body += value.name + ";" + String(value.icon) + ";"
    case let value as Entry:
        body += String(describing: value.time) + ";"
        body += value.note + ";"
    default:
        body += ""
    }
}
```

Code 4-16: Generating CSV body through reflection (CSVSerializable.swift)

Note, the delimiter used is a semicolon (;).

JSON

The JSON serializer protocol defines a single computed property, *JSONRepresentation,* which is of type *Any*, meaning it can store anything, however, internally the JSON representation is stored as a *dictionary* (equivalent to a *map* in Java) of *String* to *Any,* where the *String* is the actual variable name and the *Any* represents the value of that variable. The protocol is defined for all relevant model classes as well as any relevant types, and will use reflection to determine the type, and handle the serialization accordingly.

The JSON serializer protocol was developed based on a pre-existing example.[2]

[1] See Model

[2](Konchev, 2017)

For the model classes, it will call *JSONRepresentation* on itself, causing it to then handle the serialization of the variables in that model. Calling *JSONRepresentation* on an *Entry* object, will serialize the variables *_time, _position, _angleInDegrees, _note* and *_tagId* as specified in the extension, but the *Entry* model class also contains a *_legend* variable of type *Legend*, here the serializer will call *_legend.JSONRepresentation,* and the serializer will then go through each variable in the *_legend* instance and serialize them (Code 4-17). This way the procedure for serializing a model class is not dependent on whether it contains an instance of another model class.

```
29              case let value as Entry:
30                  jsonDict[label] = value.JSONRepresentation
```

Code 4-17: Model classes will generate their own JSON representations (JSONSerializable.swift)

For other types, the serializer will process these based on what the actual type is. With an *Int,* the serializer will simply append the variable name string and the value of the variable to the JSON dictionary. However, some types can't simply be handed to the dictionary as a key-value pair, but requires some processing, this is the case with the *Date* type. The *Date* object is converted to a string using a *DateFormatter*, and then handed to the dictionary (Code 4-18).

```
32              case let value as Date:
33                  let df = DateFormatter()
34                  df.dateFormat = "yyyy-MM-dd'T'HH:mm:ssZ"
35                  df.timeZone = TimeZone(abbreviation: "CET")
36                  jsonDict[label] = df.string(from: value)
```

Code 4-18: Preprocessing Date object for JSON serialization (JSONSerializable.swift)

Deserialization is implemented per model class, so each model class handles deserialization of its JSON representation separately. Since all types except *Date* and the model classes are serialized to a format Swift understands, the JSON entry for each variable can simply be cast to the type it represents. The dates need to be converted from a string to an actual *Date* object using a *DateFormatter* and since each model class handles its own deserialization, each JSON representation of a model in the project will call its JSON constructor with its representation as input (Code 4-19).

```
177             self._entries = [Entry]()
178             for ent in entries as! [[String: Any]] {
179                 self._entries.append(Entry(json: ent)!)
180             }
```

Code 4-19: Deserialize all entries in a project (Project.swift)

5 Discussion

This chapter details the choices that were made, feedback received on the project and eventual alternative options that were considered.

5.1 Understanding the Assignment

The group had no prior knowledge of behaviour mapping before starting the assignment. To be prepared for creating an application for that purpose, it was necessary to properly understand behaviour mapping. This was solved in several ways. The group read existing reports where behaviour mapping had been used extensively. Primarily "Behavior Mapping: A Method for Linking Preschool Physical Activity and Outdoor Design"[1] and "Kjøpesentres påvirkning på sentrum - En casestudie av Svortland sentrum og Hellvik Senteret"[2].

The group also had several meetings with Daniella Müller-Eie and Monica Reinertsen. Daniella is an associate professor at the University of Stavanger in City Planning. Monica Reinertsen is master level student, studying City Planning and Urban Design. They contributed in explaining behaviour mapping. During the meetings, they were also shown the latest version of the application, and provided feedback on it and the group's ideas for further development. This ensured that the direction of development was reviewed by someone with authority on the subject of behaviour mapping.

5.2 Platform

The assignment didn't have any restrictions regarding choice of platform. Developing the application for several platforms was also possible. To ensure that application would have the features and level of quality deemed satisfactory, it was decided to spend the time to develop the application for a single platform.

The next choice was to select what operating system the application would be developed for. At the time, there were 3 operating systems commonly found on tablet devices: iOS, Android and Windows. Android was both the platform the group had the most previous experience with, and the platform with the largest market share.[3] Despite this, iPad and iOS were chosen. While this would result in some time spent getting acquainted with the platform, we felt the advantages were worth it. There is very little uniformity in the tablets running Android. Two units running Android can have very different screen resolutions and screen aspect ratios. To remedy the different models, one needs to make several compromises when designing the user interface. This makes it hard to create an application that will behave consistently on different devices. The iPad on the other hand, is a much more predictable

[1] (Cosco, et al., 2010)
[2] (Madsen, 2015)
[3] (IDC, 2016)

platform. Every iPad has the same screen aspect ratio and a minimum resolution of 1024 by 768 pixels. This means that user interface elements will look similar and have the same proportions across all iPad versions.

It was decided to develop it for the latest version of iOS – iOS version 10. At the time, 79% of the active iOS devices was running this version.[1] This version of iOS is compatible with all versions of iPad except the 3 first generations of the regular iPad and the first generation of the iPad mini. Choosing this version of iOS means that the application can only be run on 9 types of iPad, making it easier to create an application which performs equally well on all the devices it can be used with.

5.3 Mapping Design

A big challenge in designing this application, was deciding how the behavioural mapping would be translated to a digital user interface system. There doesn't exist a template for how the mapping in behavioral mapping is done. Many different methods are used, depending on the nature of the research, and the preference of the researcher. In many cases the observer might draw observed participants as small icons (Figure 5-1), in other cases they might be drawn with full paths tracking their movement (Figure 5-2), and in some cases observed participants are simply counted in different tables (Figure 5-3).

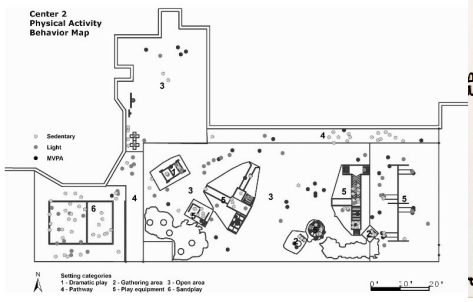

Figure 5-1: Mapping with icons[2]

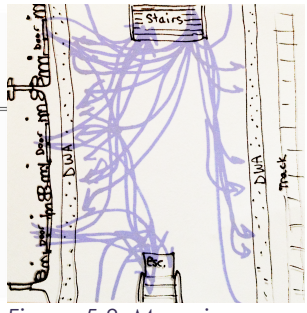

Figure 5-2: Mapping with paths[3]

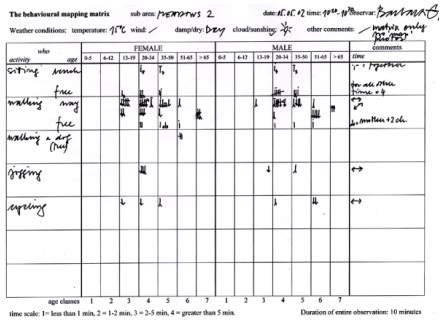

Figure 5-3: Mapping with tables[4]

To account for this divergence in methods, the system needed to be highly flexible. The initial design required the user to create location objects. Mapping participants would involve tracking how many were at each pre-determined location, or how they moved from one location to another (Figure 5-4). While this solution would be adequate in many situations, it was ultimately considered too restrictive, and therefore discarded.

---

[1] (Apple Inc., 2017)
[2] (Cosco, et al., 2010)
[3] (Wilden, 2012)
[4] (Marušić & Goličnik, 2012)

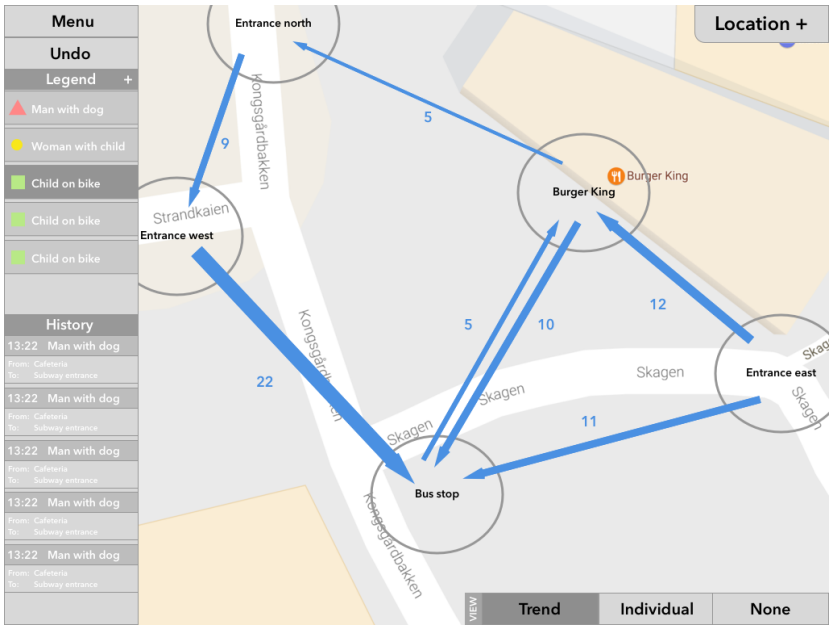

Figure 5-4: Early design of Behaviour Mapper

In the final version, none of these restrictions exist. The entries can be placed anywhere on the map with customizable icons. This system should be compatible with most mapping situations. The observer can even track the path of a participant's movement. This can easily be done by creating several icons with arrows, mirroring the participant's movement (Figure 5-5).

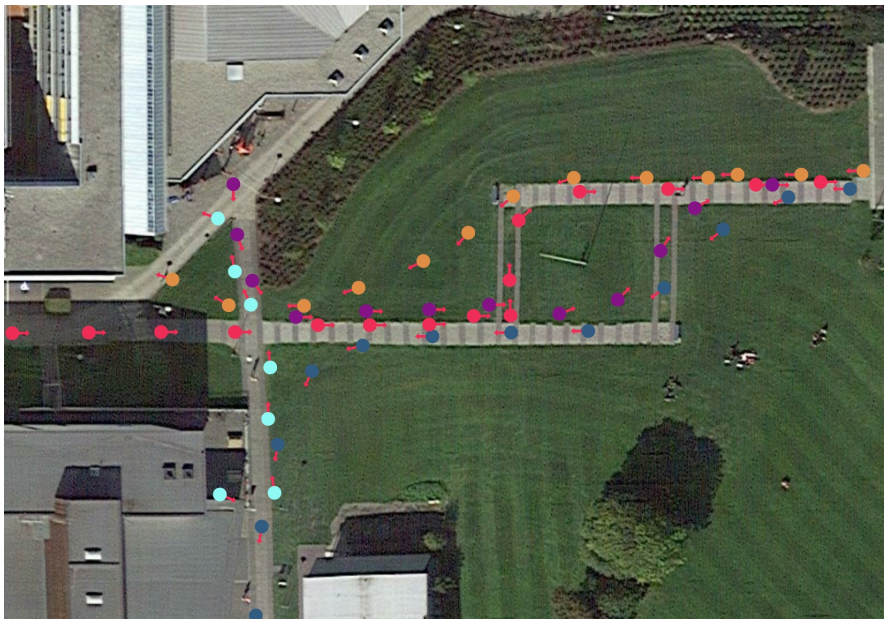

Figure 5-5: Pathing on Behaviour Mapper

Another priority was to store and retrieve as much, or more information than what was gathered from using pen and paper. As the data model for each entry stores their relative location, direction, name, icon and time of entry, it greatly surpasses the information that can be collected from an entry drawn with pen and paper. These data can also be easily exported as a CSV file. An

immediate benefit from this is that almost no time needs to be spent manually getting the hand drawn entries registered digitally. This also removes the chance for misreading the entries, resulting in data that always matches what was initially registered (Figure 5-6).

Behavior mapping Torget										
Torsdag 23.03.17	Været	Gå	Sykkel	Barnevogn	Lufte hund	Jogge/løpe	Skate/sparkesykkel	Stå	Sitte	**Totalt**
Kl. 08:15-08:20	1 grad, Sol, Lite vind	31	5	0	0	0	0	0	0	**36**
Kl. 12:11-12:16	3 grader, Sol, Lite vind	132	1	11	0	0	0	5	5	**154**
Kl. 16:02-16.07	6 grader, Sol, Litt vind	125	13	1	0	1	0	1	4	**145**
Kl. 19:25-19:30	2 grader, Tørt, Lite vind	95	1	0	1	0	0	6	0	**103**
Kl. 23:40-23:45	2 grader, Tørt, Lite vind	68	1	0	0	0	0	9	0	**78**
25 minutter	Totalt	451	21	12	1	1	0	21	9	**516**

Figure 5-6: Spreadsheet of data from Behaviour Mapper

5.4 User Interface

The goal of the application was to be a replacement to pen and paper during behavioural mapping. To that end, it needed to be intuitive and fast to use. When doing behavioural mapping, there might be many participants observed at the same time. To register them all correctly, the registration needs to be very quick. Because of this, it was decided to front-load a lot of the registration. The application lets the user define all the symbols with name and icons before the mapping process begins, removing what could become a very time consuming task from the actual mapping. While symbols need to be created beforehand, it is still possible to add additional ones while in the mapping screen. With the symbols already defined, registering a participant is as simple as tapping the screen where it's located. Switching to another symbol is done by tapping that symbol in the vertical row. To add a direction, the observer only needs to press and drag in the desired direction before releasing the finger from the screen. Using this solution, a participant can be registered in less than a second. The assignment's suggested functions included the ability to edit the time stamp of an entry, to allow registration of multiple entries at the same time. As the final system lets the user register entries with less than a second in between them, this ability to edit time stamps was deemed unnecessary, and not implemented.

A drawback inherent to using pen and paper is how cluttered it can get. Behaviour Mapper mitigates this in several ways: It is possible to filter visible entries on the mapping view. Each new entry also gets logged in the history table view. This gives the observer a full overview of the entries. Each entry is presented in a list as a table cell with name, icon and time of registration. Tapping on an entry will animate circles around the corresponding object on the map, letting the observer know which object it belongs to. Another disadvantage of pen and paper, is how hard and time consuming it can be to correct erroneously drawn entries. In Behaviour Mapper, such entries can be easily deleted by sliding the selected entry to the left, and tapping "Delete", instantly removing it.

To make the application intuitive to use, there was made a conscious effort to use interface functionality already familiar to users. Both the list of entries and the list of symbols can be dragged up and down, like other similar elements found in many applications on the same platform. Sliding table cells to remove additional options is also a common trait in Cocoa Touch applications. This guarantees that many users recognize and understands this functionality, which reduces the amount of learning needed to use the application.

5.5 Visual Style

The overall visual style of the application was intended to have a minimalist, professional tone, with only a few different muted, colours used throughout. The background on the starting screen was designed to resemble a cityscape, communicating the intent of the application while making it more visually engaging. The icons used for mapping were largely inspired by the behaviour mapping symbols used in the master thesis "Kjøpesentres påvirkning på sentrum - En casestudie av Svortland sentrum og Hellvik Senteret"[1]. At the same time, they were constructed to be visually distinct and easily readable. All graphical assets were created with Sketch.[2]

5.6 Quality Assurance

To ensure that the application will function as intended, it was important to run tests. These would reveal the applications level of performance and how stable it is.

Performance

The application was tested on simulators in Xcode on all available iPad versions that could run iOS 10. In addition to the simulators, it was tested on an iPad Air and an iPad 4th generation, as the group had access to these devices. Performance was considered good on all devices, so no changes were made to the application in regards to performance.

Testing

There was continually performed manual tests of the application. The tests attempted to trigger all the events of the user interface, to uncover eventual bugs.

[1] (Madsen, 2015)
[2] https://www.sketchapp.com

5.7 Field Test

Fellow University of Stavanger student Monica Reinertsen, was studying City Planning and Urban Design when Behaviour Mapper was under development. She was writing her master thesis[1] about the market square in Stavanger. Because of this, she was asked to try out the application for her behaviour mapping research. An unfinished version of Behaviour Mapper was given to her. She used the application on her personal iPad when performing behavioural mapping in the market square in Stavanger. This was done five different times, each day for a week, late in march 2017.

After her week of testing, she provided valuable feedback based on her use. Monica suggested many features that wasn't initially considered, as the group had no practical experience in behavioural mapping. The option to create a new project based on a previous one, and the ability to delete all entries at once, were both suggestions from her. Her feedback proved very valuable, as these features made the application more flexible.

Despite missing these features at the time of testing, Monica was very satisfied with Behaviour Mapper.[2] She said the speed and ease of use made it possible to gather more data than what would have been possible with pen and paper. Monica gave the application her recommendation.

5.8 Limitations

Quality Assurance

Due to a lack of experience in quality assurance, only manual tests were performed on the application. Written, automated tests would have been much faster. They would also have been more consistent, and could potentially have uncovered problems that manual testing failed to discover.

Field Testing

Field tests of the application is highly valuable for understanding how well it performs for its intended use. While the field testing done by Monica Reinertsen was extensive, it would have been very beneficial for Behaviour Mapper to be tested by several people, ideally in very different situations. Not only would this reveal how well it functions in varied settings, but the user could potentially provide feedback on how to improve how it functions in the given setting.

---

[1] Byens festplass i hverdagen - en casestudie av Torget i Stavanger (Reinertsen, 2017)

[2] Impressions of Behaviour Mapper after testing (Reinertsen, 2017)

6 Conclusion

Behaviour Mapper is an application that can perform most of the same tasks as pen and paper during behavioural mapping. Many of these tasks the iPad application achieves with much greater efficiency. The goal of implementing the suggested features[1], is considered met. Despite this, there are still potential features missing from Behaviour Mapper. Some of them were only ideas, while some entered development. Due to time, they never made it into the final application.

Having the user interface in English was a deliberate choice, made to increase the potential user base. For a long time, the plan was to also make a Norwegian version of the user interface available as an option.

The user interface was designed to be very intuitive and would ideally require little to no instruction to use. A tutorial feature was still considered beneficial, but never left the planning stages. It would describe the different interactive elements on a given screen, either automatically for first time users, or when requested by user input.

Experiments were made to implement accurate GPS data in the entries, which would allow the user to export the recorded data to GIS[2] tools. This was considered unimportant for most users, and further development was dropped in favour of other features.

While the feedback based on using the application has been positive, it can be argued that the lack of breadth in field testing lessens the value of this feedback.

Overall, Behaviour Mapper is a tablet application that can serve as a valuable replacement to manual registration during behavioural mapping.

[1] See list of suggested features in Description of Assignment
[2] Geographic Information System

7 References

Apple Inc., 2015. *Cocoa (Touch).* [Online]
Available at:
https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/Cocoa.html
Apple Inc., 2017. *App Store.* [Online]
Available at: https://developer.apple.com/support/app-store/
Apple Inc., 2017. *Foundation | Apple Developer Documentation.* [Online]
Available at: https://developer.apple.com/reference/foundation
Apple Inc., 2017. *The App Life Cycle.* [Online]
Available at:
https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html#//apple_ref/doc/uid/TP40007072-CH2-SW2
Clyne, K., 2017. *Behavioural Mapping.* [Online]
Available at: http://designresearchtechniques.com/casestudies/behavioural-mapping/
CocoaPods, 2017. *CocoaPods.* [Online]
Available at: https://cocoapods.org/
CocoaPods, 2017. *CocoaPods Guides - Getting Started.* [Online]
Available at: https://guides.cocoapods.org/using/getting-started.html
Cosco, N. G., Moore, R. C. & Islam, M. Z., 2010. *Behavior Mapping: A Method for Linking Preschool Physical Activity and Outdoor Design,* s.l.: Americal College of Sports Medicine.
Google Developers, 2017. *Getting Started | Google Maps SDK for IOS | Google Developers.* [Online]
Available at: https://developers.google.com/maps/documentation/ios-sdk/start
Google Developers, 2017. *Google Maps APIs | Google Developers.* [Online]
Available at: https://developers.google.com/maps/
IDC, 2016. *Worldwide Tablet Shipments Decline More Than 12% in Second Quarter as the Market Shifts Its Focus Toward Productivity, According to IDC.* [Online]
Available at:
http://www.businesswire.com/news/home/20160801005182/en/Worldwide-Tablet-Shipments-Decline-12-Quarter-Market
Konchev, I., 2017. *Swift 3: JSON-serializable structs using protocols.* [Online]
Available at:
https://gist.github.com/stinger/803299c1ee0c95e53dc3d9e59c37b187
[Accessed 14 05 2017].
Madsen, Å., 2015. *Kjøpesentres påvirkning på sentrum - En casestudie av Svortland sentrum og Hellvik Senteret,* Stavanger: Universitet i Stavanger.

Marušić, B. G. & Goličnik, D., 2012. *Behavioural Maps and GIS in Place Evaluation and Design.* s.l.:IntechOpen.
Reinertsen, M., 2017. *Byens festplass i hverdagen - en casestudie av Torget i Stavanger,* Stavanger: University of Stavanger.
Unknown, 2017. *GitHub - CocoaPods.* [Online]
Available at: https://github.com/CocoaPods/CocoaPods
Wikipedia, 2017. *Software Design Pattern.* [Online]
Available at: https://en.wikipedia.org/wiki/Software_design_pattern
Wilden, A., 2012. *Research Studio - Oct 29.* [Online]
Available at: https://ashwilden.wordpress.com/tag/behavioural-map/

8 Figures

9 Code

10 Applications and Services Used for Development

Name	Description and Use
Xcode[1]	Integrated development Environment – All programming and simulation of the application was done in Xcode.
GitHub[2]	Online Git service – Used as a repository for the Xcode project. (www.github.com)
Sketch[3]	Vector graphics editor – Visual mockups were created with Sketch, in addition to all the graphical assets used in the application.
Trello[4]	Project management – Used for tracking project status and tasks.
Excel[5]	Spreadsheet software – Gantt chart for time management was made using Excel.
Dropbox[6]	File sharing – Dropbox was used for sharing various files related to the project.

[1] https://developer.apple.com/xcode/
[2] https://github.com
[3] https://www.sketchapp.com
[4] https://trello.com
[5] https://products.office.com/excel
[6] https://www.dropbox.com/home

11 Appendix

11.1 Impressions of Behaviour Mapper after testing

Monica Reinertsen used the application for a week when doing behaviour mapping for her master thesis in march 2017. This is what she said about it:

> I used Behaviour Mapper when doing research for my master thesis in City Planning and Urban Design at the University of Stavanger, spring 2017. The application simplified much of the work saved me much time. It made it possible for me to gather more data than what would have been possible without the application. This because the alternative would be to do it manually with pen and paper, and then digitalize the data later. The application makes it easy to register in the field, and then to export the data as maps and numerical data.
> I used the application 5 times every day in a week from 20.03.17 – 26.03.17. In March, the application was still in development, but it was still very useful to me. Behaviour Mapper is an application I would recommend to other students and companies when doing behavioural mapping.

11.2 External appendices

A: Project Files.7z
B: Instructions for running project.docx