Accessible Smartphone Applications for the Visually Impaired

Jasmina Pelivani

Kongens Lyngby 2012
IMM-BSc-2012-31

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

The main purpose of this project is to understand the needs of visually impaired users when using smartphone technology. Many of the applications that are already developed do not have focus on accessibility. This thesis reviews the possibilties which exist in order to add accessibility to smartphone applications - what can be done in order to make them as useful as possible for the visually impaired?

In the thesis I will consider the design of user interface for this kind of Android applications, by working with problems of using a smartphone, problems provided by a visually impaired, who also suggested this project.
In order to evaluate if the user interface principles which I have come to are useful, an example of an Android application is developed. The application, TravelBuddy, helps the user to find information about the public transport in Denmark, using Rejseplanen's API. Android's built-in function TalkBack is used for better interaction between the user and smarthphone - TalkBack reads what is going on on the screen.

Based on the information I got from the meetings with the client, I have established the use cases for the program, designed the program structure and finally implemented the application. The application, and ideas for this, have continuously been reviewed by the client in order to make the application accessible for the target group.

The study concludes that if developing with the focus on accessibility for visually impaired, useful applications can be developed which actually make it possible for a visually impaired to use them. The focus must stay on the accessibility when choosing between two solutions, otherwise it often result in unuseful applications for the visually impaired.

Preface

This project deals with a very serious problem - the problem of not being able to see. The thesis is a research of using smartphone technology in making it easier to travel and plan a trip using public transportation. The main focus is to find out how to develop an application for the visually impaired, in terms of learning to understand the user's needs.

This thesis was prepared at the Institute for Informatics and Mathematical Modeling at the Technical University of Denmark. It consists of a thorough report on what I have accomplished and learned during this project.

Kongens Lyngby, August 2012

Jasmina Pelivani

Acknowledgements

First of all, I would like to thank my supervisor Associate Professor Christian W. Probst, who offered me this project. I also wish to express my gratitude to Daniel Gartmann and IBOS (Institutet for blinde og svagsynede).
Both of you were abundantly helpful and offered invaluable assistance, support and guidance.

Deepest gratitude to DTU for providing me an Android smartphone, enabeling me to develop and making this project possible.

Last but not least, I wish to express my love and gratitude to my beloved family and friends; for their understanding & endless support, through the duration of my studies.

Contents

Chapter 1

Introduction

Smart phones are becoming a part of our everyday life. A HTC-commercial says: "It's the first thing you see in the morning, it's the last thing you see at night." We use our smartphones for everything, googling a recipe, chatting, finding directions, it follows us to work and school and much more. One of the many things we use smartphones for are traveling, using travel applications, helping us to get around.

Along with smartphones and applications, traveling has become easier in many ways. You can plan a trip long before you need it, from anywhere in the world. All you need are three things - a smartphone, battery and internet connection.

So how do we travel?
First of all, we need to find the possibilities of how to get to our destination. To plan a trip, we either use a homepage or an application. In Denmark, for example, Rejseplanens application is used to plan a trip using public transport. There are many applications for this purpose and the way they compete is based on functionality and design. The smarter and fancier the design is, the more popular the application gets. With the first applications of this kind, the user had to provide his location, destination, time of the travel, etc. - basically everything. This has changed. The applications use the GPS to find the users location and usually there are instant results based on this location, giving the nearest bus stops and destination possibilities as a result.

The one thing that all these applications have in common is that they expect that the user can use these smart designs, which are based on graphics and need visual exploration. A thing which not every person is lucky enough to be able to provide - a blind or a visually impaired person.

Besides the challenges of planning a trip, we also need to physically find the stops and transport vehicles, also a thing which is quite a challenge for visually impaired.
Traveling; some of us consider it a routine and it is a thing we need to do every day, but still, some of us have to struggle to do this every day thing.

The visually impaired also use smartphones and applications. There is a possibility of a large market consisting of applications which are not yet made. There is a lot of motivation here, because there is the unique possibility to help make a group of people's life easier in many ways.

The challenge of this project is to understand and get acquainted with the problems which occur when programming for the visually impaired - what restrictions are there, how to make the program user friendly and especially how to make it useful?

1.1 Target Group and Motivation

This project was suggested by Daniel Gartmann from the Institute for Blinde and Visually Impaired (IBOS), a blind himself. I have been in contact with him during the project, getting his assistance and opinions at various choices I will have to make. Further on in this thesis, he will be referred to as "the client".
The visually impaired are the target group of this project, but I will continually compare my design choices with the choices of developing for a target with no visual disability.

I was drawn to this project both because of a personal wish of learning about a new technology, smartphones, the interesting aspects of working on an uncovered field - programming applications for the visually impaired, but the motivation also comes from that this is a needed application and is requested by an individual and also IBOS.

1.2 Problem Definition

Design and implement an Android application for the visually impaired. The main focus is to understand the user's needs and design the application in an accessible manner. The function the application should offer the user is to provide information about public transport in Denmark and the ability of planning a trip, including the possibility of getting the trip description.

1.3 Solving the problem

During this project I will study and analyze how to make a general user interface for an application with the visually impaired as the target group and finally make an example of such an application.
The application I will focus on will be a travel application, to help the user get from point A to point B, using directions, where the route also uses public transport. This can be done by using data from for example Google Maps or Rejseplanen, the Danish travel plan.

As I wrote before, there are many examples of applications for this purpose. But when used by a visually impaired, smart graphics are confusing instead of smart. Without being able to see, it is difficult to imagine the way of using the many options and functions or their purpose, when sometimes their purpose lies in their position, their size or other design choices.
There are very specific user needs for an application for visually impaired - to make the application as accessible as possible and thereby take input and represent output in a useful way.

The technical aspect of the project is to program an Android application, with Java as the programming language. This contains functionality as well as (graphical) user interface.

One of the outcomes of this project will be to learn to listen to the user's needs and translate it to a product that is useful and satisfying.

The application developed during this project is developed for Android smartphones and is developed on a Samsung Galaxy Nexus, with Android IceCream 4.0 as the operative system.

This is a 15 ECTS point project and also my first larger programming project which I will work on alone. I hope it will show what I have learned during my

2 and a half year of studies at the Technical University of Denmark.
I hope to achieve important knowledge about programming for the visually impaired and end up with a suggestion for a helpful and useful application.

CHAPTER 2

Theory - Programming with Android

This chapter contains overall theory about the important components used in this project - Android components as well as general information about Rejseplanen.
I will not cover theory about Java programming in general, but special design choices will be discussed continually instead.
This theory chapter is based on web research and mostly uses the official Android developers home page[2], Lars Vogels blog on Android programming[4] and other blogs and internet research.

2.1 Android Components

This section is an overview of the basic Android Components that are used in this project.

2.1.1 Activities and threads

Every screen in an Android application is based on an **Activity**, meaning that an application can have many activities, each representing the different screens of the application.

Whenever an application is started, so is a thread. This is the main thread, also called the UI (user interface) thread. All the activities of the application run on this thread and the UI is modified by this one thread.

If you need your application to do something which takes a long time, you will have to run it on a different thread, otherwise, the main thread will be blocked untill the time consuming code is done running. This would result in a very slow application and bad user experience.
Therefore, slow or time consuming operations need to run asynchronously with the main thread.

2.1.2 AsyncTask

In order to run methods asynchronously, the **AsyncTask**[6] class can be used. The AsyncTask class has three important methods, which describe its purpose.

- `doInBackGround()` The code you want to run asynchronously to the main thread is executed by this method. It is done in background through another thread than the main thread and thereby does not block the main thread.
- `onPreExecute()` This method takes care of code which needs to be run before running the code in `doInBackground()`.
- `onPostExecute()` If anything needs to be done after `doInBackground()`, it is coded here. Also, if you need to use the outcome of `doInBackground()`, which you probably do, here is where you handle it. The AsyncTask thread synchornizes itself with the main thread again while executing `onPostExecute()`.

2.1.3 Views and Layouts

Interfaces like buttons, imagefields, textfields etc. are called **Views**. They occupy areas on the screen and are responsible for event handling, such as touch

and other user events.
The views can be seen on the screen and are mostly defined in XML-files, which are the layout files of the application. Views can also be created with Java code, but the creation is most commonly done together with the layouts of the activity it belongs to.

2.1.4 Intent

Intents are used when starting a new activity, as a kind of link from one activity to another. For instance, if you desire to exchange data between two activities, you would use intents.

2.1.5 Android Manifest

The Android Manifest is an XML file, which holds the essential information about the application.
This file holds the name of all the activities, application information and various permissions the application needs in order to function, like for instance internet or GPS access.

2.2 Internet and GPS Access

If you would like to have internet access or to get GPS information in your application, for downloading information, checking in with your location or other things, you need permission to do it.
To get internet permission in your application, you have to add the permission in the Android Manifest file, which is an XML file.
This is done by adding the following line:

```
<uses-permission android:name="android.permission.INTERNET" />
```

2.3 Rejseplanen and XML

In order to use Rejseplanens API, I only need internet acces and Rejseplanens API documentation. The documentation is not public, so I gained it by contacting Rejseplanen and therefore it is not to be found in this rapport.

After adding internet access, with a HTTP client, one sends an HTTP request with a specified URL to Rejseplanens API. When this is answered, we gain access to a XML file, holding the needed information.

An XML file consists of tags, elements and attributes. A tag can have both element and attributes, leaving various possibilities in constructing an XML file.

Different HTTP requests return different HTTP response and in this case different XML files. This means that an XML parser for all the different outcomes is needed.
I will cover used Rejseplanen API calls continuosly, when I encounter them while implementing the services.

2.4 Accessibility and Input

Accessibility is an embedded function in the newer Android smartphones.[7] Its purpose is to allow users to use their smartphone eyes-free, which is essential for visually impaired users.

Enabling the Talk-Back and Explore-By-Touch functions, the user can explore the smartphone by touch. The smartphone reads out loud what view is touched. This way the user knows which application he is at, before starting it.
When developing for Accessibility, one has to add in the program what should be read by the smartphone when the different views of the application are activated by user events.

By interviewing a couple of visually impaired, I have found out that they lately have startet to use touch-screen keyboards - untill now, they felt it was easier to use a physical keyboard.
Now, the touch-screen keyboards are more user friendly. With Explore-By-Touch, anything you touch is read out loud. When touching the keyboard, the letters are read out loud, but are written when released. So you can skate through the keys and find the key you were looking for - then you release the key and its corresponding character will be written.
There are various other downloadable touch-screen keyboards, designed for this purpose exactly, which can be customized by many different settings.

Chapter 3

Requirements and Project Analysis

The working done on this project, is with hope to result in TravelBuddy - an application which helps the visually impaired to move in the traffic.
Before the programming can start, this project needs a lot of researching - first of all I need to find out what problems need to be solved, pick what problems I am going to work with and last but not least, find out what possibilities I have in order to solve them. I have done a lot of web researching, but there is not much material available for accessibility programming, because it is a rather uncovered field.

Therefore I turned to my main source, the client and other potential users of the application and asked what kind of problems they are facing when traveling, or in general when using their smartphones. Some of the problems I found were:

- The smartphone applications aren't strictly designed for the visually impaired, so the accessibility isn't the first priority
- The user doesn't know the difference between an input field or just an ordinary information label
- Physically find the nearest stop

- Finding what direction the transport vehicle at the stop is headed
- The next transport vehicle from a stop
- Planning a trip using public transport
- When to get off of a transport vehicle
- When and which vehicle is at the stop

Along with problems, there are various ways to solve them. I need to analyze the possibilities and find the most suitable for this specific project. First of all, the general choices needed to be considered - how to approach the programming. My considerations on several topics, i.e. platform, choices of problems to solve, are following.

3.1 Platform: Android versus iOS

The first important choice to be made is what platform is the application going to be developed for. My choice stands between Android and iOS.
Given that the Android developer tool kit is open-source and the client has an Android smartphone, the choice was quite obvious for me. Although developing for iOS could be interesting, sources say that one needs to have an Apple computer, in order to program for iOS. Additionally, the software development kit isn't open-source, one needs to register and pay annually in order to be an iOS developer. [5]
Another advantage with Android is the embedded function Accessibility, which is shortly described in chapter 2, Theory. A Accessibility funtction which can be turned on is TalkBack, which "talks back to the user" and reads what is going on on the screen. This will be covered later in this thesis.

Based on these facts, I choose to develop TravelBuddy for Android, and therefore have done very little research on whether or not iOS has a similiar function to Accessibility. The iOS Accessibility has the function VoiceOver[**?**] which is the equvalent to Android's TalkBack. So the oppertunity to develop this application for iOS is also there.

3.2 Approach

This being my first program for Android, I have to find out how to approach our client's interesting problem.

It is very important for me, that the client feels that I have treated his needs as the most important thing, mostly because if I don't do that, this application will be one of the many travel applications - and then there would be no purpose making it.
Therefore, I will try to solve the problems in the order as he prioritized them, but at the same time I will try to get the most out of the project.
Given there are two parts to this project, accessible user interface being one and getting information from the internet in order to travel being the other, I divided the whole project and thereby the application in two parts.

So in order to do the first part, I decided to do a small application with only user interface as main focus. I need to gain insight in Accessibility programming - what possibilities are there and how much help can I get from the embedded functions versus what I need to implement and design myself, in order to make the application useful for the visually impaired.

The second part is finding out how to get the needed information, in order to program the application which can help one to travel using the public transport. So without thinking about this specific application, I want to get information about departures, arrivals, stops and how to get to the stops.

This way divide and conquer is used, by solving small parts of the problems independently and afterwards assemble them to be one functional program.

3.3 Google Maps versus Rejseplanen

In order to get the needed information about the public transport, there are two obvious possibilities - Google Maps API(short for Application Programming Interface) or Rejseplanen's API.

Google Maps is a technolgy provided by Google, and is a web service application, which offers route planning.
Rejseplanen is a Danish webpage, which distributes departure and arrival times for Danish public transport. It is held by Rejseplanen A/S.

There are pros and cons for both technologies. Rejseplanen is Danish, while Google Maps is used internationally and thereby is used more than Rejseplanen, meaning there is more documentation and help to be found on the internet for Google's API.

Rejseplanen's API offers information about the various transport vehicles, like their direction, name, route and information about stop locations, but not walking directions to the stop. When using Rejseplanen's homepage on the other hand, there are directions, but these are provided by NAVTEQ, another company and therefore these information can't be accessed by the API. This is of course a problem, because I would like to add directions from and to bus stops. This would be avalible by using Google Maps API - but I evaluate using Google Maps to be a risk and this is why. While Rejseplanen only uses Danish locations and public transport, I reckon that there would be less misleadings, or in a smaller scale, then using the whole world as a potential location. For a regular user, it wouldn't be that confusing and could easily be changed if the found location is in another country, but it would create big problems for users who only have their hearing to navigate by.
Having the user in mind, I choose to use Rejseplanen's API as the primary source to the needed information.

3.4 Development Choices and Approach

As earlier described, I have some clear goals with the application - to provide help to solve the problems described in the first part of this chapter. These goals are my client's wishes and therefore I have prioritized in the same order as he has. After some meetings with him, I found out what functions are most important for him.
They are as follows: firstly the function to find the nearest stop and the departures from there. The second function he requested was to be able to plan a trip to a specific destination. I have to find out how to tackle the assignment - whether I want to do things in order to be more challenging for me or to answer his whishes.
Being inexperienced in using API's and getting information from the internet to use it in programs, I decided to incorporate the easiest parts first and then build more and more to the program, I wouldn't like to risk it and get nowhere with the application.

Therefore, first of all I want to implement the Next Departure function, which returns the nearest stop and its departures and afterwards the Trip from A to B function, which finds a trip from one location to another.

Having no information about the directions to and from a stop when using Rejseplanen's API as the solution for finding the public transport, it is difficult to provide them. They can be provided by using Google Maps, but since the client hasn't prioritized this, it is something I hope to implement, but it isn't the main priority.

CHAPTER 4

Design and Program Structure

In order to end up with a successfull program, one needs to consider the design and program structure before starting the actual programming.
In this chapter I will analyse the use cases, to get an idea of a fitting program structure and end up with a class diagram, which will be used when programming.

4.1 Use Cases

The use cases in this project are simple to find. They are based on the difficulties a user has, when using a smartphone to be his travel guide. IBOS and our client gave me an insight to their problems and I will formulate them in this section. It is important to keep in mind that the use cases are for visually impaired people, so the actor in every of them is a visually impaired user.

4.1.1 Use Case: NextDepartures

Description:
The user wants to know the next departures from a nearby stop.

Main Scenario:

1. The user presses the Next Departures button on the main screen.
2. The user enters his location and specifies time and date if he desires to know the departures at a certain point in time.
3. The user presses Search.
4. (a) The program represents the found departures.
   (b) The program can not find any departures.

Alternative Scenario:

1. The user presses the Next Departures button on the main screen.
2. The user presses Search without specifying anything.
3. (a) The program represents the found departures using the current time and date and the GPS location.
   (b) The program cannot find any departures for the specified input.

What this use case allows is to give the user the opportunity to just search for the departures from a stop, based on his current position and time of the search or provide specified information for the search.

After analyzing this use case, I know that I will need three activities. TravelBuddy needs a starting screen in order to choose the Next Departures option, a search screen and a result screen. In order to get the results, I will need internet access, a XML-parser and possibly the GPS position.

4.1.2 Use Case: TripFromAtoB

Description:
The user is interested in a trip from either a provided location or his position

to another point.

Main Scenario:

1. The user presses Trip from A to B from the main screen.
2. The user enters his location, destination and specifies time and date if he desires to know the departures at a certain point in time.
3. The user presses Search.
4. (a) The program represents the found departures.
   (b) The program cannot find any departures.

Alternative Scenario 1:

1. The user presses Trip from A to B from the main screen.
2. The user presses Search with only specifying the destination.
3. (a) The program represents the found trips using the standard time and date and the GPS location.
   (b) The program cannot find any trips for the specified input.

Alternative Scenario 2:

1. The user presses Trip from A to B from the main screen.
2. The user presses Search without specifying anything.
3. The user is alerted that a destination in needed in order to perform the search.

In addition to what the analysis of the first use case showed, we need two more activities, namely another search and a result screen. Once again an API call will be needed, but this could possibly be done with the same methods.

4.1.3 Use Case: TripDescription

Description:
The user needs a description of one of the recently found trips.

Main Scenario:

1. User chooses a trip from the trip results.

2. A trip description screen appears.

If a trip is found, there must be a description to it. Therefore there can not be an alternate scenario. We only need to add another activity, namely the description screen.

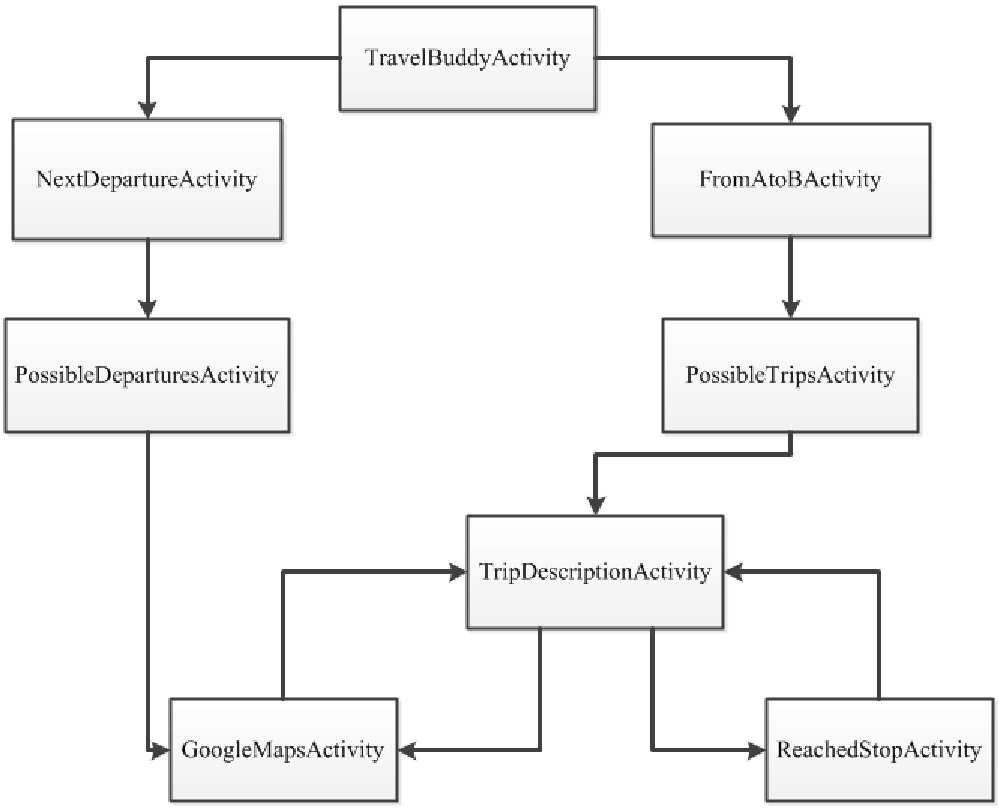
4.2 Flow chart

After looking at the use cases, I have an idea of what activities are needed for making the user interface. Figure 4.1 is a flowchart showing the flow of the program and therefore only the activities are shown. [1]
The arrows show the natural flow of the application, but there is of course the possibility of going back and forth between the activities.

The flowchart is mainly based on the previous use cases. The TravelBuddy-Activity on the top od the flowchart represents the main screen - there are two possibilities from here, representing the two main use cases - NextDeparture and TripFromAToB. Both of these return a result screen, corresponding to the PossibleDepartureActivity and PossibleTripsActivity like the arrows show.

[1]The flow chart has been revised to reflect the final program.

Figure 4.1: TravelBuddy - Flowchart



It can be observed that there are two cycles in the flowchart. One from TripDescriptionActivity to GoogleMapsActivity and one from TripDescriptionActivity to ReachedStopActivity.

A trip can have several steps, which either is to walk to the stop or take a ride with the public transport - meaning the user will need to go back and forth between these.

When in TripDescriptionActivity, the user takes the first step of the trip - to walk to the stop, using GoogleMapsActivity. When he has arrived, he returns to TripDescriptionActivity and then takes the next step with a public transport

- thereby comes to ReachedStopActivity. When the stop is reached he returns to TripDescription. This continues until the final destination is reached and this is illustrated by the cycles. [2]

4.3 Class Diagram

Based on the use cases I have come up with a class diagram for the application.[3]

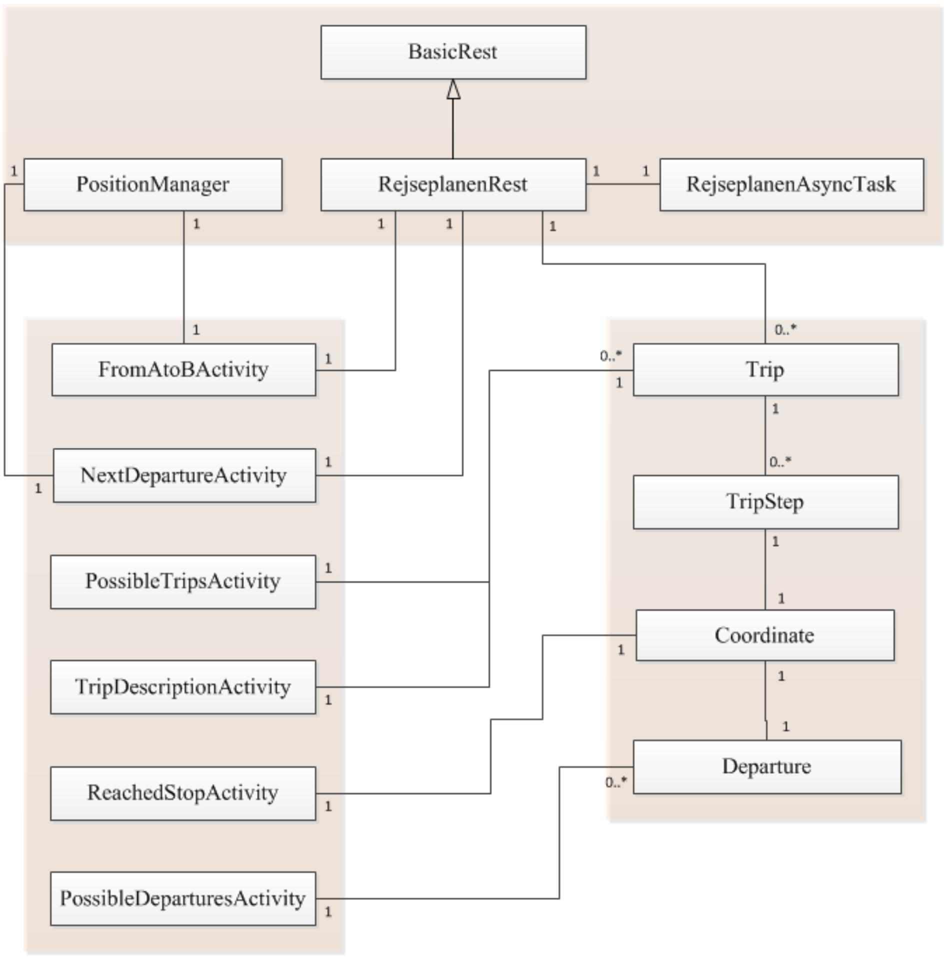
The diagram, figure 4.2, shows the relations between the classes. The colored boxes divide the classes in three parts - activity classes to the left, data classes to the right and the upper box holds the classes which are responsible for making the API calls, internet connection and running the code asynchronously.

RejseplanenRest is the class holding Rejseplanen's API services and it extends the BasicRest class - which takes care of internet connection and performing the API call.

[2] ReachedStopActivity and GoogleMapsActivity are two functions which I decided to implement later on in the project and can be read about in chapter 5, Design and Implementation.

[3] The class diagram has been revised to reflect the final program.

Figure 4.2: TravelBuddy - Class diagram



Chapter 5

Design and Implementation

After considering the program structure and design, the programming can begin.
This chapter covers the design and implementation choices I have made, based on the former chapter on program structure.
As described while analyzing the requirements, the project is divided in two parts, designing the user interface and retrieving information from the internet, in order to make it easier to develop.

First of all, I will describe my experience of programming a general user interface for the visually impaired, how it is to work with Accessibility and TalkBack. Later in this chapter, I will describe the second part of the project, namely getting connection to Rejseplanen and retrieving data. After reviewing the design and implementation of this, I will comment on the UI choices for this specific application, TravelBuddy and show the associated screen shots of the user interface.

5.1 Accessibility and User Interface in general

This section is about the first part of the project, UI in general. The challenge here is to keep the user in mind constantly and find out what are the best design

choices for the application based on the user's needs.
Chapter 1, Android theory, describes the structures of user interface. This being an application for the visually impaired, graphics is not important; the main point is the accessibility. To develop an accessible application, one has to use the provided tools for this and find out how to arrange the layout.

Android allows adding Talk-Back commands by adding
android:contentDescription to a view[1]. This means, the added content description will be read out loud when the view is touched. There is nothing difficult in adding the content description, but having no GUI the user can rely on, these descriptions are his only help and therefore it is really important to add a covering content description. So Android and Accessibility only provide the opportunity to add the description, but it is up to the programmer to use this correctly.
I used TalkBack to gain insight to how it can be used to understand what is on the screen - so I started off with using my smartphone as a blind person would; I explored the smartphone with closed eyes.
There are some things which are different than using the phone normally. To slide from side to side, you have to use two fingers, because one touch is for exploring the screen. To start an application you have to press twice on it - the first time to find it, the second time to launch it.

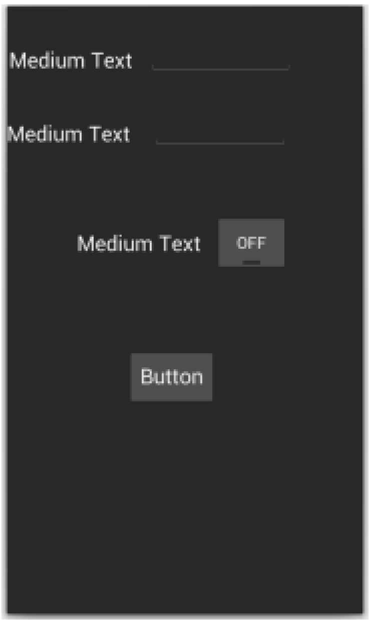
What I have done, is just implement one activity and explore the possibilities that come with accessibility. So basically, this activity shouldn't do anything, besides showing me the user interface. I used different views; text fields, buttons, radio buttons and similar, to get an idea of how it works with accessibility and understand the user's difficulties.
So I explored this small test-application with closed eyes, trying to understand how much information is needed with TalkBack to picture the screen and when the content description makes it too confusing.
Being the one who designed it, I had a picture in my head of it before even starting. So I also asked others to try it, which haven't seen the layout, to see their reaction and see how they would navigate.

I will show an example of an activity, which is an (simplified) example of a regular application. Afterwards I'll demonstrate how I would implement it as an application for visually impaired.

Let us look at a simple application screen.

[1]Views are interfaces like buttons or image fields, described in chapter 2, Theory

Figure 5.1: Example of a regular application

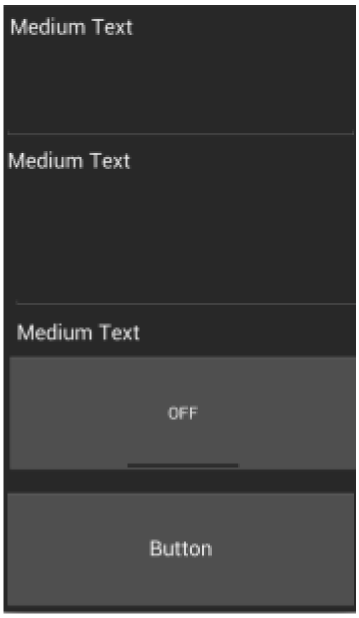


If a visually impaired was to use this application, he wouldn't know where to begin. If the user explores at the left part of the screen he will just meet two labels, which may or may not read what they are when invoked. If he explored the right side, he wouldn't meet any view, because the input fields do not stretch all the way. These are very bad design choices for this kind of application. Hopefully, when the input fields are found, TalkBack would read what needs to be done.
Then, we have a toggle button in the middle of the screen, which is small and placed slightly to the right, so even if the text field beside the toggle button is found there are various possible positions of finding the toggle button itself - the obvious choice is possibly to try to the right of the text field, but this should not be something left for the user to figure out.

And at last we have a button in the middle of nowhere. It is small and located in the lower part of the screen. There isn't anything to navigate around to find it, which makes it difficult to use.

Overall this activity could be used without problems by a regular user - it is obvious what is going on. But for a visually impaired it isn't that simple. So after trying this out with some test persons, I tried to improve the layout based on the things I have learned.

Figure 5.2: Example of an application for visually impaired



What is changed here is mainly the size of input fields and buttons. I would like to make it easier for the user to find the views and therefore they are bigger, but mainly they fill the width of the screen - the user can't be in doubt of what it is on that "row" of the screen. By doing this, the layout automatically becomes a list, so you can just explore from top to bottom.

Also, the text fields are just there for information reasons if a regular user uses the application - they do not do anything with TalkBack, so a blind user wouldn't know that they exist. Everything that is needed is read when a button or input field is invoked. The idea is that the content description replaces the labels.

So here is what I have learned through this experiment.

I found out that comprehensive descriptions are very confusing for the user, especially when you have an application you want to use on the go - it takes

time to read a long content description and you might get confused about what view the description describes if it is too long.
When only having your ears to rely on, it is important that it is easy to access every input field or any other view. You can't have too much content, because that would be very confusing.
Therefore the description must be short and precise, only the needed information should be read.

Besides adding the content description, there is another thing you can do to make it easier for the user to use the application. That is, to place the views in a smart way and avoid making them small and difficult to find. The larger the area of a view is, the easier it is for the user to find it. With this, it is easier to find the views if the different activities of an application are consistent - so you have an idea of what to expect when exploring the activity.

What I have learned through this small experiment will be used when designing the UI of TravelBuddy. I will round this up by describing the design choices, but first, I will use the next sections to review the implementation of the use cases. After the implementation review, which explains how TravelBuddy works, I reccon it will be easier to discuss the user interface.
So the next section will cover the implementation and understanding the function in order to afterwards show screen shots of UI and comment on the UI design choices.

5.2 Internet access and Rejseplanen

The second part of the assignment, was to get internet access from the application. For TravelBuddy, the internet access was needed in order to perform API calls to Rejseplanen and get the desired information. This is done simple:

Listing 5.1: Internet connection

```
URL url = new URL(serviceName);
   HttpURLConnection connection =
      (HttpURLConnection) url.openConnection();
   connection.setRequestMethod("GET");
   connection.setRequestProperty("Accept", "application/xml");
```

When performing this, we use an URL from Rejseplanen's API. The output is an XML file, which is stored as a document object. But before we can do all this, we need to create the correct URL for the API call, which is based on the user input.
To create the URL, handle the API calls and working with the output of the

API calls is the difficult part. The difficulty lies in how to do this, without interrupting the main thread.

Earlier, I have described the AsyncTask class. This is the key in performing an API call. We don't want to disturb the UI thread, and block the application. Instead, we use AsyncTask and perform the API call in background, with doInBackground method. The user of course needs to wait until the call is done in order to get a result, but the UI thread isn't blocked, so he could potentially stop the request.
All the services that Rejseplanen's API offers are located in the RejseplanenRest class. The name comes from Rejseplanen's documentation, as they use ReST (Representational State Transfer) interface for the different API services. The API calls are performed from RejseplanenAsyncTask class, that runs asynchronosly to the UI thread. I will shortly introduce the services that I use when I work with them.
I will analyze the design and implementation of the classes, by looking at the use cases.

When TravelBuddy is started, we are in TravelBuddyActivity class.
Before the activity can be shown to the user, he is asked to turn the GPS on. I have implemented this as the first thing, because it takes a while before a GPS signal is established and I want to have a position ready before the user begins to use the application. Therefore the TravelBuddyActivity implements a LocationListener. I will comment on the use of the LocationListener later, for now, the permission for GPS is given and we proceed with the application.
The main screen is shown, see figure 5.3, and there are two options and thereby two use cases - either TripFromAtoB or NextDeparture. I will start by looking at NextDeparture.

5.3 Use case: Next Departure

When the "Next Departure" button is clicked, a new activity is started. This is the NextDepartureActivity class.
The way I wanted to solve (and started off with doing) the Next Departure request, was to show the nearest 5 stops to the user's location and he could choose the stop he was interested in, or if he was interested in another location, an input field would be above the nearby stops, so the user can specify if he desires another location than the one found by the GPS. After choosing a stop, a new activity starts and a list of departures from this stop is shown.
While this arrangement gives you flexibility and options, it also gives you a lot of choices. Choices which can't be illustrated, but explained by sounds. There-

fore, this was a bad design for the primary user. After talking with the client, we agreed that this should be changed. So this is how it works instead.

The user has two options now - either to just search and find the nearest stop using the current location, time and date or he can specify these, if he is planning a trip for later, as described in the use cases in chapter 4. The user interface for the NextDepartureActivity is shown in figure 5.4.

When interested in getting the departures from a stop, two URL requests to Rejseplanen need to be performed. In order to ask for a stops departures, a stop ID is needed for the corresponding stop. This is the first API call. The API needs coordinates, so if the user doesn't provide a location, the GPS coordinates are used. The URL then looks like this:
`http://<baseurl>/stopsNearby?coordX=12565796&coordY=55673063`
`&maxRadius=1000&maxNumber=30`

This URL is Rejseplanen's stops nearby service and it returns an XML with stop locations and their coordinates based on the provided coordinates.
If the user desires to make the search from another location, we have to find the coordinates first in order to find the stops nearby. When we have the coordinates of the location, we can repeat the API for getting a stop ID. Here is Rejseplanen's Location service, which returns a list of locations and the corresponding coordinates based on the user input.
`http://<baseurl>/location?input=userinput`

Here, I make the choice for the user. The two API requests return an XML file, with several locations or stop ID's. I make the choice that we are interested in the first location in the case of using the Location service and the first coordinates in the stop nearby service, and continue to work with them. Otherwise, I would have to display the different possibilities to the user, and we agreed to avoid the many options. So the application takes care of the choices itself.
As mentioned before, I wanted to include these options, but keeping the user in mind, we want to get rid of them.

When making these API calls, I discovered that Rejseplanen's XML files I got as response weren't necessarily the same every time. This made it difficult to get the desired stop ID and coordinates.
For instance, when making a location API request, the XML returns a list of locations, holding different elements, StopLocation and CoordLocation, which are sorted in a specific way. Both elements have coordinate attributes, but the StopLocation also has a stopID attribute. Rejseplanen's documentation doesn't have the explanation of these, so I figured it was smart to use StopLocation, because I could directly get the stopID and thereby always get the needed information by one API call only.

So, my first try was just to find the first occurrence of StopLocation and thereby find the needed stop ID.
Eventually, when trying different locations, I got some errors. First of all, there wasn't always a StopLocation element, so I couldn't get any result. So I started researching the XMLs, and found this example. If the user input is "Nørreport", here are the first occurring elements of StopLocation and CoordLocation:

Listing 5.2: StopLocation example

```
<StopLocation name="Nrreport st" x="12571306" y="55683050" id="008600646"/>
```

Listing 5.3: CoordLocation examplelabel

```
<CoordLocation name="Nrreport 7500 Holstebro, Holstebro" x="8619972" y="56363102"
     type="ADR"/>
```

The StopLocation is in Copenhagen, while the CoordLocation is located in Holstebro, so the two results are totally different. The StopLocation returns a stop name with a resemblance to the user input, while CoordLocation returns a location with a resemblance to the user input.

After many different try outs, I found out that the first element was always the best match, so the elements are sorted after the best match. So instead of checking the element name if it is CoorLocation or StopLocation, I simply use the first element.
Doing this, I encountered another problem. The XML holds some elements with the string "#text", which are not shown in the XML file and thereby can't be seen - they come from the whitespaces of the original XML. It took me a while to find out that this caused the trouble, but I ended up with a method, getCoordinate(String location), which returns a coordinate object from the user provided location:

Listing 5.4: XML parser - from a location to coordinates

```
Document doc = getXMLfromURL(url);
Node firstNode = doc.getFirstChild();
Node node;
if(firstNode.getFirstChild().getNodeName().equals("\#text")){
    node = firstNode.getFirstChild().getNextSibling();
}else{
  node = firstNode.getFirstChild();
}

String xCoor = node.getAttributes().getNamedItem("x").getNodeValue();
String yCoor = node.getAttributes().getNamedItem("y").getNodeValue();
coor = new Coordinate(Integer.parseInt(xCoor), Integer.parseInt(yCoor));
```

The XML is firstly stored as a document and then it is divided into nodes for each element. The code snippet shows that we get the first element which is holding the coordinate attributes and how to get the attributes from the XML elements. When we have the coordinate attributes, we perform the next API call to find the stop ID matching the coordinate set.
One option that our client wanted to keep, was to choose whether he is interested in a bus stop, train station or else.
Meanwhile, I found out that this isn't exactly possible. When adding this in the URL, it doesn't change the result. The reason is that Rejseplanen looks at a stop as a potential stop for every transport vehicle - so in theory a bus could stop at any station. So this is actually not possible for this kind of API call.
The XML for the stops nearby doesn't hold information about which kind of stop the ID belongs to.
Having one stop ID, we can get the departures. This is done by using the Departure board ReST service:
`http://<baseurl>/departureBoard?id=8600626&date=19.09.10\&time=07:02&useBus=0`

This API call will return the next departures from the given stop ID. Now it is possible to choose whether you are interested in bus departures or train, but this is only useful on big stations, where you have all kinds of transport vehicles, so this is actually removed from the UI, for simplicity.

Once again, after the API call is performed, it is time for the document parsing, which of course is also done asynchronosly, so the method for this is also located in the RejseplanenAsyncTask class.
As mentioned before, the XML is stored as a document. Now it is time to parse the XML document, so the desired results can be presented. Listing 5.5 shows a snippet of the XML:

Listing 5.5: XML snippet from the departure board service

```xml
<Departure name="Re 5837" type="REG" stop="Kbenhavn H" time="14:06" date="
     30.08.12" direction="Lejre st">
  <JourneyDetailRef ref="http://xmlopen.rejseplanen.dk/bin/rest.exe/journeyDetail
       ?ref=147885%2F58508%2F324836%2F113128%2F86%3Fdate%3D30.08.12%26"/>
</Departure>
<Departure name="H" type="S" stop="Kbenhavn H" time="14:06" date="30.08.12"
     direction="Frederikssund st">
  <JourneyDetailRef ref="http://xmlopen.rejseplanen.dk/bin/rest.exe/journeyDetail
       ?ref=960780%2F331618%2F953564%2F156536%2F86%3Fdate%3D30.08.12%26"/>
</Departure>
<Departure name="Bus 6A" type="BUS" stop="Hovedbanegrd/Vesterbrog." time="14:06"
     date="30.08.12" direction="Emdrup Torv">
  <JourneyDetailRef ref="http://xmlopen.rejseplanen.dk/bin/rest.exe/journeyDetail
       ?ref=556386%2F192409%2F243110%2F63908%2F86%3Fdate%3D30.08.12%26"/>
</Departure>
```

```
<Departure name="Bus 1A" type="BUS" stop="Hovedbanegrd/Tietgensbro" time="14:06"
     date="30.08.12" direction="Hellerup st. (bus)">
  <JourneyDetailRef ref="http://xmlopen.rejseplanen.dk/bin/rest.exe/journeyDetail
       ?ref=424629%2F144044%2F90398%2F96347%2F86%3Fdate%3D30.08.12%26"/>
</Departure>
```

Now, we are interested in every Departure element of the shown XML and want to have its information. It is easier to look at every Departure as an object; therefore I have created a data class, Departure, to store the information in. A Departure object has a vehicle name, direction and a departure time. Here is a code snippet of how a Departure object is created.

Listing 5.6: Creating a Departure object

```
Document doc = getXMLfromURL(url);

NodeList listOfDepartures = doc.getElementsByTagName("Departure");
Departure[] departuresArray = new Departure[listOfDepartures.getLength()];
for(int s=0; s<listOfDepartures.getLength() ; s++){
  Node node = listOfDepartures.item(s);
  String vehicleName = node.getAttributes().getNamedItem("name").getNodeValue();
  String direction = node.getAttributes().getNamedItem("direction").getNodeValue
       ();
  String departureTime = node.getAttributes().getNamedItem("time").getNodeValue()
       ;

  departuresArray[s] = new Departure(vehicleName, direction, departureTime);
}
```

The snippet shows how to get the attributes from the XML elements. For every departure we get the name, direction and time from the XML and then we can create a Departure object. When the code finishes, we have an array of Departure objects - departureArray. The array holds the information that we want to present to the user.

Now we are done with RejseplanenRest class with all the API services and RejseplanenAsyncTask class which is the asynctask class and we want to return to the main thread. The difficult part is not to return to the thread, but to return with the found information, in this case with the departure array.

I have tried this in several ways. Sending an empty array from the main thread to RejseplanenAsyncTask and fill it up - this returns an empty array again in the main thread, although I have checked that it is filled before leaving RejseplanenAsyncTask. Another way was to use the onPostExecute method of asynctask in the RejseplanenAsyncTask class. This actually worked, but the way it was done, was to start a new activity from onPostExecute and send the array with the intent.
I later found out that this is a bad design - this way, I changed the GUI from

a asynctask class, instead from the main thread. This mixed up the different activities and it was difficult to keep track of the activities and return to the right ones.
The last and best solution that I found, is to use a callback method.

Before starting the API calls, we are in NextDepartureActivity class. From here we start the AsyncTask and go to the RejseplanenAsyncTask class. When we are at the onPostExecute() method we return to the UI thread and send data to the callback method Finished() in NextDepartureActivity. In the callback method, I create a new activity with the results, PossibleDeparturesActivity.

It is a bit troublesome to send data between activities. Intents are used for this, as described in chapter 1. It took me a while to figure out how the intents are used; it is easy to send one object, but an array of objects is another thing. Implementing Serializable in the data class solves this problem. So when we are in PossibleDeparturesActivity, the sent array is extracted. Therefore a new array must be created and afterwards be filled with the intent data.
After this, we are ready to show the result screen, showed in figure 5.5.

Overall, looking at the way I have designed a solution to this use case, there is a difference in designing it for a visually impaired user and as a accessible application compared to a regular travel application.
The main difference is that some of the options are actually removed, in order to make it more accessible, while more options are desired for a regular user, or at least, the possibility is there. Some of these options, could be added as settings whether the user wants to have few or more options, but I haven't implemented this.

5.4 Use case: Trip from A to B

Now, I will be looking at the case where the user wants to plan a trip from A to B. Compared with the NextDeparture use case, the user now has to provide a destination, assuming he knows where he is going. This is the only mandatory information that needs to be provided - the others are optional.
From the main screen, the button From A to B is clicked and we come to the next activity, FromAtoBActivity, which can be seen on figure 5.6.

This use case needs several more URL calls than the earlier use case. The coordinates for the destination are found like described in the earlier use case. If a location is specified we need an API call for the coordinates, otherwise the GPS coordinates are used.

There are several ways of finding a trip from A to B, but I have chosen to use the coordinates, like this:
http://xmlopen.rejseplanen.dk/bin/rest.exe/trip?originCoordX=12530950&originCoordY=55795058
&originCoordName=your+location&destCoordX=12571306&destCoordY=55683050
&destCoordName=your+destination&useBus=0

I have done this, because I believe that with coordinates we get the most precise result. I will provide a snippet of this XML, in order to explain its parsing, because it is more complex than the earlier use case.

Listing 5.7: XML snippet from a trip service request

```
<Trip>
  <Leg name="til fods" type="WALK">
    <Origin name="Kbenhavn H" type="ST" time="14:43" date="07.08.12"/>
    <Destination name="Nrreport st. (bus)" type="ST" time="15:02" date="07.08.12"
         />
    <Notes text="Varighed: 19 min.;(Afstand: ca. 1,5 km);"/>
  </Leg>
  <Leg name="Bus 150S" type="EXB">
    <Origin name="Nrreport st. (bus)" type="ST" routeIdx="0" time="15:02" date="
         07.08.12"/>
    <Destination name="DTU/Rvehjvej" type="ST" routeIdx="13" time="15:25" date="
         07.08.12"/>
    <Notes text="Retning: Kokkedal st (bus);"/>
    <JourneyDetailRef ref="http://xmlopen.rejseplanen.dk/bin/rest.exe/
         journeyDetail?ref=856104%2F286702%2F311200%2F129768%2F86%3Fdate%3D07
         .08.12%26station_evaId%3D461%26"/>
  </Leg>
  <Leg name="til fods" type="WALK">
    <Origin name="DTU/Rvehjvej" type="ST" time="15:25" date="07.08.12"/>
    <Destination name="Rvehjvej, DTU" type="ST" time="15:27" date="07.08.12"/>
    <Notes text="Varighed: 2 min.;"/>
  </Leg>
</Trip>
```

For the earlier use case, I implemented a data class, Departure - I will do the same now. At first I made an abstract class, for the common fields of Departure and the new data class, Trip, but then I found out that they do not have that much in common because of the diversity of the XML files. A trip consists of several legs as it can be seen on 5.7 - these are the steps of the trip. So, a result should return a list of trips, which consists of a list of steps. To make this manageable, there is another data class, namely TripStep.

Listing 5.8: Creating a Trip object

```
Document doc = getXMLfromURL(url);
doc.getDocumentElement ().normalize();

NodeList listOfTrips = doc.getElementsByTagName("Trip");
```

```java
Trip[] tripsArray = new Trip[listOfTrips.getLength()];
int numberOfShifts;
//We find all the trips and are ready to extract the steps from them
for(int b=0; b<listOfTrips.getLength(); b++){
  numberOfShifts = 0;
  NodeList listOfTripSteps = listOfTrips.item(b).getChildNodes();
  ArrayList<TripStep> stepArray = new ArrayList<TripStep>();

  for(int j= 0; j<listOfTripSteps.getLength(); j++){
    if(listOfTripSteps.item(j).hasAttributes()){

      NodeList listOfTripStepsChildren = listOfTripSteps.item(j).getChildNodes();
      TripStep step = new TripStep();
      Node stepNode = listOfTripSteps.item(j);
      step.setType(stepNode.getAttributes().getNamedItem("type").getNodeValue());
      step.setTransportName(stepNode.getAttributes().getNamedItem("name").
          getNodeValue());

      for (int s=0; s<listOfTripStepsChildren.getLength(); s++){
        Node node = listOfTripStepsChildren.item(s);
        if (node.getNodeName().equals("Origin")) {
          step.setOrigin(node.getAttributes().getNamedItem("name").getNodeValue())
              ;
          step.setOriginTime((node.getAttributes().getNamedItem("time").
              getNodeValue()));
          step.setOriginDate((node.getAttributes().getNamedItem("date").
              getNodeValue()));
        }

        if(node.getNodeName().equals("Destination")){
          step.setDestination(node.getAttributes().getNamedItem("name").
              getNodeValue());
          step.setDestinationTime(node.getAttributes().getNamedItem("time").
              getNodeValue());
          step.setDestinationDate(node.getAttributes().getNamedItem("date").
              getNodeValue());
        }

        if(s == listOfTripStepsChildren.getLength()-1){
          step.setDestinationCoor(tripDestinationCoor);

        }
      }
      stepArray.add(step);
    }
  }
  tripsArray[b] = new Trip(stepArray.toArray(new TripStep[0]), numberOfShifts);
}
```

This piece of code shows how the Trip objects are created and how information is extracted from the XML document. We make a node list of trips, and for every trip we make a list of TripSteps. Now, every TripStep has some attributes that we are interested in, but it also has an origin and destination element. We

want to store these separately, which is done by checking if the trip step node's child is an origin or a destination element.
So the result of this method is an array of Trips, where every Trip has an array of TripSteps.

Again like in the earlier use case, after the API calls are performed and we are in onPostExecute() we go back to the UI thread and the callback method Finished() is called, now with a list of trips. Of course the data classes Trip and TripStep need to implement Seriazable like before.
The result of this use case is a list of the next three trips to the desired destination and can be seen on figure 5.7.
What is shown here is a list of trips. For every trip there is a button holding the origin of the first step and the destination of the last step of the trip. This is handled in PossibleTripsActivity class.

5.5 Use case: Trip Description

The earlier section explained how a request for a Trip from A to B is implemented.
While constructing the array of Trips, the trip description is also stored. If the user chooses to se its details, a new activity is started, TripDescriptionActivity. Once again we need to use the intent in order to send the information to TripDescriptionActivity from PossibleTripsActivity. Here, the screen shows a list, showing every step of the selected trip and can be seen on figure 5.8.

5.6 Loading Screen

Without having graphics to rely on, it is difficult to know when a program is doing something. Therefore, I have added a loading screen, which stops the background and reads out loud that it is loading.
The loading screen is shown right after the API call is started and dismissed in the Callback function Finished(), before starting the result activity.
Depending on how long it takes for the asynctask to run, the loading screen is shown. So if it runs fast, the loading screen can't manage to read the whole command - this can be confusing, leaving the user without knowing what screen he is at. A sleep() command could be used to get enough time to read it, but this would slow the program. Here it should be considered what is most important; speed versus accessibility.

I have removed the sleep() command and chosen the speed here, because the user can find out what is on the next screen by exploring it, although the first times he uses the application would perhaps leave him confused with this.

5.7 Additional

After working with the application for so long, I decided to spend more time on adding some important and useful functionalities to the application. In this section I will cover two extra functions that I have implemented - both of them are important for the client.

5.7.1 Directions and Google Navigation

In the introduction Chapter, I mentioned, that I would like to implement the possibility of getting directions by foot. This wasn't a use case, but the user said it would be a nice thing to add to the application.
When a trip is selected, we get a list of steps in order to travel from A to B. For every step there is a button. When selecting a step where the user has to walk from in order to get to the next step, I want to add the directions to the stop. This could not be provided by Rejseplanen's API, but it can be done with Google Navigation.
The way it is done is by using the Google Navigation application. This requires that the application is installed on the smartphone, but this is a standard application on all new Android smartphones.
To start Google Navigation, I have added an onClickListener on the buttons, where the user needs to walk. The onClickListener starts a new activity, Google Navigation, when such a button is clicked.

There are many ways of starting Google Navigation and it is done by one line of code, so it is not that complicated. Once again, an intent is used to pass the desired location or coordinates to Google Navigation.
The difficult part here is to start Google Navigation in ”walk” mode. Usually, the user specifies this once Google Navigation is started, but I want to do it as easy as possible for the user, spare him the trouble and start Google Navigation in ”walk” mode.

After a lot of reaserch, I found only one way of doing this. Listing 5.9 shows how it is done.

Listing 5.9: Starting Google Navigation Activity

```
startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse("google.navigation:ll=" +
             ((double)stopCoor.getStopCoor().getLat()/1000000) + "," + ((double
                 )stopCoor.getStopCoor().getLon()/1000000) + "&mode=w")));
```

When started this way, I have encountered one problem. One can only send a destination along with the intent as a parameter - Google Navigation will use the GPS coordinates as the users current location.
I tried other ways of starting it, which allowed me to provide both location and destination, but then the problem was to start it in "walk" mode - Google Navigation required a specification of user mode before one could get directions.
I chose to use the first option and here is why. First of all, the application is automatically started in "walk" mode.
Second of all, optimally, you are interested in the directions when you need to get them - and you need them when you have to walk the distance from the current location to the destination. So when turning on Google Navigation, the location is your current location - if you get the directions to step 3, while actually you are at step 1, the location will be the same as step 1. So the directions should be used when you are at a certain step, otherwise they could be confusing.
It is easy to navigate between Google Navigation and TravelBuddy just by goin back and forth as usual.

5.7.2 Get off the bus!

In the Project Analysis chapter, the challenges of traveling are listed and one of these is when to get off a transport vehicle.
It would be great to be able to get in the bus or another vehicle, without the need of asking the driver to inform you when you have to get off and getting rid of the risk of the driver forgetting to inform you about this. In many public transports, the stops are read out loud, but not always. So therefore I would like to give the possibility of this to the user.
This problem isn't listed as a use case, but is something I have added because of the importance of the problem.
The main issue in solving the problem is not to implement the functionality, but to evaluate the pros versus the cons in having the functionality in the application.
I will start by discussing the implementation of the solution and then I will look at the pros and cons.

We are interested in finding out when the location is changed to the arrival coordinates - so the user can be warned and be ready to get off.

In order to do this, the location must be checked regularly and be compared to the stop location.
When the user gets to Trip description activity he can follow the steps of the trip. After getting to the stop using Google Navigation, the next step is to take the bus or another vehicle. If pressing the button of the step of a vehicle ride, the application will know it is time to check the distance to the stop regularly.

The way I have solved this is by having another activity, ReachStopActivity, which is started when a vehicle ride step button is chosen.
A new screen appears which lets the user know that the bus is driving towards the destination. When he is close to the destination he needs to be warned.
So first of all I need to get the information about the route and send it to ReachStopActivity. What I need is the stop coordinate. The XML snippet on listing 5.7 shows that besides the two elements of a trip, Origin and Destination, some of the trips have another element, namely JourneyDetailRef. This element can be found for every TripStep that involves a public transport. The ref attribute of JourneyDetailRef element is an URL, which has further journey details.
If we use listing 5.7 as the example and need to know when we have reached the stop, we would need to get the coordinates. So I will modify the parsing shown in listing 5.8, in order to also store the ref attribute from the JourneyDeatilRef.
If we look at the XML snippet from the new API call, JourneyDetailRef, there might be some way to get the destination coordinates.

Listing 5.10: XML snippet of JourneyDetailRef

```
<Stop name="Nrreport st. (bus)" x="12572933" y="55683985" routeIdx="0" depTime="
    15:02"/>
<Stop name="Slvtorvet" x="12573868" y="55688956" routeIdx="1" arrTime="15:04"
    depTime="15:04"/>
<Stop name="Blegdamsvej/Tagensvej" x="12567108" y="55694089" routeIdx="2" arrTime
    ="15:06" depTime="15:06"/>
<Stop name="Fredrik Bajers Plads" x="12562236" y="55696030" routeIdx="3" arrTime=
    "15:07" depTime="15:07"/>
<Stop name="Universitetsparken" x="12562128" y="55702017" routeIdx="4" arrTime="
    15:09" depTime="15:09"/>
<Stop name="Vibenshus Runddel" x="12561975" y="55706044" routeIdx="5" arrTime="15
    :10" depTime="15:10"/>
<Stop name="Lyngbyvej/Haraldsgade" x="12561705" y="55710071" routeIdx="6" arrTime
    ="15:11" depTime="15:11"/>
<Stop name="Hans Knudsens Plads" x="12559845" y="55712930" routeIdx="7" arrTime="
    15:13" depTime="15:13"/>
<Stop name="Ryparken st. (bus)" x="12557813" y="55715690" routeIdx="8" arrTime="
    15:14" depTime="15:14"/>
<Stop name="Tuborgvej/Lyngbyvej" x="12550325" y="55728248" routeIdx="9" arrTime="
    15:16" depTime="15:16"/>
<Stop name="Kildegrds Plads" x="12538621" y="55737866" routeIdx="10" arrTime="15
    :17" depTime="15:17"/>
<Stop name="Brogrdsvej/Lyngbyvej" x="12523313" y="55752896" routeIdx="11" arrTime
    ="15:20" depTime="15:20"/>
```

```
<Stop name="Klampenborgv/Hgr.Motorv." x="12523780" y="55775108" routeIdx="12"
     arrTime="15:24" depTime="15:24"/>
<Stop name="DTU/Rvehjvej" x="12528311" y="55786920" routeIdx="13" arrTime="15:25"
      depTime="15:25"/>
```

The XML snippet shows that there are coordinates for every stop, which can be found by the routeIdx attribute.
Every departure has also a routeIdx, shown on listing 5.7. So if these two are compared, the coordinates can be found.
Now that we have the coordinates, they will be parsed along with the other attributes and send to TripDescriptionActivity and then to ReachStopActivity. Having the needed data, we can proceed.

ReachStopActivity class implements LocationListener, which has an onLocationChanged method. This method is exactly what we need. Having a location manager, we can specify how often it needs to request for updates. I have used 1 second as the update time.
So every second we go to onLocationChanged method. Here, the GPS provided location is compared to the stopLocation where we are heading.
The user needs to be warned in time, not too soon, because there could be stops which are located close to each other, so he could get off too early, and not too late, so he can't make it to pres stop. So this could be defined as 100 meters before the stop, the user should be warned, either with saying it or vibration (this could be a setting).
To calculate the distance between two coordinates, various mathematical formulas could be used. Of course, the distance is an indicative distance, calculated as an air line and is not the distance the vehicle needs to drive before it is a the stop.
If the distance is no more than 100 meters, the user should be alerted that he needs to get off the bus.
I reckon there is no need in calculating the distance in a complicated way, because the only thing we need it for is to check when we are 100 meters from the stop point. This is such a short distance, that it almost is the same if it is the route or an air line.
So when the user is less than 100 meters from the stop he is warned that he needs to get off. This is done by an alert box, reading *Destination is reached within 100 meters*. I have considered to increase the distance to 200 meters, so the user has more time to react in - but this depends on how close the stops are - if they are any closer than 200 meters, the application would give you the wrong stop, so there is another thing to be considered.

Another way of solving this problem is to follow the route. Instead of checking the distance to the stop we are heading to, it could be checked when the stop before our destination is passed - this would automatically mean that the user

should get off at the next stop.
There are several problems to this. If the stops are very far away from each other, the user could be confused and doubt if he has missed something - or could be confused if the bus stops at a traffic light - he could think that it was the stop and be confused about why the doors don't open and then doubt if he had pressed the stop at the bus.
This is another option, which could be optimized, but I have chosen the first solution as described.

In the beginning of this section I wrote one needs to evaluate pros and cons. What I have stated until now are clearly pros - giving the user the possibility of knowing when to get of a transport vehicle.
The single bad thing about this functionality is the battery usage. Every smartphone user knows that you need to charge your smartphone often enough, and do it a lot the more you use it. GPS service is the very application which consumes much battery - and that is what we need in this application.
If a bus ride takes one hour, you would need the GPS to be active the whole time. I was thinking about implementing the functionality of performing the destination check rarely when far away and more often when the distance decreases down to 1-2 km. But even if you implement that the check is performed more often when you get closer to the destination, the GPS is still on and active. This consumes the battery time.

There is no guarantee that the smartphone will have enough battery to direct you the rest of the trip if you use it up on a simple function, to get off the bus.
This functionality is important, but it takes a lot of battery time, so perhaps it is not such a great help after all.
However, now it is a function of TravelBuddy and should be used wisely.

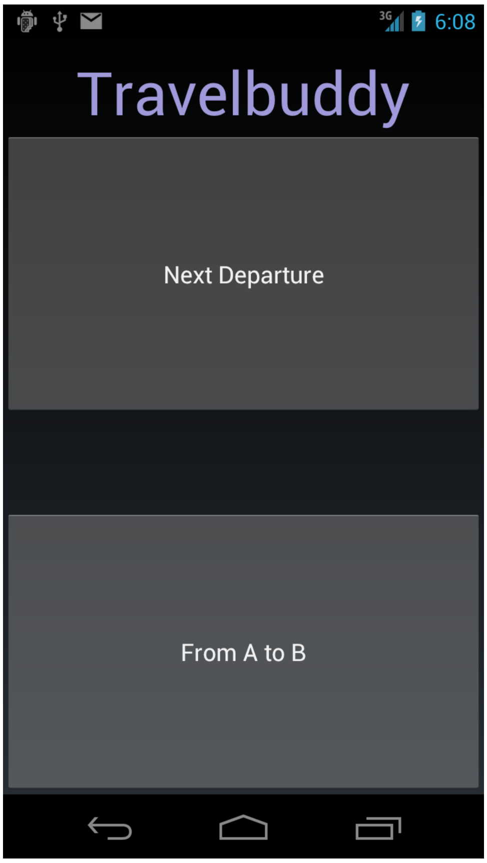
5.8 User Interface

During the last sections, I have commented on the implementation of the use cases. In this section, I will comment on my choices on the user interface.
I have observed how our client uses his smartphone, during our meetings. This gave me an insight to how difficult it is to use it. Although you have TalkBack turned on and it is easy to navigate between the applications, it is difficult to use them.
Almost none of the clients applications are developed for visually disabled users, so when they were developed, his needs were not prioritized. When TalkBack is turned on, sometimes everything on the screen is read out loud, without any

user input.
This is very confusing and not useful at all. I have tried to use these observations when designing the user interface for TravelBuddy. Following is a review of the thoughts and choices I made when designing the user interface.

When starting the application, there are two possibilities. If the GPS is not turned on, a popup window is shown, asking if the user would like to change the settings. If this is the case, a new activity starts, which is a default activity for changing the GPS settings. There is nothing I can change about the design of this, because it is the default handler of the GPS settings and needs to be done in order to use the GPS.
When the user turns on the GPS TravelBuddy is ready to be used. The main screen is shown and can be seen on figure 5.3. The design is very simple and there is almost nothing on the screen except the few necessery buttons.

Figure 5.3: UI - Main screen

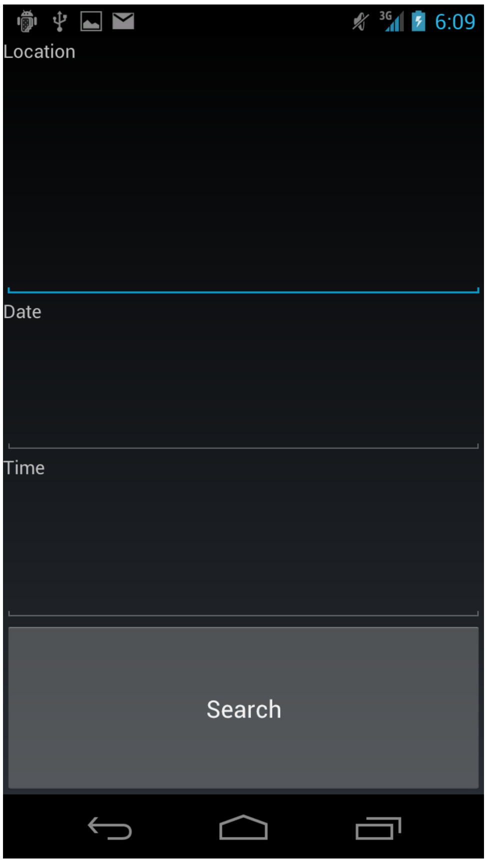


The main screen has a view for the name of the application - just for information purposes. Additional to this, there are two very large buttons, which are easy to find because of their size. Everything is read out loud if invoked by touch. If the buttons are touched, they will be clicked the second time they are touched. I want to minimize the number of steps the user needs to go through, before he gets the wanted information.
The layout is black and simple, with not too many options - but therefore it is easy to use by any other user also.

Earlier, I discussed the options I wanted to add to NextDepartureActivity. It wasn't much, but the few adjustments and additions I had, the client said were

too much.
The layout of NextDepartureActivity can be seen on figure 5.4.

Figure 5.4: UI - Next Departure

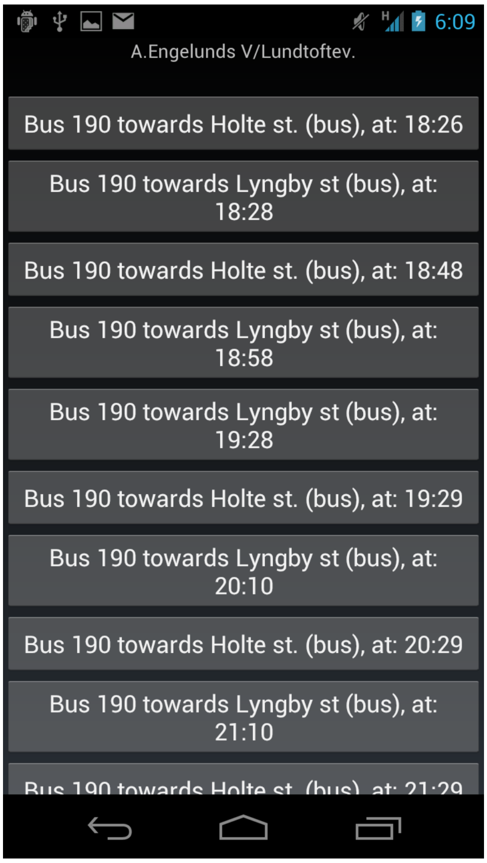


I have kept the simple style and big views, to make it easier for the user to explore. The simplest way to get to the stop is just to use the large Search button and use the general settings. It is located at the bottom of the screen, so the user can find it fast and doesn't have to go through the other views in order to find it.
Enabling TalkBack, the user knows where he is on the screen. The only things that I have implemented to be read are the input fields and the buttons. This was to not confuse the user further more - for instance the first field on the

Next Departure screen is ”Location”. This is just information for a user with no visual disability and is not read out loud by TalkBack. Also, when a input field is invoked, like ”Location”, the TalkBack reads: *Input your location. Edit box.* The last part, ”Edit box” is something Android adds to input fields, which I wanted to remove. But the user explained that he was often confused by other applications, whether fields just were information fields or input fields, so I left the extra explanation.

A thing which would be unusual to a regular user, would be the input of time and date.
The way this is done in TravelBody, is just by writing the time as one number. For instance ”12.45” is just written as 1245. The same for the date. It is of course punctuation insensitive. Usually, for normal apps, this could be divided in two different fields, so if you got the hours wrong, you do not have to change the minute. I suggested this to the client, but he thought it would be smartest to leave it as one big field. After the search the result screen is shown, and can be seen on figure 5.5.

Figure 5.5: UI - Result from a Next Departure request



6:09

A.Engelunds V/Lundtoftev.

Bus 190 towards Holte st. (bus), at: 18:26

Bus 190 towards Lyngby st (bus), at: 18:28

Bus 190 towards Holte st. (bus), at: 18:48

Bus 190 towards Lyngby st (bus), at: 18:58

Bus 190 towards Lyngby st (bus), at: 19:28

Bus 190 towards Holte st. (bus), at: 19:29

Bus 190 towards Lyngby st (bus), at: 20:10

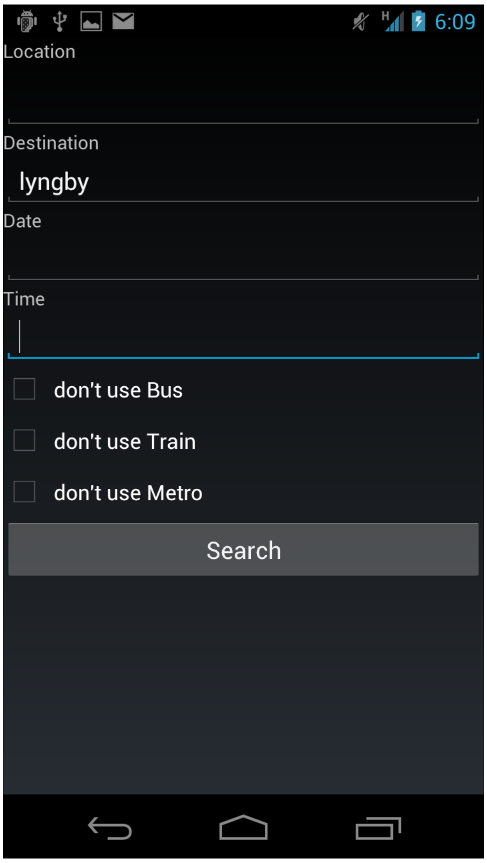
Bus 190 towards Holte st. (bus), at: 20:29

Bus 190 towards Lyngby st (bus), at: 21:10

Bus 190 towards Holte st. (bus), at: 21:29

Additional to this, Trip from A to B has a few more options, which can be seen on figure 5.6.

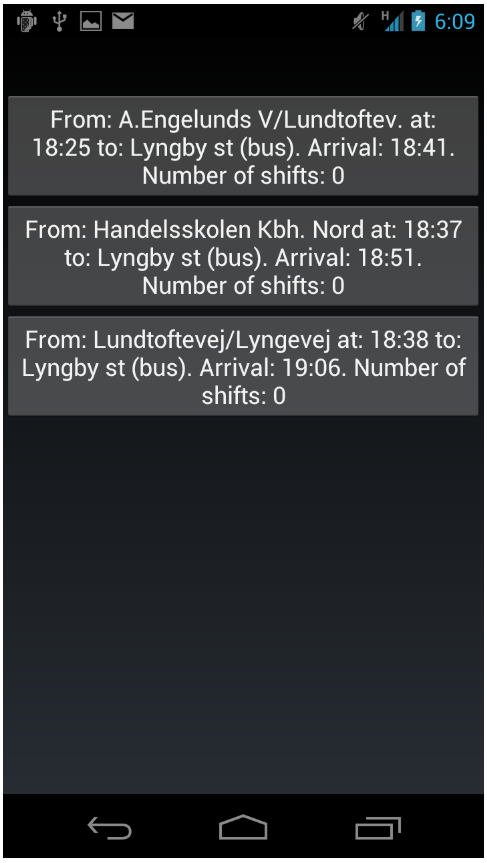
Figure 5.6: UI - Trip from A to B



There are three checkboxes, which are used for choosing a transport vehicle. I think this is the best way of adding these options.
Another way could be to add it as a setting, but this is probably inconvinient to change every once in a while. So I stayed with checkboxes.
So much for the input. The output on the other hand, is almost the most important part. As I mentioned earlier, output can appear when the user presses the "Search"-button. I wrote about how I would like to have some output, as instant service based on your GPS location. This is difficult for the user to examine though. The way that it is now, the user follows the application step by step, and knows what is going on - logically, he gets results when he has searched for them.

The output for Next Departure generates buttons from the XML the API returns and is filled by the information. The buttons can be clicked and Google Navigation starts with direction to the bus stop.

Figure 5.7: UI - Result from a Trip from A to B request



6:09

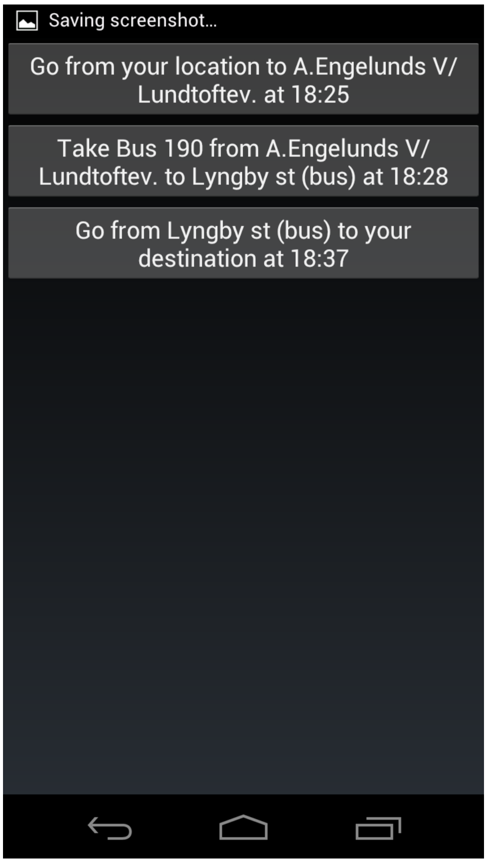
From: A.Engelunds V/Lundtoftev. at: 18:25 to: Lyngby st (bus). Arrival: 18:41. Number of shifts: 0

From: Handelsskolen Kbh. Nord at: 18:37 to: Lyngby st (bus). Arrival: 18:51. Number of shifts: 0

From: Lundtoftevej/Lyngevej at: 18:38 to: Lyngby st (bus). Arrival: 19:06. Number of shifts: 0

This works the same way for "Trip from A to B". Except when clicking on a trip we come to a trip description screen, which looks the same way - it has buttons, based on how many steps the trip has. An addition is also that if a step is a ride with public transport, a new activity starts informing the user when he needs to get off the transport vehicle.

Figure 5.8: UI - Trip Description



Saving screenshot...

Go from your location to A.Engelunds V/ Lundtoftev. at 18:25

Take Bus 190 from A.Engelunds V/ Lundtoftev. to Lyngby st (bus) at 18:28

Go from Lyngby st (bus) to your destination at 18:37

The output for both use cases is showed as a list of buttons. Every button can be clicked and lead either to Google Navigation or an activity showing when to get off the vehicle.
All the results are shown in form of a list - I chose this first of all so the result activities would stay true to the layout of the other activities, but there is also a functional aspect to this. The user can explore the information row by row and thereby control what is read out loud and get a repetition of the information if he needs it.

For the result of TripFromAtoB use case we discussed adding an option which should sort the results by different preferences - sort by number of transport shifts, duration or time for starting the trip. This could be added easily, but the problem is that the trip API only returns three trips at a time. I reckon that there is no need to add a sorting algorithm for three items. Even if I added an option for finding the next three trips, there would only be 6 trips. Perhaps it would make sense to sort them, but this would mix up the older results, making it more confusing than helpful.

Overall, I have chosen for the design to have as little content and options as possible and also make every view large enough, so the user can't get confused about which view he is at.

Another thing I have tried to accomplish is to use the same layout for every activity and make the application layout consistent. Every activity should be explored from top to bottom and has a form of a list layout. There is only one view per line - meaning there is no need to explore the screen to the sides. This should help the user have an overview of how to navigate on the screen. This layout idea is used on both input and output.

The most used view in the application is the button view. I have used buttons in order to create the feeling of clicking around in the application - also because when a button is used, the user is informed that it is clicked, TalkBack reads "Clicked".
So input fields are used for input purposes, because they read "Edit Box" and buttons are used when you get from one activity to another - this is consistent in the application and will make the user feel safe and familiar with the application.

Above the input fields, there are labels explaining what needs to be provided in the input field. Although they are there, they do not read anything - I have added them with the purpose of making it possible to use TravelBuddy without TalkBack, or allowing a not visually impaired to help a user the first time he uses the application, so a blind user would not in fact know of their existence. The same goes for the buttons - they have text and information, and this is for the same purpose as before - a not visually impaired user could navigate around in the application. So the only thing that is read by TalkBack is actually when an input or output view is invoked. When the application is designed like this, the extra information like "Edit Box" and "Clicked" information shouldn't be needed. Never the less, I have left them because our client was fond of this. This is because other applications he has used also read the labels, leaving him without information about what kind of view he is at and what he needs to do. Although I have optimized this in my application, he is uncertain because of his experience from other applications, so I have left this of consistency reasons.

Chapter 6

Test

In order to make the application robust it has been through a lot of testing.

Testing this kind of Android appplication can't exactly be done as general software testing. First of all, there are not many Unit tests to be done, when using the internet to get the results. Of course, a hard-coded XML file could be used to see if the output is as expected. On the other hand, this wouldn't be so efficient, when having many different and many forms of XML files. Of course Unit tests of different input could be tested, but this is taken care of when checking if the XML files return anything. It will be commented on later on in this chapter.

Besides, this project has the accessibility as the main focus and therefore manual testing was necessary.

6.1 Manual Testing

While programming TravelBuddy, the client has performed tests continuosly and given feedback in order to improve the program in a way that he finds fitting.

After the last revision I got his feedback. He had some few comments, but

otherwise he thought that the application was good and easy to use.
One of the comments was that the user somehow should be informed that the GPS location was used if no location is provided.
I thought that it was enough that the permission for the GPS was asked when starting the application, but this is of course not something that should be assumed.
There are various ways of fixing this problem. I figure that it is too much to add this information as content description, but could be added when asked for GPS permission. A better choice could be to add it as an information menu in the settings, but this is something I would like the client to specify what he thinks is more useful.

Another thing that he would like to be a setting is the time of the trips and departures. The way that it is now, is that the the descriptions hold the time of the departure of the transport vehicle. The client's suggestion was to indicate the time until the the vehicle arrival in minutes instead - so he knows in how long it will arrive and for how long he needs to wait.

Personally I think that it is easier to plan a trip when having the time instead, but the problem is that a visually imapired user can't just check the time when he likes. It is easier to relate to the remaining minutes untill the departure instead of the time.
In order to change the time to minutes it would require to update the results so the minutes are a form of a countdown. Right now, there is not implemented a function to update the search result, but I will discuss the option of updating in chapter 7, Further Development and Perspectivation.

The update aside, the solution to the client's suggestion could be adding a setting for the arrival time presentation, so the different users can use whatever they prefer.

6.2 Other forms of testing

There is a test that an application needs to pass in order to get allowed to be uploaded on the Android Market. I have used this test for testing TravelBuddy. The test comes with the Android tools and is run from the command promt.

The test is called The Monkey and is used to stress test the application using the UI. It performs a provided number of randomly generated user events in form of clicks, touches etc.

I have used the Monkey on TravelBuddy, but no errors were reported. Not until I used 100.000 user events. After so many events, the Monkey performed a random search - which made the application crash.

The crash occured when the location string only consisted of non-letters. I thought that I had taken care of this, but the application performed the search and stopped when a Null-Pointer exception occurred. The Null-pointer occurred when an API call for finding coordinates to this location was performed - no location could match a non-letter user input.

After the testing, the error has been fixed by adding an alert box when the input returns an XML with an error. The alert box informs the user that the input is invalid.

Another crash occured after various runs of the Monkey test. Accidently, while the test was running, the user events turned of the internet connection. This means that the XML file can never be fetched and therefore all the methods which handle the XML file will return a Null-Pointer exception.
I have stated that TravelBuddy needs internet connection in order to fully function, but sometimes the internet connection is lost, which I haven't taken care of.
There are various ways of solving this situation. Most applications on the market simply let the user know that there is no internet connection and let the user cancel the application or try again - many times the connection is found right after the first try. So I would also solve the problem this way, by showing an alert box or similar, letting the user know what the problem is and give him the opportunity to try again or stop the search.

It was very efficient to use the Monkey in this case, but I reckon that it is better to use for an application with more focus on graphical user interface.

An obvious way of testing an Android application would be to test it on other devices. TravelBuddy is developed on Samsung Galaxy Nexus, but is also tested on a Samsung Galaxy S2.
What TravelBuddy requires is Google Maps, GPS and Android Accessibility. Any smartphone which doesn't support these functions would TravelBuddy not be conpatible with. Otherwise, the layout of TravelBuddy is not based on the size of the screen, but on percentage - therefore the layout would be the same on every smartphone and can both be used in horizontal or vertical orientation - as the user perfers. Of course the bigger the screen the better it is for this exact application.

Chapter 7

Further Development and Perspectivation

Other than the implemented and described functionalities, I would like to research on other problems I have thought about, but didn't have time to implement myself.

7.1 Your Friend's Address

A smart addition to TravelBuddy, could be to use an address of a contact in the contact list as a potential destination in a trip. This would require that the contacts address' are stored in the contact list correctly - which is a quite big assumption.

Just like Google Navigation is used in order to get directions, to launch the contact list would require starting a new activity. This is easy, but it would require many checks, whether the address is there, if it is of a valid format etc. It would be a nice addition, but it should be evaluated how much longer it takes to find the correct contact for a blind person than typing an address.

7.2 Which bus is it?

My research and contact to people with visual disabilies showed that a very important problem for them was to find the correct transportation vehicle. If there are two busses at the stop, which one should they take?

I have been researching on how this could be implemented in TravelBuddy. Using live updates from the internet is one idea - there is a way of getting the live updates, but one would have to gain access to this by contacting Rejseplanen.

This idea would remove some uncertainity but not all. The user could almost always be sure of when the bus has arrived. The smartphone could signal when the live update says it is time for the bus to arrive. But they are not always precise enough and this could be misleading for a blind person.

Another problem is if two busses arrive at the same time - the user would automatically walk over to the first bus, which isn't necessarily the one he is interested in.

My best suggestion to solving this problem, would be to use some kind of vehicle-to-smartphone signal. The vehicle would have some kind of device, which sends a signal to the smartphone, when it stops or is in range - or could beep more and more when the vehicle is getting closer.
This way, the user would be sure when the bus has arrived and be sure of which one to take.
Using blutooth or infrared signals don't use so much battery time, so this could be a good solution. It of course depends on what distance this can be done, but this is just a suggestion.
There are already various devices in public transports, for example for checking in - here could the sender be added.

7.3 Updating During a Travel

While programming TravelBuddy I was thinking about if it was smart to update a found trip. Here is an example: the user has found a trip that he would like to take and is looking at the trip description. He is done with the first step and is on the next. Should the result screen update and remove the first step - there is no need for it to still be there. I did not do this because it would be confusing for the user that the screen is different than before, so this was an easy decision to make.

While considering this, I found a more important situation to consider. While being on the trip the user takes the bus but doesn't manage to get off in time.

The functions of the application as it is now have no way of checking this. But updating the route would make it easier. A good addition to the application would be to let the user know that he didn't get off and then calculate another route for him. Also if the user gets on the wrong bus, the application should alert the user.
I have considered ways of doing this - using the GPS and the JourneyDetail element it would be possible to follow the ride and perceive the first time the route is not followed. Checking the start routeIdx and following the increasing routeIdx's combined with the GPS position would work. This addition would not stop the user from taking the wrong bus but it would help him get on the right track.

As discussed in chapter 6, Test, updating the search results could also allow the setting of showing the time of a departure as minutes untill departure.
Updating the trip would be a great help this way and would be a good addition to TravelBuddy.

7.4 Security

When I first started off with the project and heard the project requirements, I was in doubt if I wanted to take it on.
My main doubt lies in the security of this kind of program. The main idea was that TravelBuddy should follow the user from A to B. The question was, could this be done in a secure way, so a visually impaired could use it?
The plan was that TravelBuddy should have its own navigation system, which I doubted could be used because of the required precision - what happens if the GPS signal is low? Therefore I have put this requirement aside and focused on the public transport information which was the main requirement to the application.

I ended up with adding the possibility of getting direction by using Google Navigation as the navigation system, which is probably the most precise there is - but is this good enough?

I chose not to try to implement something better than Google Navigation - why try to improve something that is already working well. My client also said that he has used it before and liked the help there is provided. Therefore I chose to use Google Navigation - it could be interesting to develop this function on my own, but the focus of this project is to investigate and explore the possibilities of developing such an application and Google Navigation has the function that was needed.

Being visually impaired, it is very dangerous to move in the traffic. If any kind of navigation is used, the user can never fully trust that the directions are totally precise - but also needs to use his sense, intuition and ofcourse what he sees. This is not that simple for a visually impaired. Therefore, TravelBuddy is just a indicative help for moving in traffic - the user would still need to have awareness and be careful.

7.5 Applications for both Target Groups

My assignment was to implement a traveling application for the visually impaired. But what if we could develop applications for both the visually impaired and the regular users? What would be needed then?
I can imagine that TravelBuddy or another traveling application could be developed for both kind of users. TravelBuddy's purpose was to be accessible in order for the blind and the visually imapired to use it - but as I stated before, there are still labels and text fields so a not visually impaired could easilly use it. For the graphics, fancier things could be added, which a blind person wouldn't know that they existed.

I stated that the idea behind the design of the layout is to fill the whole screen and strech the views so the user wouldn't have to search in a long time for the views. A regular user would have to live with this as the design, although these choices wouldn't be the same when designing applications for regular users.

The only big and important difference there is, is that some of the options are simply removed when programming for the visually impaired. In the case of the function "Next Departure", I have removed the options like choosing what stop the user is interested in, letting the application choose the first and nearest stop to the given coordinates, instead of presenting the best 5 results as suggestion first.

When I presented this idea to the client, he would rather have the first and the best instead of having to spend time on finding the needed one. Also if you are using this function while you are at the stop, the departures for that stop would be the output - which is exactly what the client asked for.

If the same function was designed for a regular user, the option would probably not be removed. It isn't that time consuming to chose the desired stop yourself - and it could spare you the time if the application found another stop than the one you were interested in.

So this could be a setting that could be changed to the way the user likes it - maybe another visually impaired than the client would like the possibility of having this option and a not visually impaired who needs the information fast wants to turn it off.
So making it a setting could solve the problem.

Working with this kind of developing, I believe that it is better to develop two different versions of an application instead of trying to satisfy both parts in one application. There is no need to not fully satisfy both kind of users and thereby simply develop two different user interfaces. It is more considerate for both parts - the visually impaired would have an application with not to many choices and thereby an application which is easier to use, while the design for the regular user would be up to the developer and could be smart in other ways than found during this project.

However if one would like to develop an application for both target groups, it would be more considerate to start by developing the application for the visually impaired and then add options so a regular user also could get use of the application.
I reckon that this would be easier than the other way arround because it is easier to add options than removing them. To program for the visually impaired is troublesome as there are more special cases to consider than when programming for ordinary users.

7.6 Developing other Applications for the Visually Impaired

After my expirience with developing for the visually impaired, I evaluate that it is not that easy to develop any kind of application for this target group. It is limited what you can do and requires a lot of planning and consideration. Not that regular applications do not need this, but in this case it is very essenential that the layout is thought through.

My opinion is that applications for any kind of information could be developed for the visually impaired, like for instance for travel purposes, schedule, wheather forecasts or similar.

The limitations of the layout design make it difficult to implement games and other entertainment applications which already exist for regular users. An option is to research what kind of games there already are for the visually impaired and evaluate if they could be transfered to a smartphone.

One thing that I think would make programming for this target group easier is to have some kind of template that can be followed. For instance, there only be content descriptions on views which are either input or output or clickable - the way that I have done it in TravelBuddy - there is no need for labels and similar because the user hasn't any use of them.

It would be easier for the users to know how to navigate in different applications and also make it easier for the developers to have one way of doing this.

I believe that this field needs more research and can be improved by making applications for the visually impaired in some other ways to make them more accessible - for instance with speech input and output. This would it make much easier for the user to use the application, especially when the voice recognition is a continuosly improving field. There is still the discussion if the user would like to actually speak to his smartphone while in public, but that is also a problem that needs to be considered.

CHAPTER 8

Conclusion

In this project I have developed an Android application. The application has the visually impaired as the target group and its purpose is to make it easier to find information about the public transport and planning a trip.

During the project I have researched how to optimize the design in order to make the application useful for the visually impaired. To make the application more accessible, the transition from one activity to another has to be logical - this means there cannot be any smart precalculated information that the user hasn't searched for yet, this would be confusing.
Besides this, the layout is very important - the placement of views and their size is essential for making the application accessible. Big views that stretch on the whole screen are much easier to find than small views in a side or middle of the screen.

Another very important thing is not to have too much information on a screen or for each screen - it can only be read by TalkBack and if there is too much, the user can not just hear the information he needs, he has to hear it all again.

In order to provide information about public transport I used Rejseplanens API. There are different API calls for Rejseplanens API services which return different XML files. I have implemented XML parsers in order to achieve the user requested information.

The application, TravelBuddy, lets the user find the next departures from the nearest stop and also offers navigation to the stop, provided by Google Navigation.

Another function is that the user can plan a trip, where the only mandatory information that needs to be provided is the destination. The user will recieve a list of trips as a result and can choose one of them - and get a trip description. Here he can get navigation to the various stops of the trip and also use the application to get warned when he needs to get of the vehicle.

In order to make the application accessible, I wanted to make less mandatory fields that the user had to fill - therefore, the GPS plays a large role in this application. Using the GPS, the application uses more battery and is something the user needs to be aware of.

Programming for the visually impaired has its difficulties when wanting to implement many functions - it is definitly easier to program for regular users, because many of the options can be represented with icons or similar, which is unfortunatly impossible when programming for the visually impaired. Therefore I believe that it is difficult to develop an application which is optimal for both users.

The client felt that TravelBuddy is easier to use than regular applications, because his needs were prioritized. Based on this I believe that Accessibility and TalkBack can be used in order to make applications useful for this target group - the developing would require a lot of guidence and directions from the user.

However, I evaluate this field still to be new, the Accessibility function is fairly new and therefore the developers haven't yet found a common way of implementing things. But I believe that there is a future in this kind of applications - perhaps a better and easier technology than Accessibility and TalkBalk will be available soon.

Chapter 9

User Guide

9.1 Getting started

In order to use TravelBuddy, it needs to be installed first.

1. Copy the TravelBuddy.apk file onto your smartphone.
2. Find the file manually with any kind of file manager.
3. Click on the file and it will start the installation.
4. TravelBuddy is ready to use.

9.2 How to use TravelBuddy

9.2.1 Main Menu and Start

1. Find TravelBuddy in your application list and click to start.

2. If the GPS is not turned on, a popup will ask for permission. It is located in the middle of the screen. Press ”Yes” to change the GPS permissions. Press back to go back to TravelBuddy.

3. The main screen is shown with to options.
   The first option is an option for finding the nearest stop to your or a specified location and find information about the departures from here. The second option is to find a way to get from a point to another, including a trip description of what transport is needed, a navigation from and to the stop and getting of the vehicle.

9.2.2 Find the Nearest Stop

1. In order to find the departures from the nearest stop find a button in the middle of the main screen which reads *Click to find departures from nearest stop.*

2. If a search with default time, date and GPS location is desired, simply press the search button, which is located in the bottom of the screen and reads *Search.*

3. If another location is desired than the GPS location, it is specified in the first field, at the top of the screen. The field reads *Type your location.*

4. If another date than the default is desired, specify it in the second field, in the middle of the screen. The field reads *Input day, month, year.*

5. If another time than the default is desired, specify it in the third field, in the middle of the screen. The field reads *Input hour and minute.*

6. When the information is provided, press the Search button.

7. A progress dialog appears, reading *Progress. Loading data.* Wait until the next screen appears.

8. If there are no results to the search, an alert dialog will show and read *No departures were found.* Press the ”Ok” button which is located in the middle of the screen. This will lead back to the seach screen, where another location, time or date can be specified.

9. If there are results to the search, they will appear as a list. In the top of the screen, the name of the found stop is shown and is read if invoked. Start exploring the results from the top and get information about the departure time, direction and vehicle name.

10. Click on a departure in order to get a navigation to the stop. The navigation is provided by Google Maps and navigates from your current GPS position to the stop.

11. Click back in order to get back to the previous screen.

9.2.3 Trip from A to B

1. In order to find a trip from A to B find a button in the bottom of the main screen which reads *Click to find trip from A to B.*

2. If another location is desired than the GPS location, it is specified in the first field, at the top of the screen. The field reads *Type your location.*

3. Provide your destination in the next field, which reads *Type your destination*

4. If another date than the default is desired, specify it in the second field, in the middle of the screen. The field reads *Input day, month, year.*

5. If another time than the default is desired, specify it in the third field, in the middle of the screen. The field reads *Input hour and minute.*

6. If any transport vehicle is to be excluded from the search, it can be done by deselecting it by clicking on the boxes which read *don't use Bus*, *don't use Train* or *don't use Metro.*

7. When the information is provided, press the Search button, which is located in the bottom of the screen.

8. If a destination isn't provided, an alert box will show, reading *Please provide a destination in order to search.* Click "Ok" on the alertbox which is located in the middle of the screen to go back to the search screen. Provide the destination and click Search again.

9. A progress dialog appears, reading *Progress. Loading data.* Wait until the next screen appears.

10. If there are no results to the search, an alert dialog will show and read *No departures were found.* Press the "Ok" button which is located in the middle of the screen. This will lead you back to the seach screen, where you can specify another location, destination, time, date or vehicle.

11. If there are results to the search, they will appear as a list. Start exploring the results from the top and getting information about the departure. The information provided here is the start stop and end stop, the arrival time and the number of shifts of the trip.

12. Click on a trip in order to get additional information. A new screen will appear, showing the steps of the search. The steps are shown as a list and should be read from top to bottom.

13. Click on the different steps in order to get navigation. If the step is a walk to a stop, navigation is provided by Google Maps from your location to the stop. If a step is by transportation, a new screen will appear when clicked, which reads *Waiting to reach the stop* if invoked. When there are less then 200meters to the stop, an alert box will alert you that you need to get of the vehicle, by automatically reading *Destination is reached within 200 meters.*

14. Click on "Ok" button in order to get back to the previous screen.

OBS! It is assumed that the Accessibility option TalkBack is turned on while using TravelBuddy.

Bibliography

[1] Android Developer *UI/Application Exerciser Monkey.* http://developer.android.com/tools/help/monkey.html

[2] Android Developer *Android Developer page used for in chapter 3, Theory and for implementation.* http://developer.android.com/index.html

[3] Lars Vogel *Android Development Tutorial.* http://www.vogella.com/articles/Android/article.html Version 10.4, 2012.

[4] Lars Vogel *Android Tutorials.* http://www.vogella.com/android.html Version 10.4, 2012.

[5] Geoffrey Goetz *iOS and Android Development Compared.* http://gigaom.com/apple/mac-dev-notes-ios-and-android-development-compared/l Version 10.4, 2011.

[6] Lars Vogel *Android Threads, Handlers and Asynctask Tutorial.* http://www.vogella.com/articles/AndroidPerformance/article.html Version 1.9, 2012.

[7] *Android Accessibility.* http://eyes-free.googlecode.com/svn/trunk/documentation/android_access/index.html

[8] *Android Developers - Accessibility.* http://developer.android.com/guide/topics/ui/accessibility/index.html

[9] *iOS Accessibility.* https://developer.apple.com/technologies/ios/accessibility.htm