Native Visualization of Mobile Activity Patterns

Bachelor Thesis in Computer Science

Christian Janßen
302530

Advisor:: Prof. Ulrik Schroeder, RWTH Aachen
Second Advisor:: Prof. Jan Borchers, RWTH Aachen

Supervisor: Dipl.-Inform. Hendrik Thüs

Christian Janßen
Learning Technologies Research Group
LuFG Informatik 9
RWTH Aachen University
September 1st, 2013

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed.

Aachen, September 2013 Christian Janßen

Contents

Part I

Chapter 1 Introduction

Smartphones and their daily usages

Since the introduction of the smartphone and its ongoing boom in sales, those devices getting more and more important for their users. Nearly 30% of the German population own a smartphone and 50% of them use it on a daily basis [8]. Googling a question, making a phone call, sharing your location and status with your friends via facebook or just taking a snapshot of your lunch - smartphones are used in nearly every location and situation. With all these possibilities and potential in usage for everyday situations, it is getting harder and harder to keep track of when one has used its smarthphone, where it was used and most important, for what was it used. Knowing this may have a positive influence in productivity. While on the topic of productivity, another interesting fact is, that 72% of owners state that they use their smartphones at work [8]. The obvious question is, did one use its smartphone for relevant research or emailing, or was it used to chat on whatsapp or checking facebook. It would be desirable with focus on efficiency and productivity if by the end of the day one could check his or her smartphone activity and would see that one may have to improve his or her working or learning behavior, because the smartphone was used in a distracting way or, the more preferable case, that the smartphone was mostly used to get work done.

Need of a structured overview

Another situation in which nine out of ten users utilize their device is while they are on the move [8]. Again, it would be desirable if one had information how he or she got from place A to place B, how much time it took and what has been done in this time.
Not only has the computation power of smartphones significantly improved, their integrated cameras are also quite satisfying and are comparable to low-end digital cameras. Since most people tend to take pictures with their smartphone nowadays, it

would be helpful to get a simple overview in which one can easily see the exact position of the picture's location without taking any further investigation.

The lacking of the possibility to check what has been done with the smartphone may unconsciously lead to a distracting use of it. That is, because, as already mentioned, keeping track of all activities is extremely difficult and thus the total time of the daily usages is normally unknown. There exists a need of a structured overview which provides informations about time, location an d applications to get a self reflecting impression about one's daily mobile activities.

1.1. Objectives

Requirements for the application

Due to the stated need of an application which provides information about one's daily activities the main goal of this Bachelor thesis is to create such an application. Requirements for applications are normally high and the developer has the task to find optimal solutions for them. This application is no exception and thus has its own requirements. Those are on the one hand, that it should display enough information to grant the ability to draw conclusions about one's daily activities but on the other hand it should also display the data in a visual appealing and intuitive way such that the visual appearance is not too crowded with unnecessary information.
The application should also offer the possibility to take minor adjustments to fit in one's individual needs and thus making the experience with the application personalizable. In contrast, those adjustments should not be to detailed and low in number to keep the options structured and understandable.

Applications for partial solutions

As mentioned, the main objective of this thesis is to create an application which provides feedback and a self reflecting view for one's daily activities. There are some existent applications that partially fit in the described situation and offer solutions. For example one can use Google Latitude to see where he or she has been traveled respectively which routes were taken to reach a specific place, but it is not capable of showing which applications have been used at a specific place. Also, while this thesis was written the support for Google Latitude was discontinued [7]. Another partial solution is facebook. One can share his or her location and photos on facebook but this will not show a route on a map nor does every one want to public his or her whole life on facebook.

To see the most used applications of the day one could take a look at the operating systems utilization chart which displays the percentage of used battery life and shows the cpu usages in total time. Not only does it not take into account the visited location this is also a very impractical and non intuitive way of gathering information about one's daily activities.

Testing of graphical possibilities

Another objective was to create an application which provides manifold views of the user's daily activities. It should not be limited to a map displaying pins or a list of application names with the total time of usages. Instead the application should show what is possible to develop with the help of already existent graphical views and, what is even more interesting, with creation of new views. All views should differ from each other and show different possibilities of visualizing informations of one's daily activities while each one could be used as a stand alone self reflecting view.

The application of this Bachelor thesis should perform as a central information provider which combines the listed objectives and displays the daily activities of the smartphone. Those information will be presented in various ways and can be filtered by different criteria within a clear, intuitive graphical user interface.

Chapter 2 Background

This chapter provides background information about the main topic of this bachelor thesis. First, the ongoing trend of using a smartphone in nearly every situation and therefore the need of keeping track of one's own mobile activities is discussed. To grant an application the possibility to give self reflecting impressions the application itself needs to be provided with personal data of the user. The second section is about this provided data, its origin and how it is gathered. Once the origin of the information is discussed the next section talks about the representation of the data. In this context the meaning of native visualization is explained and an alternative solution is presented which involves a short introduction of a currently written Master thesis. The last section lists and explains which hardware and software was used during the implementation.

2.1. Self reflection

Smartphones as modern multitools

A smartphone has many usabilities. It can be used as a camera, newspaper, music player or as a portable gaming device and those are just a few examples. There are a lot more ways to use a smartphone and as mentioned in the introduction, they are used in nearly every situation. It is a modern multitool which was used by more than 23 million people in Germany in 2012 [8].

Smartphones as helpers

As advantageous as it may be in everyday situations, the downside is that most people do not know how much time they spend on their phone and thus do not know how distracting it may be. For example, checking new mails may lead the user to also check the newest facebook messages and stay within this application a few minutes longer than expected. At least 50% of smartphone owners access the Internet with their device [8], thus most people are always available through instant messaging services like whatsapp. The result is that people write and receive messages

more often. And because smartphones are capable of running diverting games, one may uses its device to beat the last achieved high-score.

Distraction

But a smartphone can be a great helper too. It is an easy to use digital calendar which reminds the user of all upcoming events, it can be used as a travel guide or a navigation system, it allows to quickly respond to an important email and has many more useful advantages. But the previous short examples demonstrate that a smartphone can also have a distracting influence to its owner.

At this point the the idea of self reflection is needed. The concept of self reflection is the critical reflection on one's own actions and positions and coming to a conclusion. This can be used to assess the distracting influence of smartphones to its owner. But for an accurate assessment one needs to keep track of his or hers own daily activities, which is nearly impossible to do for a smartphone without the help of tools that provide background information.

Provide a self reflecting view

As mentioned, the help of a tool is needed which provides information about the owner's mobile day in a self reflecting manner. This tool in form of a smartphone application should display the information such that the user is able to instantly see where, when and for what the device was used. If one is provided with this data, he or she is able to draw conclusion, whether they used their smartphone in a productive manner or if they used it for entertainment. Further more, one could tell if he or she used the smartphone to divert themselves from working or studying. Although the application can display the needed information, the conclusion must been drawn by the user.

Possible improvements in productivity

With this application and the provided information a user may be willing to rethink his or her work respectively study behavior. This could lead to a less frequent use in distracting applications, thus improving efficiency and productivity in daily tasks.

As described, there is a possible usage area for such a smartphone application. It should visualize data and information about the owner's daily activities, such that the application could be used as a self reflection tool.

2.2. Provided Data

In the last section the idea of an application which displays information in a way such that one can use it for self reflection was described. What has not been described is the source of the data to be visualized and how it is gathered.

SmartDay and Big Brother

This thesis' application, namely *SmartDay*, will not gather the data it uses, instead the data is downloaded from a server and stored internally. The mentioned data arises from an external application called *BigBrother*. This application is based on the Master thesis of Thorsten Kammer [13] and was reimplemented and developed by Hendrik Thüs in 2013 [3].
The data Big Brother gathers, is sent to a server where it can later be downloaded in an aggregated way and used by the application developed for this bachelor thesis. The data contains amongst other, information about the user's visited locations, the name of the currently used application, start and end time of used applications. This data is uploaded and stored on a web server, which is then accessed by this thesis' application.

Privacy issues

With the revelations published by Edward Snowden in June 2013 about the U.S. American spy program PRISM one might be concerned about privacy violation by third parties. It should just be said, that this project is still in an experimental phase. If it should be published for a larger audience than the developers, much work would be put into encryption and ensuring the prevention of unauthorized access by third parties. Another solution would be the release as an open source project. In this case users would be provided with the needed applications and programs to host such a centralized service on their own.

Reasons for online stored data

One of the reasons why the data of daily activities is stored online, is the limited storage of mobile devices. By uploading the data, the used size of storage can be minimized and only needed information would be downloaded. The possibility to merge data from other devices the user owns, like PCs, laptops and tablets is another reason for uploading the data. Being able to access the data from multiple devices like tablets is also an important point. This method is of course at the cost of data traffic but is needed to centralize data especially in case of merging data and accessing them from different devices.

The idea of a data set which also contains information about other used devices has great potential in granting an even better overview of one's daily activities and would make the self reflecting view provided by the application even more meaningful. Having the data of the activities of a laptop could be powerful for a student who uses it as a learning tool, because he or she may not use the smartphone during that time, but instead one could see if he or she was productive, by checking if the laptop was actual used for studying by reading a pdf file or if it was used to browse non important things on the Internet. With additional data available from laptops or PCs, the coverage of one's day would be more complete.

2.3. Native Visualization

Now that the origin of the provided data has been described, the idea of a native visualization will be explained in more detail.

What does visualization mean?

The term of *visualization* means the representation of abstract data or information, like a text, in a visual ascertainable form. Its meaning is not limited to computer science. It can be found in various situations and places, technically anywhere where someone tries to convey information in a visual way. This may be a picture of an artist or a even a movie. The concept is not even limited to modern time. Since the early days of mankind, humans try to express information in form of visual tangible objects. For example the Egyptians did this 2000 years before Christ, by creating pectorals which for instance display a gryphon standing on a kneeling man of different skin color to express the Egyptians position in foreign countries [10].

As mentioned, in computer science, visualization means the representation of data in an illustrative way. For example, creating a pie chart for results of a survey or drawing a graph representing the daily temperatures for a week. For this thesis the visualization has to fulfill the task of displaying one's daily activities in an appealing and easy to understand way, such that one can directly draw conclusion from the information.

Native visualization

Native visualization describes the creating of visual ascertainable objects only by means of resources which a specific system provides without any addition. A visualization would be the generation of a website with with the use of JavaScript and then displaying this website in the application. But this would be non native because a website would be used to create the view.

In this thesis, the application will be implemented for the mobile operating system Android 4.0.3 and higher. Native visualization under Android means the use of Java and the access to the standard Android application programming interface. The application will not use JavaScript or any other non native help, as this would infringe the terms of native visualization.

A non native visualization of daily activities

An alternative to native visualization of activities can be found in Thomas Honné's Master thesis "Interactive Visualizations of Activity Patterns in Learning Environments" [11]. The thesis describes, among other topics, the visualization of daily activities in a web browser environment. For comparison an interesting fact is, that the data arises from the same origin. For more information please refer to chapter 5.

An advantage over the non native method is that data can easily be stored locally, thus making the application available for offline usage assuming that the needed data has been downloaded at some point in the past.
With a native application the user has a tool for self reflection which is not permanently bound to an Internet connection and does not require any additional non native resources.

2.4. Hardware and Software

As mentioned in the previous section the thesis' application will be developed for Android 4.0.3 and higher versions. To get an impression of the used tools in the development process this section describes the used software, the development environment and Android, as well as the used hardware, the development device.

The operating system of choice

The application was developed for Android 4.0.3 "Ice Cream Sandwich" with the application programming interface level 15. To minimize overhead due to compatibility adjustments the support is only guaranteed for Android 4.0.3 and newer versions, which account for 63% of all Android devices [6].
Android was the operating system of choice because it has a free developer license, great supportive community and with a share of 76.7% in quart two of 2013 [2], Android is the largest market share holder of mobile operating systems.

The test device

The testing device was the Motorola tablet Xoom with Android 4.0.3. The tablet with a screen size of 10.1 inches and resolution of 1280 times 800 pixel gave great advantage in the testing process of the application. Its screen is large enough to display all

relevant data without minimizing their visual appearance. In addition the screen size allows precise testing of multi-touch gestures. The alternative to the real development device would have been an Android emulator provided by Google. With the emulator the development of this application would have been nearly impossible due to the fact that it does not support the ability to simulate multi-touch input with a normal mouse and does not allow support for GoogleMaps.

The development environment

Google's recommended software development kit Eclipse with Android Development Tools served as development environment. Working and developing with Eclipse was simple and comfortable due to Google's numerous tutorials [5]. Neither installing the software development kit on a Windows 7 computer nor setting up new projects was a problem. The test device connected to the computer was directly recognized by eclipse and executing and testing written code on it proceeded without problems. Eclipse's debug mode was also very helpful at many points in the development process and helped to track down hidden bugs.

Working with Eclipse and Android was comfortable, a lot easier than expected and straightforward, even for a first time developer. Eclipse's improvement proposals and performance advices and Google's tutorials with helpful examples and background informations made this project a great educational experience.

Part II

Chapter 3 Development and Implementation

"It's done, when it's done"

—An English Phrase

3.1. Outline

The sections in this chapter will describe the various stages of development and implementation of this Bachelor thesis' application. At first the basic idea of the layout and the individual views will be described and explained. To get an even better impression the early paper prototype will be shown.

The implementation of the basic layout which was displayed and explained with the help of the paper prototype, will be the first part describing the actual implementation process. Second, the handling of the data to be visualized is explained. This covers the server based access as well as local storing and loading.

After the preparations for the visualization have been set forth, the different views will be described. Starting with the explanation of the map view, it will be clarified how the loaded data is visualized. This is followed by the section focusing on the chart view. It explains how a library is used to draw pie charts and how the loaded data has to be prepared. The last section describes how the timeline was implemented. It covers, how one can draw its own views in Android and how to make use of gestures.

3.2. Paper Prototype

The first step of the development process was the creation of a paper prototype. It was needed to demonstrate the idea and visual appearance of the project and application. An advantage of the paper prototype is the ability to show the application to different people to get feedback without even writing one line of code. This provides the ability to eliminate possible false estimations in the forefront of the application's implementation. False estimations may be the assumption of wrong needs of possible future users or the creation of an non intuitive layout. In the following the paper prototype will be shown and explained.

Basic layout

The first idea was to split the screen into two parts, an option part and a view part. The left part should be the option part, occupying one sixth of the visible screen. It should always display the following options ordered by their occurrence.
In the top left corner a view selection containing three buttons ordered vertically with titles "Map-View", "Chart-View" and "Timeline" is displayed. Taping one of those buttons causes a switch to display the respective view.
Under the view selection a list of options should be visible. Those options vary from view to view but should always contain a button at the top of the list displaying the currently selected date. Tapping on this button causes a date-selection-window to pop up. Selecting a date leads to an update of the displayed view, now visualizing the respective data. Other options will be explained together with their respective views.

Map-View

When the application has been loaded, the user will be displayed the map view thus making it the application's start view. The view will display a map with a route, based on the user's visited locations. On start up the selected date will be the current day and therefor the drawn route represents the user's latest movements. Tapping on the route should bring up a speech bubble which contains information about the tapped location. Those information will be the time spend at the specific location, the used applications and possibly shot photos.

Map-View specific options

The view provides two special options. The first option is represented by a check box and has the title "Photos" and provides the option provides the ability to highlight those locations, where a photo was taken. The second option is called "Apps". It lets the user pick applications from a list and colors all parts of the route where those specified applications were used. Those options would provide the ability to quickly access an overview of position data of specific applications and actions. An actual

image of the paper prototype representing the map view can be found in figure 3.1.

The map view can mainly be used to answer the question "Where was the smartphone used?", since the view focuses on visualizing the map and traveled routes rather than displaying numbers and percentages.

Figure 3.1.: Map-View

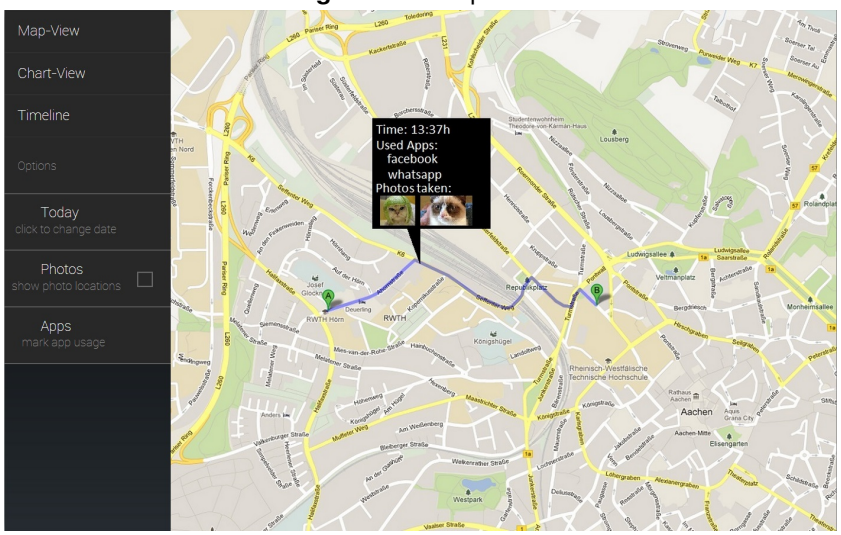


Chart-View

The second view, the chart view, shows the user his or hers daily activities in form of a pie chart. Its layout can be found in figure 3.2. The idea was to create location based charts which show for each visited place the percentages of used applications. To break down the number of shown charts and thus to give a better overview, locations would be summarized in an intuitive way. That means that one has a chart for work, home and on the move. Those charts are lined up vertically and have an *individual* chart on the top of the list. This individual chart can be adjusted by the two extra options described in the following.

Chart-View specific options

The first option lets one choose which places are taken into account for the *individual* chart and the second option determines which specific applications are displayed in the this chart. With these two options, one has the ability to customize the first chart to visualize only those data which fit one's individual needs.

Grouping of apps and locations

To get an even better overview, applications are classified into different groups. "Social" and "Productive" could be two groups, which split the applications. Applications like whatsapp and

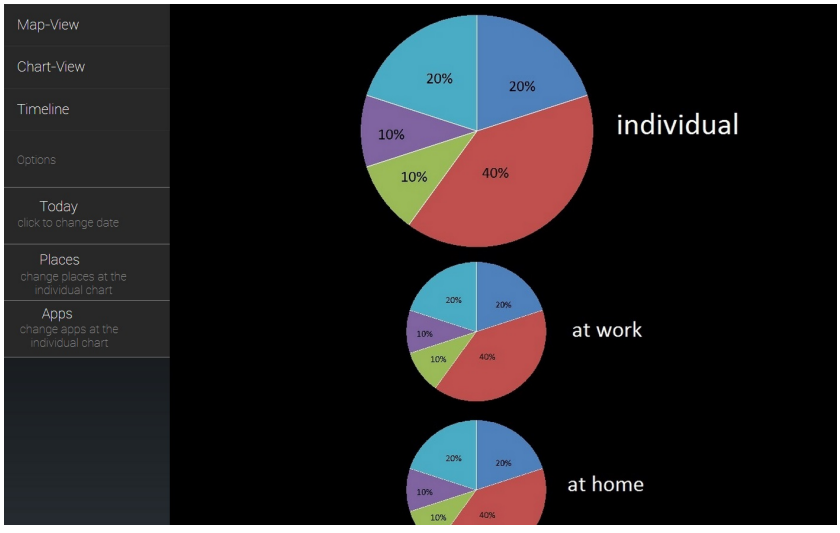
facebook may be social, while mail programs would be productive. The user would also be able to classify the applications by him- or herself, allowing to personalize the groups and thus personalize the application. Furthermore, home, work and other places are chosen individually and extra places like parent's home can be added by the user, to give him or her more space for customization.

Interacting with the view

To get an better insight of the application groups, tapping on a slice of the pie chart will bring up a detailed view. In this detail view, one can see which applications are included in the tapped group and how large their percentage of daily activities is. In addition the total usage time of each application is shown, to give the displayed percentages more expressiveness.

The chart view, in contrast to the map view, is able to give a better overview of the percentages of used applications rather than showing at which locations they have been utilized. It gives answer to the question "What was the smartphone used for?"

Figure 3.2.: Chart-View



Timeline

The timeline is the third and last view of this thesis' application. For a visual representation of the description of this view, please refer to figure 3.3. This view visualizes the daily activities in a chronological manner with two major parts. The first part is the the timeline itself and is described as follows. At the layout's bottom a horizontal line is draw with markings for every hour, from 0:00 to 24:00. Above this line, colored rectangles are drawn which represent a timespan in which an application was used.

At the layout's top one can see markings for visited locations in dependence of the displayed timespan. One has the possibility to scroll horizontally through the view to observe the consecutively occurred activities.

The detailed view

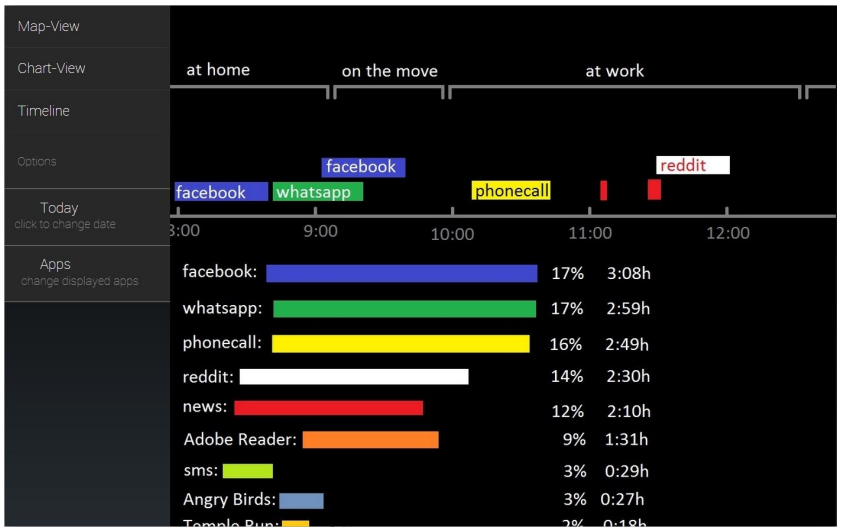
Beneath the timeline a detailed view of all applications used on the selected date is found. This second part shows the applications in descending order starting with the most used application of the chosen date. For each application a rectangle which represents the percentage of daily usage, is drawn. This rectangle will have the same color as the respective rectangles in the timeline and thus the detailed view can be used as a legend to identify the applications displayed in the timeline. Next to the application's bar the respective percentage and total amount of time is displayed.

Timeline specific options

This view offers the extra option "Apps", where one can filter the displayed applications. With that option one is able to hide all other applications and display only those which are relevant for the user and thus giving the application a feel of personality and individuality.

Although it partially acts as a bar chart, the timeline, unlike the other views, focuses on the daily schedule by visualize the activities in a chronological order. With this view the user is able to answers the question "When was the smartphone used?".

Figure 3.3.: Timeline



As mentioned in the first place, this was the basic idea of the application and a few things have been changed during the development process. A summary of changes which differ from the paper prototype, along with the reasons for those changes can be found in the next sections.

3.3. Basic Layout

The actual implementation work started with the creation of the basic layout. It characterizes the general appearance of the application and has changed during the process of implementation.

View, Viewgroups and Fragments

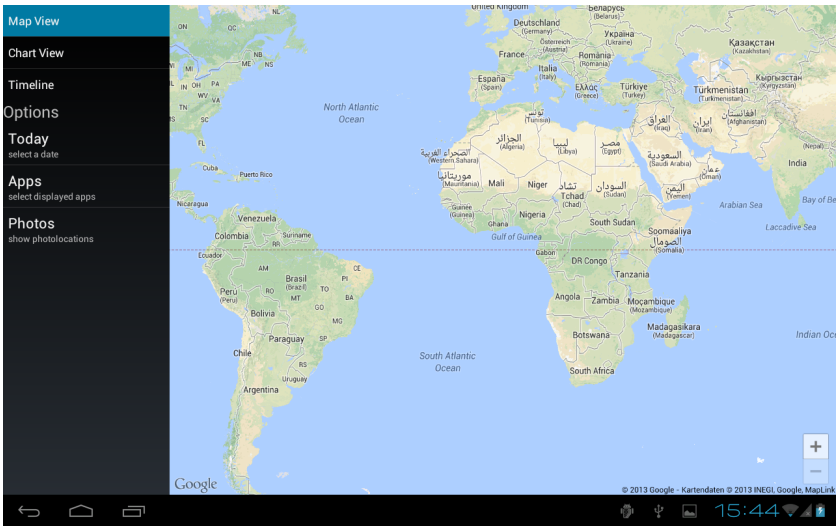
In order to display anything, one first has to pass a layout to the main process, the main activity, to create a visible view. Those layouts are defined in xml and contain views and viewgroups. Basically, viewgroups contain views and/or other viewgroups and define their layout. For example one can define, whether visible views are ordered horizontally or vertically. Views contain the actual visible content, for instance a text or an image. Views are not static objects and can change during run time. One is also able to create and delete views and viewgroups during runtime, which will be helpful later. In order to create an application which looks and works like the paper prototype, one has to be able to change or switch whole groups of views in order to switch between map view, chart view and timeline. For this purpose one can use fragments. Fragments are kind of sub-activities which handle their own lifecycle and viewgroups and they are necessary for an interactive application, as they can be reused during runtime, thus provide an efficient way of switching views.

The first basic layout can be found in figure 3.4. It resembles the paper prototype in functionality and appearance. Tapping on the view's name causes a view switch and the options bring up pop up menus to select applications or a date. To provide the application with the ability to react to user input, one can assign click listeners to views.
A listener is an interface which defines the name and the parameters of a function which can be implemented by another class. This allows the interface providing class to communicate with the implementing classes by casting those classes to a listener instance and calling the respective functions.
After creating and assigning a click listener, Android calls it as soon as the user taps on an assigned view and a specified action, for instance a view switch, will be performed.

Visible fragments

As mentioned above, at the start of the application the main activity needs a layout in order to create visible content. In this case the layout consists of three fragments. The first fragment is the view selection fragment. This is a list fragment which contains the three section names as views. The list fragment provides the programmer with the function `onListItemClick` which he or she can override. It gets called if one of the views names get

Figure 3.4.: First Basic Layout



clicked and is provided, among others, with the list position of the clicked view. With this information one is able to determine which view is requested and can then call the main activity to initiate a switch.
The second second fragment is the options fragment which is also a list fragment with the same ability to react to user input as known from the view selection fragment. If the user taps on an option the respective menu pops up. If the user makes adjustments in options, the underlying dataset is updated and the main activity is called to update the currently visible fragments.
The last fragment and most important in case of visualization and self reflection is the view section fragment. Those fragments visualize their own layouts with different views which are described in their respective section.

Add and switch fragments

In the first layout, the view selection fragment called the main activity in order to switch the view section fragment. To switch fragments the main activity uses the `FragmentManager` api. To obtain a `FragmentManager` object, the main activity calls `getFragmentManager()`. With this object one can access a `FragmentTransaction` object, which is used to add or replace fragments. With the `FragmentTransaction`'s function `add(int, Fragment, String)` one adds a fragment to a specific view. To do so, one has to specify the container in which the fragment will be placed via an integer id previously assigned. The second parameter is the fragment which should be added to the container and last one is an optional tag which is assigned to the fragment in order to be able to access it later. With a call of

FragmentTransaction's function commit() the transaction will be executed.
To switch a fragment, one calls the FragmentTransaction 's function replace(int, Fragment, String) with the same variables as previously described. It should be mentioned that only those fragments can be replaced that were added in a programmatically way and were not defined in the xml's layout file. The function replace should be used, because it removes the old fragment and then adds the new fragment to the container. If one would use the function add, it can not be guaranteed that only one fragment is visible, because fragments which display a transparent background would show the underlying, not removed fragment. But because replace() removes and destroys the occupying fragment, its a rather inefficient solution if a user switches views very often, due to the fact that fragments will be recreated every time they are added.
A solution to this problem are the FragmentTransaction's show() and hide() functions. These functions show and hide fragments without destroying them, therefore they are more efficient in case of battery power consumption and calculation time. Another advantage is the maintaining of zoom and scroll positions without adding a single line of code.

Dialogs and DialogFragments

To change the fragment's appearance, one can tap on the options on the left side to bring up a pop up menu with a calendar or a checklist of application names. These option menus are created with Android's Dialog objects and are controlled by the DialogFragment api. The first options menu, the calendar, is displayed by a special dialog, the DatePickerDialog which provides a visual appealing user interface to select a date. Once the option fragment created a DialogFragment, it has to assign itself as a listener to the object and call show() to display the graphical user interface. In order to be able to handle the choice of a date, the DialogFragment implements DatePickerDialog's OnDateSetListener, which gets called with the selected year, month and day if the user approves his or hers chosen date. Afterwards the DialogFragment calls the previous assigned listener and passes the the date to it. The options fragment then displays the selected date and stores it in the dataset, which notifies the main activity to update all visible fragments.
The other pop up options menu is also Dialog managed by a DialogFragment. It displays application names with a check box. To show this dialog, the options fragment creates a DialogFragment object and calls show(). The options fragment does not have to assign itself as a listener but has to select a mode for the DialogFragment in order to load, display and store

data correctly. Those modes are `select_apps`, `ignore_apps` and `select_highlight_apps`. Each mode builds its dialog identically with the help of a dialog builder. This builder offers different functions to create various dialogs and in this case is used to create a dialog with a multiple choice list and two buttons on the bottom. The builder provides a function called `setMultiChoiceItems()` with the following input variables. A string array of application names, an array of booleans representing checked states of the applications and a listener to handle click events. The loaded application names and the created boolean arrays are different for every mode, for example the `select_apps` mode only displays names of applications used at the currently selected date and forgets the selections after a restart, whereas the `ignore_apps` mode loads all application names ever used by the owner and stores the selections in the dataset. The assigned listener tracks selections and deselections of list items and stores the application's name and the respective boolean for every change. If the user confirms his or hers selection, the dialog stores according to the chosen mode the selections in the dataset which again tells the main activity to update the fragments.

Change of the basic layout's visual appearance

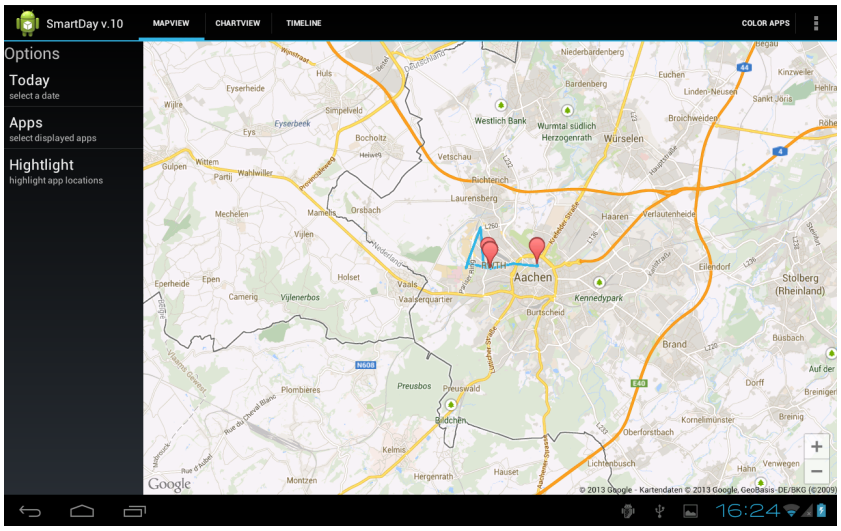
As the process of development progressed, the need of more options than previously assumed raised. Options for colorization of applications, switching the logged in user, deleting local files and permanently ignoring applications needed to be assigned to the layout. The problem with these options were, that they would not be used as much as selecting a date or temporarily hide displayed applications. Adding the options under the existing ones would cause a loss in simplicity and clarity of the application. The solution was Android's `ActionBar` api.

The `ActionBar` serves as a navigation bar which is set up at the top of the application and can be used by the programmer to add tabs and option buttons, where option buttons can also be combined in small pop up menu.

The action bar

In order to make full use of the now used `ActionBar`, the former view selection fragment was deleted and the views are now selected on the action bar by tapping on the new tab header. This layout is more intuitive as it resembles know patterns of Internet browsers and feels more structured. On the right side of the action bar, the color apps button is visible next to three dots which open a small list with the other new options - switch user, delete saved files and ignore apps. For more details see figure 3.5.

Figure 3.5.: Final Basic Layout



Action bar tab header

The main activity controls and sets up the bar as follows. By default, the action bar is visible in the application's layout but does not contain anything than the application's name in the top left corner. To add tab headers and options, one has to work with the `ActionBar` api. The main activity calls the function `getActionBar()`, which returns an `ActionBar` object. With this object one can call the `ActionBar`'s function `newTab()` to construct a new `Tab` object, which then has to be provided with a title via `setText(String)` and in order to react to taps, with a `TabListener`. In this thesis' application the `TabListener` class which implements the `ActionBar`'s `TabListener` interface and manages its own tab and the respective fragment. The reference to the `TabListeners` are saved in the main activity, such that it can programmatically select or reselect a specific tab to notify changes or just to switch the tab. Once the data is provided, one can add the tabs to the `ActionBar` with a call of the `ActionBar`'s function `addTab(Tab)`.

Action bar option buttons

Option buttons for the action bar are declared in an xml file. In this application the xml nodes representing the buttons contain four key value pairs. The first entry is an id, which is used to refer to the button from within the programs code. The name of the button is given in the field title. A priority number to define the order of appearance of the buttons and an option to define if the button should be grouped under the three dots list or if it should be directly accessible on the action bar.
The main activity then has to override the function `onOptionsItemSelcted()` which gets the tapped button as

input. With the previously assigned ids, one can differentiate between the four buttons and execute their respective functions which will be explained in the following.

Switch user

The "Switch User" button allows the user to switch the profile accessing the visualized data. If this button is tapped, the main activity looks up if the user had stored his or hers login information and deletes them. Then the underlying dataset function `createNewUser()` is called and the user is asked to provide new login data. The mentioned data set and its corresponding functions will be explained in detail in section 3.4.

Delete files

"Delete Files" gives the user the ability to regain occupied memory storage by deleting saved files. The main activity looks up the applications file directory and deletes any stored file, excluding the user login data. Those stored files are downloaded information about days along with colorings of displayed applications and the selection of ignored applications.

Ignore applications

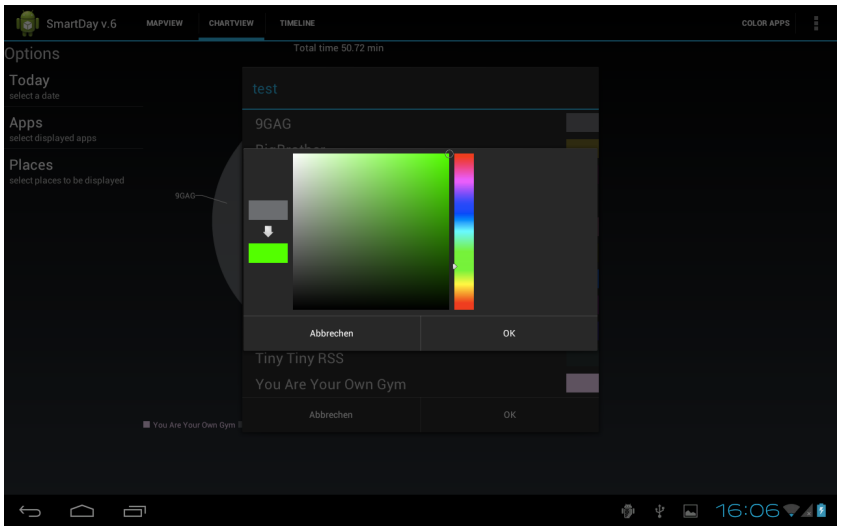
In order to permanently remove applications from selections and views, one can tap the "Ignore Apps" button. This causes the main application to request every used application from the dataset by calling `getAllApps()`. After the dataset has downloaded all application names from the server, the main activity creates a dialog where the user can pick applications which will be hidden permanently. The selection will then be stored in the dataset and all views will be updated.

Color apps

The button which is visible as long as the screen sizes offers enough space, is the "Color Apps" button. Tapping this button causes the main activity to create a dialog displaying application names with respectively colored rectangles next to them. A tap on the application name or rectangle unveils a new pop up menu which lets the user pick a color as seen in figure 3.6.

This new menu is built with Yuku Sugianto's Android Color Picker or the Indonesian translation Ambil Warna library project [14]. To use this color picker, one has to include the source code into its own application code and define it as a library project in order to access functions and classes. Once Ambil Warna was linked as a library project, the application uses it as follows. As soon as an application name or the respective rectangle is tapped, the `onClickListener` extracts the applications title from the provided view and creates an `OnAmbilWarnaListener` which overwrites the functions `onOk()` and `onCancel()`. These listeners are called if their respective button is tapped. If `onOk()` is called, the function stores the color and respective application name and afterwards updates the color of the rectangle displayed in the dialog.

Figure 3.6.: Color picker



The AmbilWarnaDialog

After the construction of the `OnAmbilWarnaListener`, the `AmbilWarnaDialog` will be created by passing the current activity, the start color and the listener. With a call of `show()` the dialog is displayed and the user can interact with it. Once the user has picked all colors and hits ok, the dialog stores the new colors in the data set which then notifies the main activity to update its fragments. If the user decides that he or she does not want to apply the changes and taps cancel, all changes are discarded.

The basic layout is an essential part of the application experience. It offers not only a structured layout for view corresponding and general options, it also presents a surrounding structure for views themselves. The possibility to seamlessly attach new views with tab headers and options on the left option fragment without destroying the uniform appearance of the application may be an advantage for possible future projects.
The basic layout was created to be minimal in terms of displayed options in order to prevent distraction from the actual function of this application, the self reflection.

3.4. Data Management

In the last section the term of a data set was mentioned a few times. This data set or the managing class of it will be explained in this section. It is the essential for the whole application, as it provides every fragment with the needed information about the daily activities. The construction of this class was one of the hardest part of developing SmartDay and pieces of it were rewritten or removed as the development of the different fragments proceeded. Especially the listener interface provided by the data set and implemented by the main activity has been adjusted a few times. This was necessary, because the demands on the provided data have change and extended during the development.

JSON format

The application's underlying dataset is stored in a *JavaScript Object Notation*, in short JSON, format. JSON structures consist of key:value pairs, where the value can be a number, string, object or array. Keys and strings are enclosed by quotation marks and the key is separated from the value by a colon. An object is enclosed by curly braces and contains a collection of key:value pairs, each separated with a comma. An array is a collection of objects, each separated by a comma and enclosed by square brackets. JSON offered an easy way to handle the data provided by the server and was also natively supported by Android. The following sample describes the actual structure of the JSON object which is primarily used by the application and can be seen in figure 3.7.

JSON object in SmartDay

In the top level object, the structure holds information about download and date time stamp, along with the total time of application usages. In this application, all information about time are given in second precision. The JSON object holds an array called *result* and contains a JSON object for every application used at the specific day. Each of these objects contain a field *app* with the application's name stored as a string and a field *duration*, containing the total duration time of the application. Each object also contains an array *usage*. This array contains objects of every timespan in which the application was used. Every *usage* object contains keys *start* and *end* with respective time stamps for start and end time. In addition the field *session* contains the applications session id and an array *location* is given which contains two objects. These objects contain a key:value pair *key* which identifies whether the object holds information about the longitude or latitude, by storing a string with content "lng" respectively "lat". The actual position information is stored in the *value* field which stores its data as double.

Figure 3.7.: The Basic JSON Object

```
{ "dateTimestamp": long,
  "downloadTimestamp": long,
  "totalDuration": long,
  "result":[
    { "app": string,
      "duration": long,
      "usage":[
          { "session": string,
            "start": long,
            "end": long,
            "location":[
                 {"key": "lat",
                  "value": long
                 },
                 {"key": "lng",
                  "value": long
                 }
            ]}, ...
      ]}, ...
  ]
}
```

With these information, one has all needed data to create a fragments displaying one's daily activities.

DataSet initialization

The data managing class `DataSet` is designed to be a singleton to guarantee that every view and fragment of the application is provided with the same data. In order to obtain a `DataSet` instance, one can call `getInstance()` which returns an instance of `DataSet` or initializes one and returns it, if none exists. In the process of initialization, a new object of the class `UserData` which holds the user name and password, is requested. While the `UserData` object is created, the correctness of the data is checked by contacting the server. The connection process itself will be explained later in this section. After requesting user data, it is checked, whether data for colorization and for ignoring of applications are stored locally and can be loaded or not. If files are available, they are read and respective JSON objects are created. After `UserData` calls `DataSet` to pass a successfully created and tested user object, this object is stored and the date is initialized. Since the application provides support to display data of multiple days in one fragment, the current date is determined, stored and selected as start and end date, meaning that at the time of

initialization only one date, the current day is selected. Storing of the current date is necessary to check if a chosen date equals the current date and thus can be labeled as "Today".
Once the initialization part and therefore the creation of the `DataSet` instance is completed, the instance calls its own function `getApps(Listener)`, providing `null` as listener. This function requests a download of data of the selected day. `null` is provided, because the function is designed to call the main activity and notify it, that data is available for the first time, which causes the creation of the fragments relaying on that information.

Creating a new user

As mentioned, while instantiating, a new user object is requested from `UserData`. This is done by calling `getUserLoginData`. This function is provided with the main activity and an optional boolean as parameters. The main activity is needed to for different functionalities like storing data in a local file or accessing the Internet, which is explained later. The boolean can be set to true if one wants to create a new user independent of a existent user. In this case, probably stored user information are deleted. This is needed if one wants to switch the user login data the application uses. In case no stored user data is available, the application shows a dialog in which one can type in his or hers user name and password. In addition, one can check a box to tell the application, that his or hers information should be saved. Once the user hits ok, the data will be stored according to the check box and then the login data will be tested. If the server's reply to the test is positive the user data is passed to the `DataSet`, else the dialog is shown again.

Asynchronous tasks

To communicate with a server one has to note a few things. First, the communication has to be executed on a separate thread. One does not want the main thread to handle the download because waiting for a server response may occupy the thread for a few seconds and thus other actions are not executed during this time. And due to the use of an asynchronous thread one should ensure that at some point the downloaded data gets passed to the requester via a listener interface, in order to be able to display the data.
Android provides a good solution to handle short usages of asynchronous tasks. The `AsyncTask` api offers the ability to easily create and maintain a separate thread. To use its functionality, one has to extend `AsyncTask<Params,Progress,Result>`, define the three generic types and overwrite the following functions. `onPreExecute` gets called on the main thread and is used in this application to bring up an dialog, showing that something

like testing of user data is done at the moment. `doInBackground` is the actual function which runs on the new thread. It gets `Params` as input and returns `Result`. In this application the server access and the downloading of data are implemented into this function. The last function `onPostExecute` is called after the asynchronous task is finished. It is used to remove the dialog which was set up in `onPreExecute` and processes `Result` which was returned by `doInBackground`. The type of `Progress` is used as input for a fourth function `onProgressUpdate` which is never used in this application and thus should be `void` [4]. Once the class has been implemented, one can instantiate it and then call `execute(Params)` to execute the asynchronous task.

Server access

Now that the use of asynchronous tasks has been clarified, the usage in `UserData` should be explained. The three generic types of `AsyncTask` are defined as `string`, `void` and `boolean` and the instantiated class gets called with a string providing the URL. Next the function `doInBackground` is executed on a new thread and should therefore be described in detail. To download data from the server, one has to establish an URL connection. This is done by creating an `URL` object from the passed URL string and calling `openConnection()`. This function returns a `HttpURLConnection` which allows different adjustments, like setting the request mode or defining the established connection as input or output. The actual communication part is started with `HttpURLConnection`'s function `connect()`. Once this is called, one can access the downloaded data with an `InputStream` which is returned from `HttpURLConnection` object's function `getInputStream()`. All data received from the server is capable of being transformed directly into a JSON object. In case of the user data verification, the last remaining part is checking which value the newly created JSON object contains for the key *result*. If the value equals 0 or the JSON object is `null`, `doInBackground` returns `false` or else it will return `true`. `onPostExecute` then passes a the new user to `DataSet`, if the input was true or it will show the login dialog again.

The Creation of the URL

The creation of the URL is an important part of the application, as it provides the address along with the request to the server. The URL consists of three major parts. The first part is the domain of the server along with the api version of the service running on it and the query type. The query type can differ from request to request and is used to define whether one wants to test user credentials or wants to download data.
The second part is the detailed definition of the query type along with a nonce, application id and the user name. These and

the information of the third part are put into `LinkedList` of `NameValuePair`'s with respective names and values. This is done because the class `URLEncodedUtils` allows to easily convert the list into the needed URL format. The detailed definition of the request is a JSON object in form of a string, consisting of different fields like *model*, *start* and *end* time. For every query one has to obtain a new nonce to provide a unique fingerprint for a request. The application id which is used to identify the application and keep track of its accesses, was assigned by the server in start of the development of the application and has to be provided in combination with the name of the user, who is currently logged in.
Part three of the URL is the field *h* in the `LinkedList` and is a hash value created with *sha1* by hashing the request along with the user and application password. This way the server is able to verify the request as it knows the passwords and is able to recreate the *sha1* hash value.
The constructed URL would resemble the following example: `domain/apiVersion/query?data=URLdata&nonce=UniqueNonce&user=UserName&h=HashValue`.

Processing downloaded data

Once the data has been downloaded, further processing is done in order to reduce computation time for displaying of fragments. To understand why this is done, one has to know how the downloaded data is structured. Every event that the *BigBrother* application tracks, like application start, application end, screen on, screen of, etc. is pushed to the server as a stand alone JSON object. That means that each object contains among others, information about the time it has been observed, the type of the observation, for example application or position data, the action of the event, in case of an application it would be the starting or ending and in case of an application, the event holds a session id and a field with the application name.
The JSON object seen in figure 3.7, however, is structured by applications and not by time of event occurrences. To achieve this, one has to merge those application events with the same session id and add the location data from the respective event. As location information the data which first fits into the application's timespan is chosen. The algorithm which constructs this JSON object requires three nested loops. One is used to loop over the downloaded data and check, what kind of event was triggered, one to find the respective application object in the output JSON Array and the last one is used to find the respective session id in the array of application usages. This makes the function expensive to run, especially for a large number of events. Combined with the time that it takes to initialize a connection to the server

and download the data, the application would slow down drastically. Without further optimization steps, every request is send to the server, thus it consumes bandwidth and the application is bound to an Internet connection.

Storing downloaded data

The solution to this problems is the local storing of downloaded data. Once the data has been downloaded and processed, the dataset stores it in a file on the internal memory of the device. In order to minimize storage occupation, the application does not store every downloaded request, instead only those requests which concern daily activities. Requests like the testing of user credentials are not stored, as this has to be performed at every startup of the application to be sure of the correctness of the user data. Another requirement for storing of data is the date of the downloaded information. If the downloaded data concerns the current day it is not stored, because one can not be sure, that no other events will be added later and thus an incomplete day would be stored. Instead data representing the current day is temporary stored for five minutes and thus one will not access the Internet every time he or she selects the current date. To keep apart files of different days and users, every data to be stored is assigned with a filename consisting of the name of the `DataSet` 's called function, the user and the data object used to create the URL. This way every file can be assigned to the respective user and since the data object stores start and end time of the request, one can map the file to its respective day.

The loading of data

Every time the user selects a new day or a timespan of days, the function `getApps(onDataAvailableListener listener)` is called. This function creates a global array of `JSONObject`s and calls `manageMultipleDays(Int i, onDataAvailableListener listener)` with parameter 0 and a listener. The integer represents the next null occurrence in the global array and because it is the first call of the function, the integer has to be 0. `manageMultipleDays` looks up the start date of the selected timespan, adds i days to it and calls `getAppsAtDate` with the new calculated date and the listener as parameters. This function creates the a JSON object `data` which is needed for the creation of URL and calls `getData(listener, String, data)` where the String represents the requested function which would be *getEventsAtDate*. Finally `getData` loads the requested data by proceeding as seen in the example code in figure 3.8. The algorithm first checks, if the requested day equals the current day. If this is the case it checks, whether the cached version of the current day is older than five minutes. If it is older, it requests a download or otherwise it calls `onDataLoaded` with, among

Figure 3.8.: Checking, which files have to be loaded and downloaded

```
1 getData(...){
2   if(requestedDate == today && !olderThan(5) )
3     onDataLoaded(cachedJSONResultToday);
4   else if(requestedDate == cachedDate)
5     onDataLoaded(cachedJSONResult);
6
7   if(fileExists(getFilename()))
8     LoadFile(getFilename())
9   else
10    DownloadFile(getURL())
11 }
```

others, the current day as parameter. If the requested date is not the current day, it is checked, if the cached day is the requested date and in case of truth, `onDataLoaded` is called as seen in the first case. If the requested day is not available in the global cache variable, the function checks if a file concerning this user, day and request is stored and loads it if it exists. Otherwise `DataSet` requests a download.
Once data has been loaded from an internal file or downloaded from the server, it is filtered for applications which should be ignored. Those applications have been defined by the ignore app option in the action bar. The filtered JSON object is then stored in the previously created array. If there exists a day in the array which is null, the function `manageMultipleDays` is called again with a number representing the next null occurrence. Finally, if all days are not null, the listener is called and provided with the array of days.

Storing of settings

The `DataSet` is also used to store colors, selected applications and ignored applications. Each function which stores those data calls the main activity in order to reselect the current selected tab and thus refreshing the view. All functions called by options, with exception of the function used for the select apps option, store their settings in a local file to recover selections after the application has been terminated.

As seen, the `DataSet` is the central managing unit for all kind of data. Some functions and classes like the `DownloadTask` were originally planned to be a direct part of `DataSet` but have been moved to an own class file and file to keep the structure and overview of this core class.

3.5. Mapview

It has been described, how the basic layout is designed and how data is acquired. The next sections will explain how this data is visualized by different views, starting with the start up view, the *MapView*.

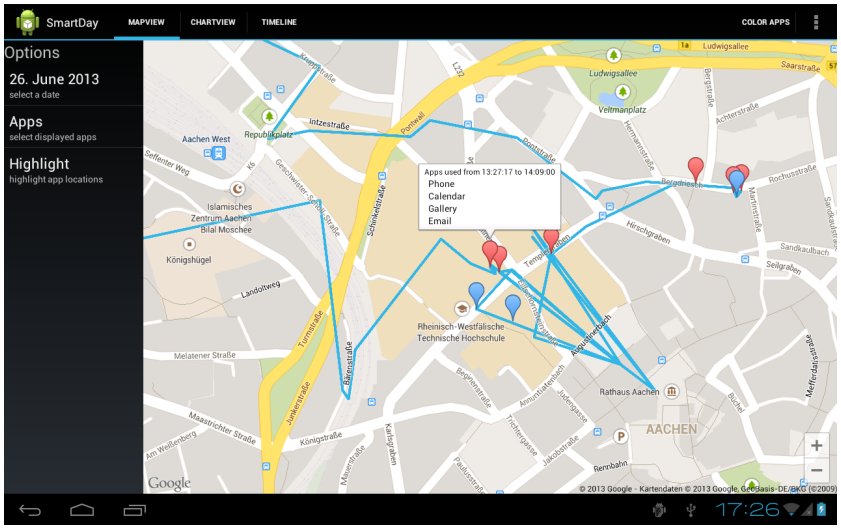
Usage of Google Maps

The map view is based on the Google Maps api, thus the class `SectionMapFragment`, which displays the map view, extends the `MapFragment` which provides the Google Maps api. In order to download and display the map and use its api, one has to register his or hers application at *Google APIs Console* [9]. For the registration, one also needs a Google account. Once the application is registered, the website provides an api key, which has to be integrated into the application's manifest. If this has been done, one has full access to the Google Maps api and can start to develop its own application or in this case fragment with it.

Changes during development

In contrast to the paper prototype, one will not see a preview of a taken photo, but instead he or she can highlight the positions where the camera application was used. Another change is the appearance and interaction of the displayed route. One is not able to press anywhere on the route to open a detailed view, instead he or she will be shown pins which can be tapped to bring up informations about the place. Furthermore, the highlighting and marking of applications will only effect the mentioned markers by colorizing them in a different color, as seen in figure 3.9.

Figure 3.9.: Final MapView



The options

The options on the left side of the view are the date selection option and the application selection option. The first option provides the ability to select a day or a timespan of days which should be displayed in the current fragment. The second option lets the user select applications which should be represented by markers respectively not represented. Both options were introduced in the basic layout section as they are not part of the view's fragment itself. The map view also provides an extra option not available in other views. *Highlight* grants the ability to select applications in a dialog fragment, much like the *Apps* option dialog, which leads to a recolorization of markers containing information about the selected applications.

Change appearance with GoogleMap

To interact programmatically with the map, one has to obtain an instance of `GoogleMap` by calling `getMap()`. With this object one can draw lines, change camera position and add markers. A marker is added at a specific location by calling `GoogleMap` 's function `addMarker(MarkerOptions)`. It returns an instance of `Marker` which should be stored, as it is needed to modify the marker later and there is no function to obtain set markers from `GoogleMap`. The `MarkerOptions` instance contains the adjustments made to a `Marker`, like the position or the content of the speech bubble. In order to create a costume layout for the markers, one has to overwrite the function `getInfoContents`. To create a line of representing daily motion route of the data, it is possible to use the function `addPolyline(PolylineOptions )`. Those polygon lines consists of points that are connected according to their order of input. Those points are defined in the `PolylineOptions` instance that its given as parameter.

Setting up markers and lines

Each view fragment in this application implements the `TabListners`'s `OnUpdateListener` interface. This allows the managing tab listeners, introduced in the basic layout section, to use the same function call for each fragment to provide data and update the view. When the update function is called, an array of JSON objects containing data for each day of the timespan is passed. In order to display locations and respectively used applications correct, the JSON objects have to be restructured. Those objects, as mentioned before, are ordered by applications and have to be ordered by location in order to easily access data for the creation of markers which represent places and locations with activities.
The algorithm building these new ordered JSON objects constructs them by looping over all applications and their usages and combines application with the same location and stores their

Figure 3.10.: The Map View JSON Object

```
{ "positions":[
    { "highlight": boolean,
      "lat": double,
      "lng": double,
      "start":long,
      "end":long,
      "dates":
        [ double,... ],
      "apps":[
          { "app": string,
            "usage":[
                 {"start": long,
                  "end": long
                 },...
            ]}, ...
    ]}, ...
  ]
}
```

name, start and end time. The result is structured as seen in figure 3.10.

Reordered JSON Object

As explained, the events are now ordered by position data. Each position contains information about its latitude and longitude, along with the dates of days at which the location was visited. The fields *start* and *end* represent the earliest and latest time of day, respectively, at which one has been at the corresponding location. The JSON array *apps* stores the used applications and in case that an application has been used multiple times, each one stores its usages. In order to easily determine if an application, selected in the *highlight* option, is part of a location object, the boolean *highlight* is set to true if one or more applications are contained by it or else it is set to false. In the process of reordering, the algorithm also filters those applications which should be hidden according to the *apps* option.

Creation of marker on polygon lines

Once the new JSON object representing the data for locations has been set up, one can start to add markers to the map. First off, the previous added markers and the polygon line have to be removed. To accomplish this task, one has to store the instance of `Polyline` and a list of `Marker` in advance because the Google Maps api does not offer access to the objects previously assigned to a `GoogleMap`. For the stored `Polyline` and every element

of the `Marker` list, one has to call `remove()` in order to remove the objects from the map. Then, one can loop over the JSON array containing the position data and create a `MarkerOptions` instance for every object in the array. Every instance of the `MarkerOptions` is assigned a `LatLng` object storing latitude and longitude, and a title containing a string of the start and end time, along with a string representation of the JSON array *dates* of the respective position. Each parameter of the string is divided by an unique pattern in order to be able to split the string in later progress. The class `MarkerOptions` also allows to set a snippet which is used to display a subtitle. This snippet stores all used applications, again divided by a unique pattern. This has to be done, because the function `getInfoContents` which is called if the user taps a marker, only provides a `Marker` as input and this only allows to access position, title, id and the snippet. But since one needs to know the dates, times and applications, one can either map the marker's id to the respective JSON object and store this in advance or one can divide the strings in order to recreate the information.
After the `MarkerOptions` have been created, one calls `GoogleMap` 's function `addMarker(MarkerOptions)` and stores the returned `MarkerOptions` object in a list. To create a `Polyline` on has to call the map's function `addPolyline(PolylineOptions)`, where the `PolylineOptions` contains a set of `LatLng` objects which will be connected in the order in which they were added. The function returns a `Polyline` object which is stored and can be used to delete the line and to change its appearance, for example the color and thickness.

Detailed position information

If the user taps on a marker, a speech bubble with detailed information about the tapped location pops up. The speech bubble's layout can be changed according to one's needs. In this application, the detailed view is described by an linear layout, showing date and time of visitation on top and all used application names below. The needed information to fill the layout's views are gathered by dividing the strings which are stored in the title and snippet. In addition, the fragment implements the interface `OnInfoWindowClickListener` which provides the function `onInfoWindowClick`. This function is called once the user taps on the speech bubble and is used to switch the fragments of SmartDay. In this case the user will be presented with the *Timeline*, focusing on the timespan in which the position was visited. This allows a detailed view of the sequence of passed activities and the duration of their usages.

Camera positioning

To grant the user a better experience using the map view, one has

to take in mind, that the map view's camera position has to be adjusted according to the displayed positions. If one does not take care of the camera, the user would be presented a look at the at the Atlantic Ocean, because the standard camera adjustment looks at zeroth latitude and longitude. To reposition the camera programmatically, `GoogleMap` provides the function `moveCamera (CameraUpdate)`. This function sets the camera according to the `CameraUpdate` which in this application is created with a call of `CameraUpdateFactory.newLatLngZoom(new LatLng(float , float), float))`. The `LatLng` object defines the position on the map and the float represents the zoom factor, set to twelve if the application readjusts the camera.
Repositioning is needed for example if the view is visible for the first time or if another view fragment brings up the map view and wants a specific marker in focus.

Receiving data from other views

As mentioned, not only can the fragment cause a switch to another view, it can also be switched in by other fragments and be provided with data causing a speech bubble to pop up. The `OnUpdateListener` interface which provides the function `onUpdate` also provides the function `putExtra`. This function is called by the tab listener and granted a JSON object as input. This JSON object should contain informations about latitude, longitude and a time stamp. The function then searches the JSON object containing the applications ordered by location for a position object matching longitude and latitude. If a position is found, all application timespans are searched and if at least one timespan contains the provided time stamp, the index of the position object in the array of positions is used to directly access the respective `Marker` object stored at the same index in the list. This can be done, because first the location ordered array is created and then the marker list is created by sequential working off the array, thus both objects use the same index for their respective location representation. Then The `Marker`'s function `showInfoWindow` is called, which leads to the desired focus of the marker and the pop up of the speech bubble.

Although the final version of the map view differs from the described view in the paper prototype section, it still grants an overview of the visited locations. One is able to highlight positions of personal interest and get detailed information about the applications. It is capable of supporting the visualization of displaying multiple days, which provides the user with the additional ability to adjust the view in such a way, that it fits one's personal needs.

3.6. Chartview

Displaying the data in a time opposing manner provides an easy opportunity to draw a conclusion about productivity. If one sees at a glance that he or she used a messaging application for 40 percent of the total time, one may wants to rethink his or hers working behavior. An intuitive way of displaying the percentage partition is a pie chart. Pie charts are displayed in the applications view fragment *ChartView*.

AChartEngine library

The charts are created with the help of the *AChartEngine* library [1]. The library is open source and offers a variety of different charts and graphs. Working with this library was enjoyable as it has a good documentation and large detailed example project which clarified the usage of the library's classes and functions. In order to have access to the library, one has to store its JAR file in the *libs* folder of the application's source code and add the library in the project properties. This way the library can be used and is build in to the apk file which is installed on the Android device.

Description of the ChartView

The ChartView's layout is split vertically into two parts. The left part occupying ca. three fifth of the layout, contains the charts displayed one below the other and are ordered by date in ascending order. Each chart has a title above it displaying the date of the day and the total time of daily application usage in minutes. The chart itself consists of slices which represent the percentage of time the respective application was used. Each slice is painted according to the color assigned to the application by the color apps option. Furthermore the slices are labeled with their percentage and the application name it represents. If one taps on a slice, it gets highlighted by pulling it a bit out of the circle and the fragment adds new information to the right part of the layout. This part fills the remaining two fifth of the layout and displays detailed information about the currently tapped pie chart's slice. These information are the application name and its usages. Those usages are the start time of the application and the total duration along with a text "show location" on the time's right. All information are displayed vertically, ordered by time in ascending order. The described view can be seen in figure 3.11.

Options

Options affecting the views appearance are the select date option and the select apps option which let the user choose, what applications to display. Another adjustment that has a effect on the visual appearance of the view is the color apps option. The option was described in section 3.3 and allows the user to reassign the colors of applications which are set randomly in first place.

Figure 3.11.: Final version of ChartView

Interacting with the view

One has the possibility to interact with the view. He or she can swipe up and down to scroll through the different days represented by charts. Tapping on charts or, in more detail, its slices, will highlight them in the described way. The right side of the layout which provides the detailed information can be used to focus a specific marker in the map section by tapping "show location" and it can be used to focus a timespan of the application in the timeline view by tapping on the specific time.
In order prevent an overload of the view with too many slices, applications with a usage time of lesser than five percent are merged into a slice labeled *other*. Tapping this slice shows all application names represented by it on the right side of the layout and tapping on a name reveals their respective time and position information. These information also lead the user to the map view, respectively the timeline.

Differences to the paper prototype

As one may have recognized, the view differs from the original concept, in particular, the lacking of position based charts and categorized applications. The reason that applications are not categorized is that the grouping of them is context sensitive and that there was not always enough data to accurately determine the current reason for an application's usage. For example, one may uses the Internet browser to look up a work related question or he or she watches a video for one's own entertainment. The data provided does not differ between these cases. For an accurate assessment, the data would need to provide the browser's history and in general more information about the application's received

input. With such a detailed dataset one has to implement algorithms, searching for key words and analyzing the input, which, in general, are data-mining algorithms.
The position based charts, which should have been provided for work, on the move and home, have been discarded, as they also need information about context to be accurate. For example, a person working at home could not have different charts for work and home, as they are the same location. The charts would need to take the current time and category of the used application into account. Defining a position as home and work would have been possible, but their usage and functionality would not have provided a satisfying experience.

Creating charts with AChartEngine

To create a chart, one must first create a `CategorySeries` and a `DefaultRender`. These classes are provided by the library and contain the data for the `getPieChartView` function of the `ChartFactory` class. Both classes, `CategorySeries` and `DefaultRender`, have to contain the same number of added objects or else the `ChartFactory` throws an exception as it tries to access the same index for both collections and runs into a null pointer. `CategorySeries` contains the application names and their respective time. To add an application, one uses `add(String, double)` with the respective name and time of usage. The `DefaultRender` is a collection of `SimpleSeriesRenderer` which are added with a call of of `addSeriesRenderer(SimpleSeriesRenderer)`. Those renderer define among others, the color and highlight flag of each category in the `CategorySeries`.The `DefaultRender` also provides adjustments which allow for example to add title, show the legend or disable the zoom function. Once the categories and renderer are set, the call of the `ChartFactory` creates a view which can be added to a layout. In order to react to user input, the view is assigned with an `OnClickListener` which sets the highlight flag of the respectively tapped slice to `true` and adds information about the applications to the right part of the chart view.

Processing of data

The JSON object containing the data for the chart view has to be processed in order to structure the needed information for the charts. For each chart the view creates three arrays, containing the application names, their respective used time and their assigned color. For each application, the name and its used times are gathered from the provided JSON object and each one is stored in a cell with identical index in the first two arrays. The respective application's color is stored in a cell with the same index in the third array and is provided by the `DataSet`. The color is loaded from a file or created randomly and then stored, if the

application gets colored for the first time. After processing the data, all three arrays have the same length and each index represents the same application.

In the process of creation of the chart, it is checked for each applications it is checked, if the total time covers less than five percent of the daily usages. If this is the case, the application's array index and name is stored in a JSON array. Its time is summed up with all other applications representing less than five percent of the day and they form the slice with the name *other*.

The detail view

As mentioned, the detailed view on the right side of the layout is added and updated, if the user taps on the pie chart. The detailed view's layout consists of three columns, where each column is a linear layout, arranging information vertically. Every text view in the second and third column, excluding the headers, has an `OnClickListener` assigned which creates a JSON object containing information for the to be swapped in view fragment. In case of tapping a timespan, the application's name, the start time of the timespan in seconds and the respective date of the chart is put into the JSON object. For the locations, the assigned `OnClickListener` is an own class which has to be created as it needs to store longitude and latitude. This `LocationClickListener` creates a JSON object containing application name, start time of the timespan, longitude and latitude. To switch the displayed view, the listener calls the main activity's function `switchTab(int, JSONObject)` with the respective position of the to be swapped in tab and the JSON object.
In case that the user has tapped the *other* slices on the chart, the first column gets filled with text views displaying the names of the grouped applications. These views are assigned with an `OnClickListener` which adds the respective timespan and position information to the other columns. The view uses the previously stored JSON array with the application names to identify their respective JSON object in the provided data.

Finally the chart view provides a simple, statistical overview of used applications, where the user can easily determine which applications have been used mostly and which ones are rarely executed. The possibility to replace the the fragment from within the view allows the user to directly access more context information by highlighting specific locations and timespans in the respective swapped in view. Also, the view switching partly compensates the lacking of position based charts, as the user can see at which positions the application has been used and what he or she used

before and afterwards, although it is not as comfortable as described in the paper prototype.

3.7. Timeline

The third and last view visualizes the data in a chronological order. It shows the user an illustration of his or hers day, giving context about applications being used before and after a specific timespan. It also provides the user with the ability to highlight specific usages and applications. Since this view displays applications sequence above a time line it is simply called *Timeline*.

Description of the visual appearance

The view fragment displays a time line representing the hours of a day. The line starts at 00:00 and ends at 24:00 and between these boundaries, markings for every hour in form of a vertical line are displayed. Each hour marking has the respective hour written below it. Above the time line, rectangles which represent timespans of applications, are drawn. The rectangles are painted in the application's respective color, as it is done in the chart view. On top of the view, one can find the date represented by the view. The fragment also owns a detail view which is located under the timeline. This view consists of two parts. The first part is on the left and shows the names of the applications ordered descending by time, with bars on the right representing the percentage of total usage time. The second part is located on the right and contains information about timespans and locations as known from the chart view.

Figure 3.12.: Final version of the Timeline

Interacting with the Timeline

As seen in figure 3.12 the line displays the time from 7:00 to 12:00. The line is shortened due to the possibility of zooming. One can use a two finger pinch to zoom in and out to focus on a particular region of the timeline. This is helpful, especially when one uses many applications for a short time in a high frequency. When zoomed in, the user can scroll through the zoomed in timeline by swiping left and right. A double tap allows the user to rescale the view back to its original size, such that the whole day is visible or he or she will be shown a close up of the double tapped point if the whole day was visible. When the fragment is created, the detailed view is hidden to provide an overview over all selected days. To unveil the detail view, one must tap on the timeline. If one taps a displayed usage then all rectangles of the same application are lowered in order to highlight all usages. The specific tapped timespan is lowered down to the timeline. In addition, the detail view highlights the select application and timespan by coloring its background blue. Tapping an application name on the detail view, causes all application usages to lower and tapping on a specific timespan lets the timeline focus the rectangle and lowers it as if it was tapped manually. The text "show location" causes a fragment switch and will display the map view with the respective marker highlighted. To hide the detail view again, one has to double tap on the percentage bars.

Differences to the paper prototype

The timeline's layout and functionality is mainly as described in the paper prototype. The main difference is again the lacking of consideration of position data. The original paper prototype would display the locations above the rectangles, but this was rejected due to the same reasons as there were for the position based charts. To compensate that unavailable feature, the view can switch to the map view and highlight a specific location on the map. Another small difference is the non presence of the application names inside the rectangles. This has not been implemented, as the rectangles normally are to short to be able to display the full name of a application, even when the view is zoomed in. Also, the application rectangles are not displayed above or below each other as only one application session is active at a time.

Canvas and the time line

The Timeline was implemented by creating a new class, extending the `View` class. The new class `TimeLineView` has to overwrite the function `onDraw` which provides a `Canvas` instance. The `Canvas` is the important part of the view, as it defines the to be drawn object or view. The canvas provides different functions like `drawLine`, `drawRect` or `drawText` which provide the ability to change and manipulate it. First, the canvas is used to draw the

line with the time of day by calling `drawLine` and providing the x and y coordinates of the start and end point of the line along with a `Paint` object defining the lines color and thickness. The x coordinates depend on a zoom factor which is stored globally and is updated if the user executes a pinch to zoom gesture. Generally this means that the provided coordinates do not need to be in range of the display size. To ensure that certain points are visible on the display, one has to translate the canvas accordingly. This is also needed to display the scrolling of the view. Once the horizontal line is drawn, the markings for every hour, half an hour or even minutes have to be drawn. The number of markings to be drawn depends on the zoom factor. If the zoom factor is not large enough, the markings would overlap each other and the view starts to look confused, thus the number of markings and their respective displayed time of day is depending on the zoom factor.

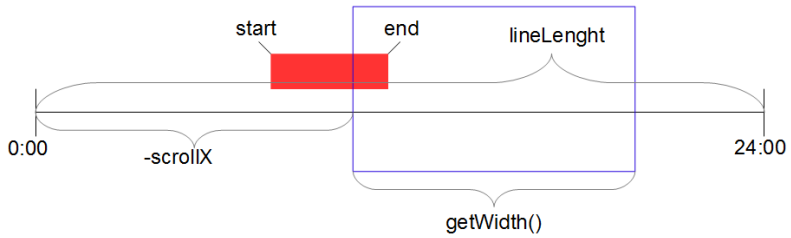
Processing input data

To efficiently draw rectangles, one first has to process the input data and create a new JSON array representing the rectangles. This is needed to provide a smooth, reactive view, as the function `onDraw` is called every time the view appearance changes, for example while zooming or scrolling, because these gestures require a redraw of the canvas. Thus the data needs to be preprocessed when it is assigned to the view. During the process, a JSON array is created which contains JSON objects representing the to be drawn rectangles by storing start and end time along with the application's name.

Drawing of rectangles

Then, in `onDraw`, the algorithm iterates over all JSON objects contained in the array and chooses which will be drawn and which not. This is decided by calculating the respective rectangle's start and end edges and checking whether at least one of them is visible in the current clipping of the canvas. The calculation proceeds as follows. First, calculate the number of pixels that can be used to visualize one second of usage by dividing the line length by 86400, the number of seconds in a day. This differs most of the time, because the line's length depends on the scale factor, set while zooming. Then calculate the start and end edge by multiplying this value with the start time, respectively end time. The last step is to check whether the start or end lies inside the visible part which is translated by `scrollX`. This is the case if `end +scrollX` $\geq$ `0` or if `start+scrollX-getWidth()` $\geq$ `0`. An illustrative explanation can be found in figure 3.13. If the rectangle passes this visibility test, the canvas' function `drawRect` is called with parameters defining the left, upper, right and lower border of the rectangle and a `Paint` object defining its color.

Figure 3.13.: Final version of the Timeline



Implementation of gesture controls

To be able react to user input, one has to overwrite the view function `onTouchEvent` and pass the provided `MotionEvent` to one's own classes, which are in this case the `ZoomListener` and the `TapListener`.

The ZoomListener

The `ZoomListener` extends the `SimpleOnScaleGestureListener` of the `ScaleGestureDetector`. Among others, the function `onScale` has to be overwritten in order react to the zooming gesture. This function is provided with a `ScaleGestureDetector` which holds the scale factor of the gesture and the focused x coordinate. The scale factor is stored globally and the focused x coordinate is used to recalculate the `scrollX` position which assures on a redraw, that the coordinate stays focused. The last action `onScale` performs is a call of `invalidate` which causes the timeline to be redrawn.

The TapListener

The `TapListener` is a more complex class. It extends the `GestureDetector`'s `SimpleOnGestureListener` and is able to recognize the following gestures by overwriting their respective functions.

Scrolling

One scroll horizontally through the timeline with the restriction, that he or she can not scroll past the the 0:00 or 24:00 mark. `onScroll` takes care of scrolling and additionally disables the vertical scrolling for the parents `ScrollView`, if the scroll direction in x-axis is at least 20 percent of the y scroll direction.

The `onFling` function allows the user to fling the view, that means, if one raises the finger from the device while in a scrolling motion, the view proceed to scroll like a wheel slowly stopping to move. This is done by setting up a `ValueAnimator`, which animates this movement. It is provided with a listener called `AnimatorTick`. The listener's function is called by the `ValueAnimator` and has to set up the the new scroll position in x direction, as long as the `ValueAnimator` calculates a new one. Once the new x position does not differ from the old one, the listener stops the `ValueAnimator` to prevent unnecessary calls of `invalidate`.

To stop a fling before the animation is over, one can tap on the

screen. Then `onDown` is called which stops the fling animation. The function is called every time the user touches the display but does not have any further functionalities.

Double tap

Double tapping causes a call of `onDoubleTap`. If the timeline is fully shown, it is used to zoom into the double tapped region by a predefined value and zooms out to show the full timeline, if it was not visible. When zooming in, the function also recalculates the new scroll position in order to focus the tapped point.

Adding details

The last function that is overwritten, is the `onSingleTapConfirmed`, which is called after the system is sure, that no double tap occurred. It interprets the tapped position as the respective time on the timeline and checks the JSON array containing the rectangles, if the tapped time belongs to a rectangle. If an application was found, the function stores the time and application name and adds the detail view. The details are added, even if no rectangle was hit.

The detail view

A detail view is added to the parent layout of the `TimeLineView` which should be `LinearLayout` with an vertical orientation in order to place the details below the timeline. The details are an new constructed class called `TimeLineDetailView` which extends the `View` class. When `TimeLineView` calls the private function `addDetail`, it is first checked, if the parent layout contains a detail view and in case of falsification, a new one is created and added. Then the view is passed the JSON object containing all data about the day and the possibly selected application and its time.

Setting and processing of data

`TimeLineDetailView`'s `setData` processes the provided JSON object, to order applications by usage time and calculate their respective percentage bar length. First, all stored applications are extracted from the input JSON object and are put into an array of JSON objects, storing their name and duration. Then this array is ordered by duration and each object is provided with an additional value for the bar length of the application. Afterwards the maximal bar length is calculated and stored in respective JSON object. The last thing to do is the loading of colors and calling `invalidate` to force a call of `onDraw` which will paint the application names and bars.

Drawing of information

`onDraw` draws the view as seen in the description of the `TimeLineView`. It looks up application names and the bar lengths in the array of JSON objects and draws them in descending order one under another by accessing `Canvas`' functions like `drawRect` or `drawText`. To highlight the selected application, the function checks every name if it equals the name stored in the string

selectedApp. If this is the case, the application name is drawn on a rectangle of different color. selectedApp can be set from the timeline by tapping a bar or from the details by tapping an application name or one of its respective usages.

The gesture detector

The drawn view is also provided with an TabListener extending the SimpleOnGestureListener and is used to capture double taps and single taps. onSingleTapConfirmed uses calculations to translates the tapped position to the respective application's name and uses it as parameter for a call of selectApp . Then it calls the TimeLineView's function selectApp with the application name as parameter. The function of the timeline stores the application name and causes a redraw of the view, such that the application is highlighted. Double tapping causes a call of onDoubleTap, which calls the global function close, which removes the detail view.

Adding further details

selectApp is used to tell the detail view that an application has been selected. The function stores the applications name in the variable selectedApp and calls addDetails with the application's name and if available, the time of a timespan. addDetails is used to add information about usage times and locations to the detail view or, in case no application was selected, deleting those information. It basically proceeds the same steps as described for the chart view's addDetail function. First a column containing Views with start time and duration for the respective application is created. Each TextView is assigned with an OnClickListener which highlights the tapped timespan and calls the TimeLineView's function selectApp which highlights the respective application timespan in the timeline. The second column contains the timespan concerning location listener which causes a switch to the map fragment. This listener also contains the respective longitude and latitude, which is put in the JSON object, needed for the switch.

The timeline shows the user his or hers daily activities in a clear overview with the ability to focus on details in specific timespans. The displayed days resemble bar codes or unique finger prints and provide the user the ability to spot differences in usages by comparing the color patterns of the day. The view itself was built without the help of libraries to show what can be done with simple drawings and interpretations of gesture inputs. The result is quite satisfying but still has a lot of unused potential, like the visualization of positions or the emphasizing of regularly occurring patterns.

Chapter 4 Evaluation

Part III

Chapter 5 Related Work

This chapter will talk about another approach of visualizing the provided data. Thomas Honné's project of his Master thesis "Interactive Visualizations of Activity Patterns in Learning Environments" [11] will be discussed and compared to the result of this thesis, *SmartDay*.

Visualizing

Honné's project and thesis describe a way of visualizing daily activities with a special focus to learning environments. Both project try to visualize the data in different ways, *SmartDay* in a native way on Android and Honnè's project in a non native way with the help of a website. Since both visualizations need access to the Internet, only *SmartDay* is able to store data locally and display them even without an Internet connection. But the website based service also has its pros, because most platforms provide a browsers which are capable of displaying Java Script, which is mainly used for the website, the project can be used on more platforms and devices.

Because both projects use data provided by the same gathering application *BigBrother*, this comparison can mainly focus on the visualization part, as both projects have the same precondition.

The map

Both projects use different views to represent the provided data but both use a map for position data. Honné's map shows, in contrast to the visualization of *SmartDay*, visited locations as unconnected circles. These circles differ in size and get larger, the longer one uses applications at this point. Those circles can consist of more than one position data, meaning that one can adjust a size which represents a radius in meter in which all data will be merged into one circle. This forms clusters on the map and creates personal points of interest, showing the user in an intuitive way, where he or she has used his or hers device the most. The map in *SmartDay* connects each point and does not weight them

by total amount of times applications have been used, nor are positions merged. This can be helpful to reconstruct routes but can also look messy sometimes.
Thomas Honné's map view also provides further details when clicking on the circle. The user is then presented a small window which shows a pie chart displaying the applications used, grouped by their productivity level. Furthermore, a bar chart, similar to the timeline's detail view, is displayed, ordered descending by total time of usage. And at last one can see all events that were tracked at this position, listed as text in chronological order. *SmartDay* may have pie charts too, but these are in a separated view, not position based and are not directly accessed by the map view. The details and completeness of the website's view dominates the map view presented in this application.

Grouping of applications

Honné's project allows the grouping of applications. This is used to rate the productivity of one's daily activities. The grouping is done by assigning productivity values to the the applications. Those values are for example *productive*, *neutral* and *not productive*. This resembles the idea of the paper prototype which should allow the user to categorize applications in *SmartDay*, but was neglected due to unavailable context information.

Visualizing of patterns

The website provides a section not included in the native application, called *Patterns*. This section makes use of Iurii Ignatko's work to recognize patterns with the help data mining algorithms in the provided dataset [12]. It displays connections between application usages, for example it may show the pattern, that if one uses the calendar application, he or she is likely to use the email client afterwards. This can be helpful especially to track down the root of usage patterns which distract one from his or hers work.

Productivity and line charts

Another view of the website is called *Productivity*. This view displays the same data as the detail window of the map tab, but takes adjustable timespans as input. This is a good solution with respect to productivity, as one can directly see his or hers daily, weekly or monthly usage of productive and non productive applications. The view *Line Chart* presents the user a view which displays a line chart representing the number of events over time. The chart is freely adjustable and one can add lines with restrictions to events and applications, colorize them and select a timespan for it. It is an interesting feature which allows to compare different activities and applications.

The website's views are reasoned and well structured. Especially the map view has some clear advantages over the application's

view. The general concept and use of position based charts is a step ahead of the applications visualization. But the website still has some unused potential. For example, the productivity tab should take time, date and position into account, when weighting an application's productivity, as it may be okay to play a game at 22:00 at home on a Friday evening. In addition, the website lacks of a view, where the user can see his or hers day in a chronological order, like it is displayed in the application's timeline view.

Chapter 6 Conclusion

This thesis has shown the developing of an application for an Android device, which visualize data of daily activities in a native way.

6.1. Summary and Review

This thesis has shown how an application as a tool for self reflection is created. First, the reasons and objectives for this thesis have been discussed, describing the need of a tool for self reflection, because one is not able of keeping track of his or hers daily smartphone activities manually. Then background information about the data of ones daily activities and its origin, a definition and counter example of native visualization and an explanation of self reflection have been given. It has been explained that Android would be the operating system of choice and the development device has been introduced.

Paper prototype and application objectives

In the second part, the first idea for layout and functionality of the application was described with the help of a paper prototype. It described the use of three major views and explained their functionality. The *Map View* should provide an answer to the question "Where have I used my device?", the *Chart View* is used for the question "For what have I used my device?" and the last view, the *Timeline*, would answere the question "When have I used my device?". Each view was supposed to provide enough information to partly answer the other questions and work as a stand alone view, but the best experience would only be made when using all three views in combination.
It has been stated, that this application should perform as a central information system which provides the user with all necessary information, presented in an intuitive way. And since the

project focuses on a native solution, it should test what is possible to create without the use of Java Script and other external help.

Implementation

The implementation part described how each view and the basic layout has been realized and showed how the objective of providing the best experience while using views in combination, has been fulfilled with each view linking to necessary other views to get more details about specific parts of the view. But the application could be improved, in case of the stand alone views, as the *Chart View* and *Timeline* do not display location information. It has also explained the use of third party code, like the color picker and the AChartEngine and the use of different Android apis.

The evaluation shows how different users reacted while using the application and showed, where the application has its strength and weakness. It also demonstrated, where the application can be improved in future works.

Conclusion

The delaying of position based charts and the position labeling in the timeline for future work had to be done, because the only acceptable solutions would require context analyzing which could not have been realized and implemented in a satisfying way for this Bachelor thesis, due to time restrictions. But none the less, one can say that the application *SmartDay* can be used as a tool for self reflection, because it provides views which answer the initially stated question in a visually pleasing and satisfying way.

6.2. Future Work

As mentioned in the previous section, the application has potential to be enhanced. This potential can be used for future work and possible the parts of the application to improve will be demonstrated in the following.

Improving controls

First, general improvements for controlling the application could be implemented. For example gestures like a two finger fling to switch the displayed view, or tap and hold to open the color menu of a specific application.

Another enhancement would be the adaption of the application to run on smartphone. Right now, the visible space for the main views is to small and there are some issues with option headers and font sizes. To realize this, the implementation of a new layout

is required along with moving the left options menu to an extra option menu.

New view

An additional view displaying a line graph could also be an improvement. This graph would display the weekly usages of an application and could be personalized by adding and deleting applications or groups of applications.

Improving existing views

But the existing views can also be improved. The map view could be more structured, for example adding numbers to the markers, different colors for different days and merging markers that are too close together. The views could be extended with displaying of position data. The chart view would be added with charts displaying applications used at work, at home and on the move and the timeline could label timespans according to the visited locations.

Context sensitive evaluation

Context sensitive information, to differentiate between time spent at work and free time, would also enhance the self reflecting views, as one may work from home or is working while on the move. Further steps would be the categorizing of applications into groups like productive, neutral and entertainment, such that a pie chart could display application groups and one could directly see if has been productive at work. Taking the idea of context sensitive information and grouping a step further, the application could analyze all used applications and weight them according to their assigned group, time and location to display the user if he or she was productive at that day.

Extending the provided data

Potential improvements in case of self reflection may arise from the provided data itself. The possibility to not just track data from one's mobile device, but instead from his or hers PC or laptop would greatly improve the available data. But to realize this, respective application for stationary devices have to be developed as they do not use Android, but run Windows, Linux or Mac OS. A potential future work could also be the development of such a program.

As one has seen, with the options of extending and enhancing the application in direct way by improving the views or in an indirect way by extending the dataset, this thesis has created a basis for future projects.

Appendix A Bibliography

Printed References

[10] Sarah Haller. “Pektorale. Eine Studie zur Form, Bedeutung, Verwendung und Praxis anhand der Dahschur-Pektorale aus dem Mittleren Reich des Grabkomplexes Sesostris’ III.”. BA thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2013.

[11] Thomas Honné. “Interactive Visualizations of Activity Patterns in Learning Environments”. MA thesis. RWTH Aachen University, 2013.

[12] Iurii Ignatko. “Applying Data Mining Techniques to Discover Patterns in Context Activity Data”. MA thesis. RWTH Aachen University, 2013.

[13] Thorsten Kammer. “Capturing Activity Context in Mobile Learning Environments”. MA thesis. RWTH Aachen University, 2013.

Online References

[1] *AChartEngine*. URL: `http://www.achartengine.org/` (visited on 06/05/2013).

[2] Stefan Beiersmann. *Android erreicht 70 Prozent Marktanteil in Europa*. URL: `http://www.zdnet.de/88160639/android-erreicht-70-prozent-marktanteil-in-europa/` (visited on 14/08/2013).

[3] *Big Brother Application*. URL: `http://learning-context.de/text/3/Collectors` (visited on 13/08/2013).

[4] Google. *Async Task*. URL: `http://developer.android.com/reference/android/os/AsyncTask.html` (visited on 24/08/2013).

[5] Google. *Development Tutorials*. URL: `http://developer.android.com/training/index.html` (visited on 15/08/2013).

[6] Google. *Distribution of Android Versions*. URL: `http://developer.android.com/about/dashboards/index.html` (visited on 14/08/2013).

[7] Google. *Latitude discontinued*. URL: `https://support.google.com/gmm/answer/3001634` (visited on 12/08/2013).

[8] Google. *Unser mobiler Planet: Deutschland. Der mobile Nutzer*. May 2012. URL: http://services.google.com/fh/files/blogs/our_mobile_planet_germany_de.pdf (visited on 06/05/2013).

[9] *Google api*. URL: https://code.google.com/apis/console/ (visited on 26/08/2013).

[14] Yuku Sugianto. *Android Color Picker*. URL: https://code.google.com/p/android-color-picker/ (visited on 23/08/2013).