Kangaroo
a mobile, location-aware task planer

Alexander Gutjahr Andreas Walz Daniel Schauenberg

April 25, 2010

Contents

1 Introduction

Kangaroo was developed as a team project at the Albert Ludwig University of Freiburg. The goal was to develop an application which knows all events and tasks data of a user. Based on this information, the best route to attend all events and complete as many tasks as possible, is computed. In this document, all information which was created during the development is collected.

Chapter 2 describes the requirements as defined by the Lehrstuhl, as well as the resulting development goals and the abstract system structure that was designed to fulfill these requirements.

Chapter 3 contains information about project planning and the development process. This means the initial and refined project plan as well as the description of the development process.

The process of evaluating suitable platforms is described in chapter 4. The main focus here lies on the three mobile Operating Systems iPhone OS, Android and Windows Mobile. Others are also described briefly. Additionally the decision which routing engine to use is described in this chapter, too.

The actual development process is the focus of chapter 5. Here the platform specifics, which impact the system design, are explained. Furthermore the main parts of the application, as well as some important implementation details and caveats are discussed.

The last chapter then provides a summary of the work with some reflections on the completed work as well as an outlook, how to use this for future development.

2 Definition of project scope

2.1 External requirements

The following requirements were defined prior to the project start. The primary project goal is the fulfillment of these requirements.

- develop a program that generates and manages dayplans for its user
- the program shall be location aware
- the program needs to run on a mobile platform
- the program should not leave digital traces of the user that can be followed by a 3rd party
- the development should result in a coherent and usable application.
- this project is a first step towards the development of a more powerful application with the purpose of protection of the users privacy

In order to meet these requirements, more concrete development goals need to be refined from these from these abstract goals. These can be found in chapter 2.2

2.2 Resulting development goals

With respect to the abstract requirements imposed upon project start a brain-storming was performed in order to gather ideas and and identify possible problems in the realization of the project. Figure 2.1 shows the mind-map resulting from a reflection of these abstract requirements. Based on the abstract requirements and the results of the brain-storming these development goals and principles ca be derived:

- The program should run on a mobile platform. At the moment several mobile operating systems are in use, competing for market shares. Since a consolidation of this market has not jet set in, the most appropriate platform for our use case has to be chosen.
- For the application to be location aware, the mobile platform needs to provide access to location data from a GPS or a radio cell based location system.
- The location awareness of the application is only then useful, when it is able to monitor its location continuous or in short intervals without interaction from the user and without disturbing the normal use of other applications. Therefore a part of the application needs to run in the form of a background process. This results in the requirements for the platform to support background execution of code and for the application to be highly energy efficient. A too high energy consumption by a continuous background task would drain the battery fast and render the whole application unusable.

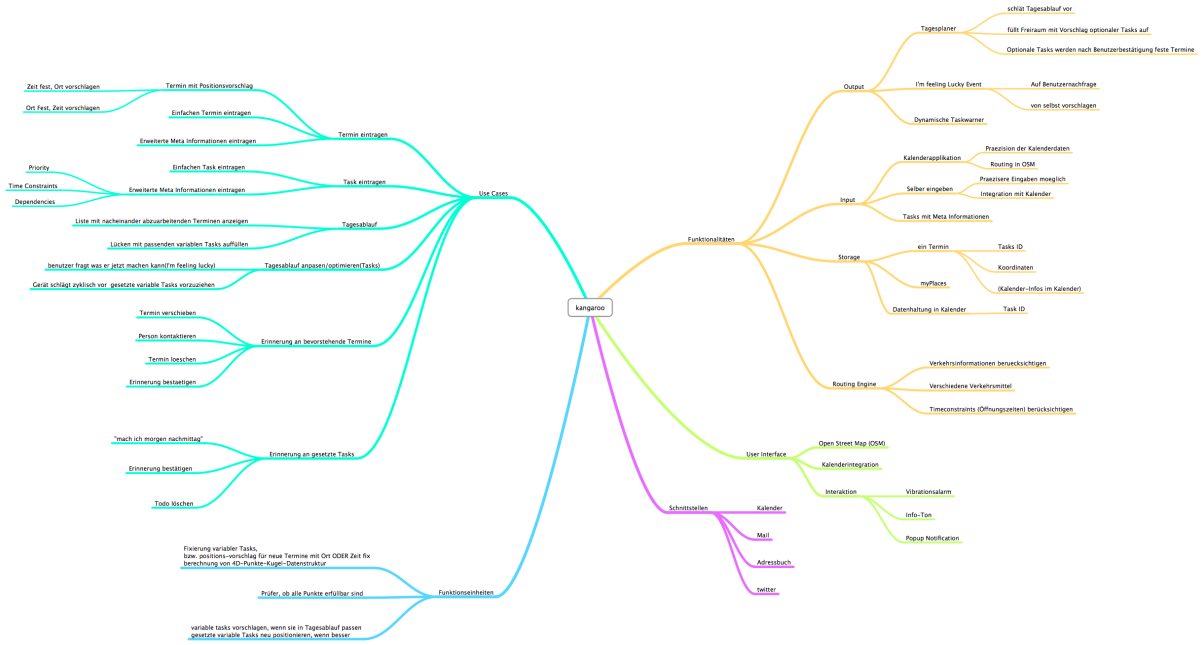

Figure 2.1: Mindmap with project goals, requirements and elements

- For the location information to be helpful the application needs to be able to perform routing operation such as route finding, estimation of travel durations and localization of meaningful places. Since this is expected to be both complicated and with high computational complexity, an suitable existing routing solution needs to be found, evaluated and adapted. This is considered as one of the most critical requirements.

- To work with users dayplan, the application needs a way to handle tasks and appointments. In order to make the usage of as intuitive and easy as possible, the calendar and task-management functionality provided by the target platform shall be used.

- In order to avoid the problem with digital traces and information leakage, the application shall run strictly local and shall be self sustained. This means all information that is used needs to be present on the device and all calculations need to be performed locally. No internet connection should be required for the application to function.

- In order to make a re-usage of parts of the software or a further development of the application as a whole possible, the following properties need to be maintained during the development:
  - a clean and adaptable system structure in order to enable the re-usage of single system parts in different projects
  - a high level of code quality to minimize the time future developers need to adapt and learn to use the system.
  - a detailed and well structured documentation to make future work with the code easier.

  - only well-known and proven open-source tools and and software components shall be used that do not pose any problems in acquisition or usage for future developers.

- In order to make a successful outcome of the project as likely as possible, a well-defined and formal project plan and development process shall be used.

2.3 Abstract System Structure

As stated in chapter 2.2 one goal for the project is the creation of system structure with a formal and clean development process. One aspect of this approach is the creation of an abstract system structure before any considerations towards the platform or the implementation details have been made. The sole basis for the system structure shown in this chapter are the requirements listed in the chapters 2.1 and 2.2.
Figure 2.2 describes the high level structure of the system. The design should be self explanatory. It implements the classical software paradigm of a distinction between three main system parts: user interface, application logic and data-storage. Note that all platform specific calls and interaction are abstracted through the data-storage module. The application-logic module doesn't need any information about the platform itself. In figure

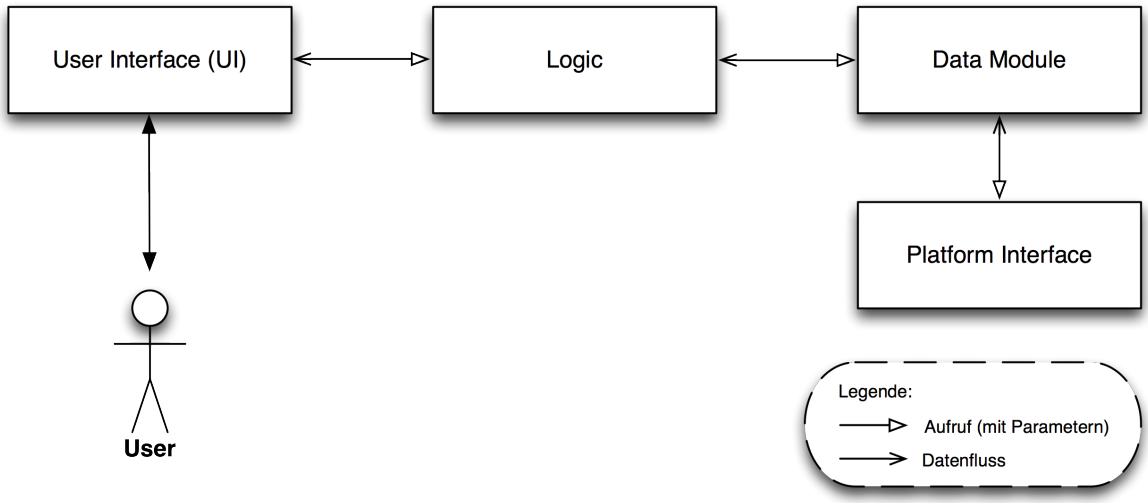

Figure 2.2: High level design of the system components

2.3 the inner structure of the application-logic is described. The sub-systems shown here are:

- Event Stuff: This module contains the main functionality of the software. It is described in details in figure 2.4.

- Event Stuff Tools: This is a collection of helper-functions that solve common tasks for the Event Stuff module.

- Routing: The routing engine and map-data management are located in this module, separated from the Event Stuff to reduce complexity and to impose the necessity to define clean interface between these system-parts

- UI Manager: This as an Abstraction of the concrete implementation of the user interface. The Event Stuff module tells the UI-Manager about events that occur and the UI Manager decides wether and how the user is to be made aware of the situation. With this abstraction the application can look and behave in completely different ways depending an the runtime-configuration of the UI Manager.

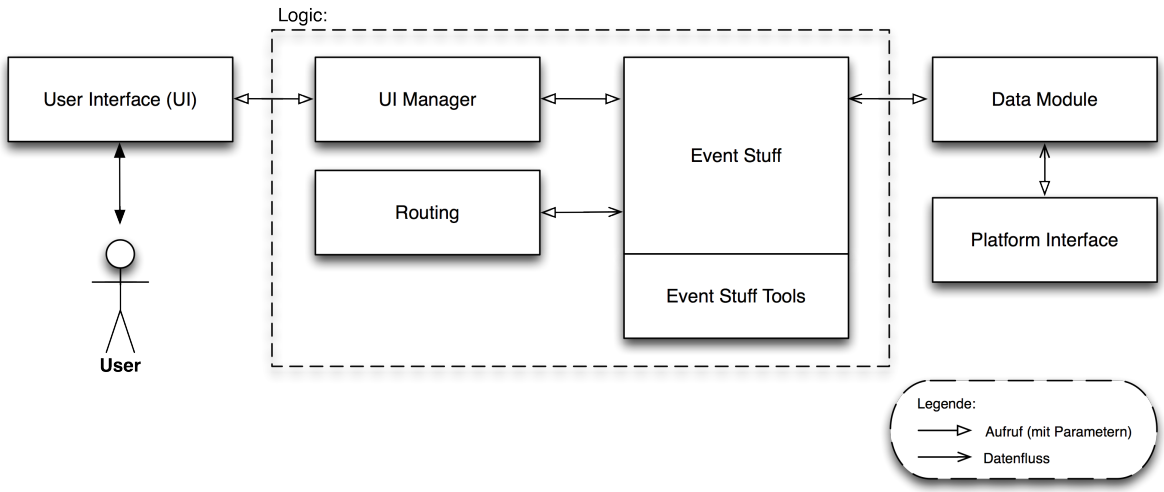

Figure 2.3: Representation of the logical system elements

Figure 2.4 shows the inner elements of the Event Stuff module. The functionality is divided into these modules:

- Event Warner: This module needs to run in the background it checks wether the current dayplan can be sustained, given the current position of the user in time and space. It informs the user when he hast to get going in order to meet his next appointment.
- Plan Bauer: This module is used to create dayplans from the events and the tasks in the database.
- Plan Optimierer: This module can optimize the current dayplan by inserting executable tasks into the dayplan at practical locations.
- Termin Vorschkaeger: This module can be used to suggest practical locations and times for meeting between two users of this application. This can only work, when the dayplans of these two users are shared.
- Erreichbarkeitspruefer: This module is part of the Event Stuff Tools. It can check, wether a given appointment can be reached, given a certain current point in time and space.
- Time-Finder and Time-Filler: These are helper modules that can suggest a place and a time for a certain task to become an event of a certain deyplan.

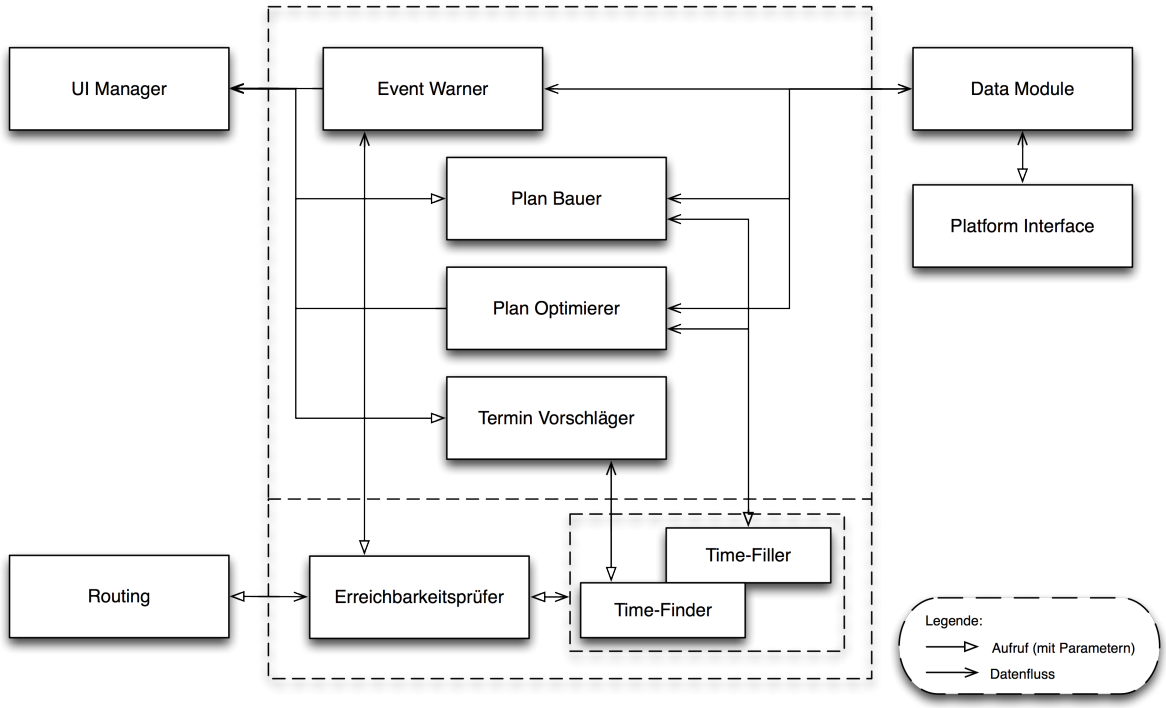

Figure 2.4: Components of the Appointment/Task optimization and verification engine

3 Project plan and development process

3.1 Initial project plan

Figure 3.1 shows a GANTT diagram that describes the progress plan for this project, as it was defined at the beginning of the project. The analysis of requirements, abstract system design and a realization analysis are important parts of the project, as about half of the project time is dedicated to these tasks. The documentation of both the development process and the resulting software is an iterative process that is performed simultaneous to the other activities.

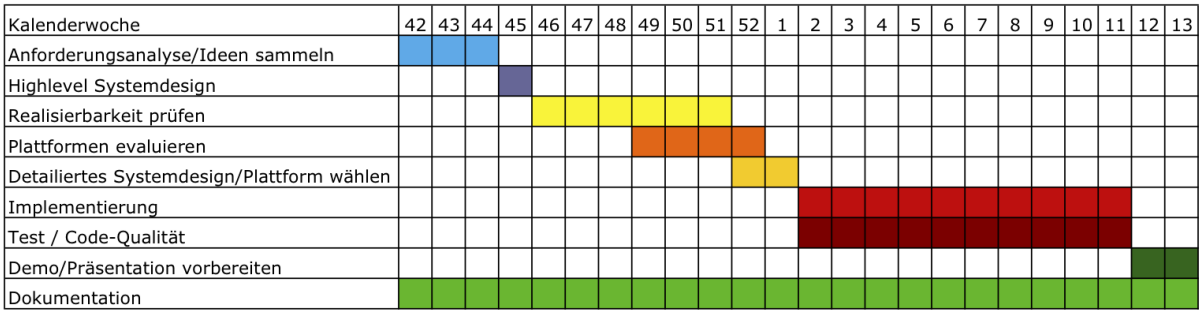

Figure 3.1: Gantt diagram of the initial project plan (planing at project start)

3.2 Refined project plan

During the development process it became obvious, that the initial project plan does not exactly reflect the amount of work necessary in the different work packages. Also milestones were introduced as a means to measure success in finer granularity than once at the end of every phase. Figure 3.2 shows the project plan as it was actually carried out. The key differences between the two plans are:

- longer platform evaluation and realization evaluation phase
- additional platform specific realization evaluation phase
- longer overall development time, especially significantly longer implementation phase
- break with the continuous documentation paradigm
- introduction of milestones for the implementation phase

The following Milestones were introduces for the development phase:

- derive a OS specific system structure
- install and get used to Android SDK

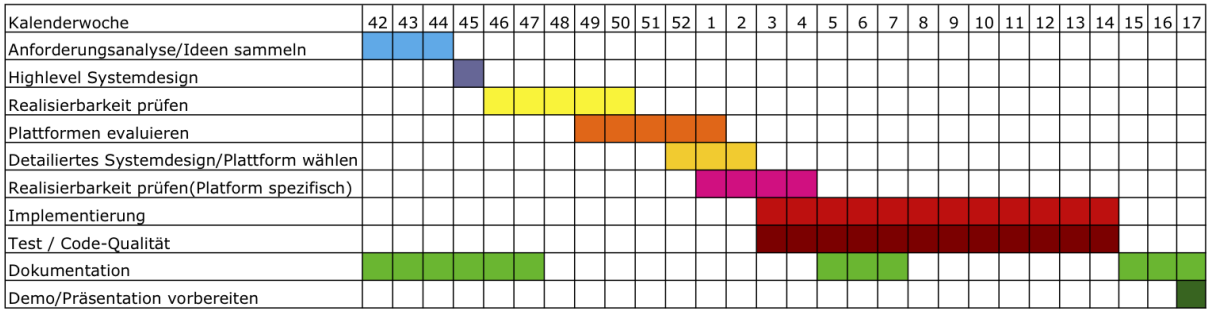

Figure 3.2: Gantt diagram of the actually carried out project plan

- port Traveling Salesman routing engine to Android
- solve biggest performance issues with routing engine
- reverse-engineer the non-public Android calendar interface
- finalize routing interface and implementation for Android
- finalize calendar API
- build system integration framework for Android OS, specifically location and time triggered background tasks and user interaction
- merge system parts into one unified application
- determine performance of routing engine, identify bottlenecks and problems for future fixing
- implement a structure to provide and manage dayplan information and checking/optimizing functionallity
- implement GUI for the task "Generate and Manage Tasks"
- implement GUI for the task "Build and Manage day-plan"
- implement background task "optimize day-plan"
- implement background task "check day-plan for reachability of appointments"
- write test-cases for the most important system parts (specifically the parts that could be reused in future projects)

The milestones are here not fixed to specific dates in the gantt diagram due to poor readability of the diagram. The order of the milestones reflects the order in which they were carried out.

3.3 Development process

In software development a process model is organizational framework that describes the way in which the software is developed. It defines different phases of the development, activities that are carried out during these phases, rules and criteria for advancing of phases and different roles for the team members. For this project an iterative waterfall-model

was chosen as development process. The regular waterfall model is defines as shown in Figure 3.3. The development phases shown in the figure are strictly executed one after the other, as each of them depend on the result of the previous ones.
In every iterative development-model, the desired functionality of the product is divided into smaller parts that can be implemented independently. In every development cycle new system elements are added, thus reducing the complexity of each cycle.
In the iterative waterfall model the simple waterfall model is used for each of these iterations.
The main advantages of this approach are:

- Reduced complexity of the individual development steps
- Continuous feasibility and success checking with the end of each iteration
- Development cycles of system parts with lose coupling can be carried out simultaneously

As code and revision management tool git is used, the repository is hosted at github: `http://github.com/mrtazz/kangaroo/`

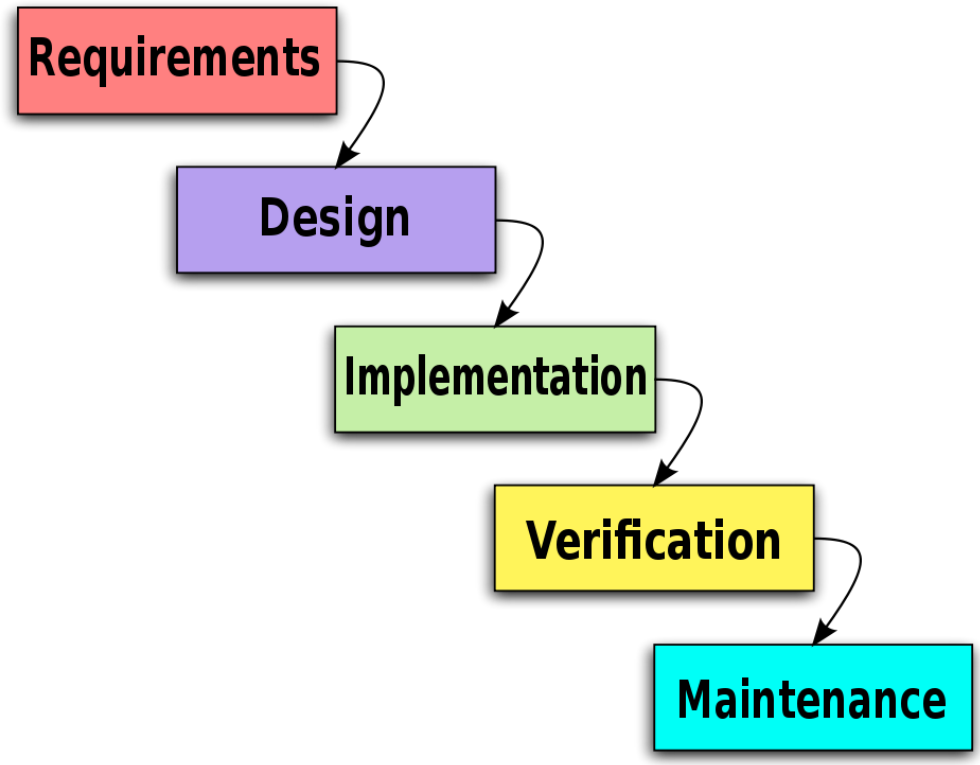

Figure 3.3: Generic waterfall model

4 Technology evaluation

4.1 Platform Choice

4.1.1 Introduction

The usage of a mobile device is implied by the nature of the planed software project. Also the wide availability of smart phones that come with all the hardware components we need makes it obvious that mobile phones are clearly the platform of choice. At the moment (Jan. 2010) there are three dominant operating systems for mobile phones that have to be considered for the present use case: Windows Mobile, iPhoneOS and Android. These platforms have to be evaluated in order to choose the one best suitable for our purpose.

4.1.2 iPhone OS

The iPhone is a very interesting device for mobile applications, since it has a very modern and intuitive interface. However due to some limitations in the iPhone OS, it is not suitable for mobile task routing and planning.

The main limitation of the iPhone OS is the fact that it does not allow applications to run in the background. This was a design decision to increase battery life. This is a big problem, since the main functionalities of the application rely on the fact that it is always running. For example, if the application isn't running all the time, it is not possible to warn the user about several situations in which he needs to act to fulfill his schedule.

In the iPhone OS there are two solutions possible to circumvent this limitation. The first one would be to have the application running all the time, which is both impracticle and would mean that nothing else can be done with the device during that. The second solution, which was integrated in the newest major release of the iPhone OS, is a notification system, which makes it possible to push messages, notification sounds and application badges onto the device. This however would mean, that the routing engine needs to be implemented on a server or other external device. Then we could get position updates from the iPhone, calculate if there was anything to be done and notify the user accordingly. However such a scenario would impose more threats and problems on personal privacy and is therefore not wanted. Additionally it is not completely sure, that the position data of the device, used for the "Find my iPhone" functionality, for example, is publicly available for developers.

That is why the iPhone is not suitable as a platform.

4.1.3 Android

TODO: describe why android is the most suitable paltform:

- inexpensive devices - open architecture: (background-tasks) - big momentum: large dev. community, adapted by nearly every phone manufacturer - high level language development, infrastructure for app distribution

4.1.4 Windows Mobile

TODO: why is win mobile not the right choice:
- no dev. momentum, (prior to win mobile 7) - clunky UI, not cool. (coolness is important factor for application acceptance) - .NET development: none of the developers has recent experience with .NET compact framework

4.1.5 Other OS

There are also several other mobile operating systems, which were not investigated further as a development platform. Main reasons for this are mainly:

- Small significance for the mobile market
- Complicated development of applications for platform
- UI rather bad to implement application

Mobile OS which are not considered are amongst others:

- Blackberry OS
- Symbian OS
- Maemo

TODO: Alex: check if this is complete

4.2 Routing

One essential feature of our application is its capability to not only consider calendar items for itself but to put them in their geographical context. The geographical context in the scope of this project is defined in a way so that questions like "Where am I?", "Where is the next bakery?", "How can I get there?" bear a meaning.

This definition implies the need for a set of data including Points Of Interest ("Where is the next bakery?") and a representation of local traffic facilities ("How can I get there?") to account for and an engine to operate on this set of data. This set of data will be called *routing data* from now on.

4.2.1 Routing data

From a very abstract and general perspective, this set of routing data can be divided into two categories, namely a dynamic and a static one. As static, in these terms, one should label all facilities which do not depend on any schedule and may in principle by used at any time (for example streets, highways, cash points). Dynamic facilities are the ones, that have a more or less fixed schedule constraining their usage (for example tramways, public bus service and shops).

Note that Points Of Interest cannot be attributed to one of these two categories across-the-board. A cash point for example will probably be a static facility, since it can be used 24 hours a day (apart from little exceptions). A bakery in contrast will have business

hours not covering 24 hours a day and thus will acutally be a dynamic facility.

In this project we will only consider static facilities, since also accounting for dynamic ones requires routing algorithms that go far beyond standard algorithms resulting in a more complex application structure and a much more complex routing engine. Besides this aspect, it is a big issue to gather scheduling data about dynamic facilities as there would be a number of carriers with different interfaces to include. This also applies to Points Of Interest, but which will *all* be treated as static facilities. However, the user might notably benefit from the abolishment of this restriction. It might be part of future work to include this feature.

The following section will give some information about potential sources of static routing data and its properties. This section may be skipped if one is either not interested in details or even already familiar with it.

Sources of static routing data

As potential sources of static routing data, one finds basically two options

- **Google Maps**

  Google Maps is an online navigation and routing service. Its map material has high quality but in some areas lacks actuality. Its main drawback may anyway be found in the absence of a free simple-to-use offline interface.

- **Openstreetmap**

  Openstreetmap is a free and open database for a world wide map. Everyone is free to add, change and remove map items. Consequently its map material varies over a hugh scale in quality, density and actuality. It can be used online and offline by downloading an Openstreetmap XML file of a rectangular map area. This file contains every map item of this area.

Besides the two options given above there are of course lots of other providers, but either not free of charge or not very popular. Consequently, Openstreetmap seems to be the one to choose for our project. The following will briefly outline its data scheme.

Data scheme of Openstreetmap

Openstreetmap uses three main entities to map traffic facilities, Points Of Interest and geographical characteristics:

- **Nodes**

  Nodes are the fundamental elements setting up model points for direct use or as a basis for superior map elements. A node imperatively has a unique ID and values for latitude and longitude, but can have optional parameters provided as tags with key-value-pairs. Adding tags to a node is the way to define, for example, Points Of Interest (POI).

- **Ways**

  Ways are made up of an ordered list of nodes which span the way in the given order. Just like a node, a way must have a unique[1] ID and may have optional tags.

[1] unique with respect to other ways; there still might be a node with the same ID

As a way can either be closed[2] or open and can have tags to specifiy parameters, it can be used to map streets, highways, roundabouts, buildings or even administrative areas.

- **Relations**

  Realtions provide a simple way to group other map elements. This feature will not be used in our project.

4.2.2 Routing engines

In our project's scope, the routing engine is ought to be used and thus its output is required to give answer to the following two questions:

1. Where, looking from a specfic location, is (are) the next Point(s) Of Interest providing a specific type of service?

   Finding a statisfying answer to that question is in principle very easy. One has to scan the local area around the given center for facilities that match the criteria and give a list of the nearest ones.

2. How long does it (approximately) take to follow the shortest path between two locations?

   To give a good answer to this question, one has not only to specify two locations as starting point and destination of the route, but also a vehicle that will be used to travel this route, since its choice might influence[3] the route. In addition to this, remember we are mainly interested in the *time* needed to travel the route, not its *distance*. For this, in a first step we have to combine the maximum speed of the vehicle with the distance of the route. As a secound step, one should also consider local speed limitations, which effectively influence the vehicle's maximum speed on a certain part of the route.

Having defined the information we are aiming for when triggering an operation of the routing engine, the following gives a brief overview of already existing open source routing engines that may be used in our project:

1. **Gosmore**

   Gosmore is a routing and navigation software for mobile devices. It consequently also consists of a very efficient routing engine to operate on data coming originally from Openstreetmap, but which is reorganized and compressed in a very smart way in advance. For this, data is ordered in a way that tries to minimize the probablity of a cache-miss and the number of comparisons needed when searching the local map for special map elements. Gosmore is based on C/C++, very compact and makes extensive use of pointers and indirect memory access. Documentation of its source code is poor and thus it is not easy to comprehend and retrace its code.

2. **OSMAndroid**

   OSMAndroid is a map viewer and routing engine for Android using precompiled Openstreetmap data. It uses a special OSMAndroidConverter to parse the bulky

[2] first and last nodes are the same

[3] think of oneways which normally are free to be used in both directions when for example riding a bicycle

Openstreetmap XML file and to compress it to an extremely small proprietary data structure, maintaining only map elements needed for routing. Its very slight data model results in little and fast data access operations and very efficient routing operations. OSMAndroid's source code has rare comments and follows a sometimes confusing design pattern, but is because of its compactness understandable with not to much effort.

3. **Traveling Salesman (TSM)**

   Traveling Salesman is a Java based open source navigation and routing framework using Openstreetmap data as its input[4]. It has a consistent modular architecture and allows to assemble a specific navigation software nearly "just-as-you-like". Traveling Salesman comes with ready-to-use implementations for every module. These modules are by default designed to keep every map element and not to filter irrelevant[5] ones. This results in a maximum generality at the cost of performance and memory space.

   One thing to note is that Traveling Salesman is up to now still in beta stage.

Since the target platform is a mobile one and resources will be limited, handling of available device resources is a crucial issue in the decision for a routing engine. Nevertheless it is not the only thing to think of in this context. In addition to the output requirements mentioned at the beginning of this chapter, there are other issues to account for. Some of them are for example the portability to our target platform and the flexibility for future adaptions and extensions.

When trying to assign priorities to these aspects just mentioned before, one will be faced with the question whether to aim for a maximum of flexibility or for a maximum of performance, since these two features are contradictory in many ways. Traveling Salesman is situated near the end of the scale where focus is set on a maximum of flexibility, while Gosmore and OSMAndroid can be found near the other end. Balancing these priorities was dynamically done during technology evaluation and therefore the next chapter (chapter 4.2.3) will give considerations and results on this aspect in greater detail.

Anyhow, as an integral part of the abstraction of the application's structure, the routing interface is designed in a way, that makes it independent of the decision for any routing engine, even if its design was influenced by it. This means, that although a final decision had to be made within the scope of our project, there is still the chance to choose another engine as part of future work.

Evaluation of routing engines

This section will give an overview of the general technology evaluation process aiming at the decision for a routing engine to be used in our project.

[4] actually it is using Osmosis as an interface to Openstreetmap data, which is a java application for processing Openstreetmap data

[5] irrelevant in the sense that the result of any operation that may be performed does not depend on the element's presence or absence

Before focusing on a single routing engine, we analyzed the given options of routing engines with respect to our definitions. Unfortunately when doing the technology evaluation, we did not know about the OSMAndroid routing engine and thus had no chance to consider it in the decision for a routing engine. Although its analysis has been done when a final decision already had been made and the technology evaluation phase had been finished, the reasons for us not to change this decision afterwards will be given in this chapter.

With the two options given at the beginning of the technology evaluation phase, Gosmore and Traveling Salesman, the Traveling Salesman project had the considerable advantage of being based on Java, in contrast to Gosmore, which is effectively a C/C++ library. This fact gave rise to put first focus on Traveling Salesman to identify its drawbacks and opportunities.

As our analysis revealed that Traveling Salesman is the most promising option[6], a discussion of this analysis will be given in a later chapter in greater detail. At this point, we will give an overview of the analysis of Gosmore, Traveling Salesman and OSMAndroid. In Addition to these three we will give a summary of considerations towards a proprietary development at the best combining advantages and opportunities of existing projects and excluding performance bottlenecks.

- **Gosmore**

  Experiments with Gosmore on a PC showed that besides a simple interface this software also has the advantage of superior performance over Traveling Salesman. Unfortunately the Android NDK (Native development kit) that would have allowed a compilation of C code for the Android platform was not available yet. Under these circumstances the only way to port Gosmore to the mobile platform of our choice would have been an direct integration into the underlying linux operating-system. That would have required root access, which poses a serious security risk and is not possible with off the shelf Android devices. In order to preserve the end user compatibility of the application, Gosmore can not be used.

- **Traveling Salesman (TSM)**

  First integration of Traveling Salesman's library into an Android project was rather simple, just as one would expect. It was integrated into a first application which was designed to use Traveling Salesman to read an Openstreetmap XML file stored on the SD card of the mobile device, output the file's summarized content and to perform simple operations on the routing data. Compiling and running this application was possible without any errors but revealed three initial problems, which will be described below.

  – Traveling Salesman uses a proprietary `FileLoader` class to read the Openstreetmap XML file and to initialize a `MemoryDataSet` object holding all data from the XML file. This `FileLoader` in turn uses the java built-in `SAXParser`[7] class and an extension of the `DefaultHandler`[8] class which seem to be implemented slightly different in Android's java library. A permutation in the parameters of the method `startElement(...)` yielded a complete discapa-

[6] remember the only alternative at that time was Gosmore
[7] `javax.xml.parsers.SAXParser`
[8] `org.xml.sax.helpers.DefaultHandler`

bility of reading the XML file, but could be overcome by re-permutating its parameters.

- After having fixed the problem given in the previous clause, Traveling Salesman did its first job on both the emulator and a physical mobile device. With minor additions to our application, we were able to perfom a `getNearestNode(...)` query on a tiny data set, showing that we will get into big trouble concerning performance, if no adaptions were applied. However, we were able to show that improving Traveling Salesman's performance in a way satisfying our requirements is in principle possible and an issue one will be able to deal with (chapter 5.6.2 will give details).
- As already mentioned, Traveling Salesman by default reads the complete map file into memory. As Android's heap size is strictly limited, this resulted in the necessity of using a very small map file. It also bared the strong need for a circumvention of this restriction.

- **proprietary development**

  The possibility to use the given routing engines only as a source of inspiration and to start a proprietary development began to cross our minds, when ideas of how to design a routing engine became clear. Deliberation of combining a slight and efficient data storage system (like Gosmore is using), a reduced routing graph to operate on and java based source code (like Traveling Salesman has) occupied our minds. This thoughts were triggered by becoming aware of the fact that Traveling Salesman is in principle much more than we need and Gosmore in turn would need too much manipulations to be compatible with the Android system.

- **OSMAndroid**

  We became aware of the existence of OSMAndroid when we already had made a decision for a routing engine. Nethertheless it would be unjustifiable not to dare a look at its features and chances.

  OSMAndroid turned out to have nearly exactly the features we planned to include in a proprietary development. However there are some issues breaking the deal. OSMAndroid for example does not include all map elements or every property of map elements which are needed by our application. This for example applies to the maximum speed on a street or Points Of Interest at a whole. As it uses a highly proprietary data structure, it seemed to be impractical to adapt and use this routing engine in our project.

Decision for Traveling Salesman

Eventually we decided to base the routing engine on the Traveling Salesman project, maintaining compatibility to its modular architecture. This decision was forced by the following reasons

- Java based source code

  Traveling Salesman is completely based on Java and does not use any external libraries. Its precompiled project library can easily be integrated in the Android environment and Android projects following the standard design approach and thus requiring zero investment in any abnormality with respect to this standard.

- adaptions and extensions may be implemented without touching its inner structure

  Traveling Salesman's modular architecture supports easy exchange of any of its components. Given the imperative need of an adapted data storage management system to improve utilization of resources, we could easily come up with a re-design of this component without the need of adapting other components. The same applies to modifications of both the routing algorithm and its routing metric which can each be modified independently of the other.

- maximum generality and flexibilty, exchangeable components

  Keeping compatibility to Traveling Salesman's modular architecture will offer the chance for easy maintenance or even upgrades. One may consequently benefit from future modifications made to the original Traveling Salesman project.

- performance improvements

  Early progress made with improving its performance gave rise to the assumption, that the advantages given in the clauses before will doubtlessly outweight the loss of performance and advantages provided by other solutions.

- "flexibility before performance"

  Last but not least, our project is thought to be the basis of more extensive projects. In order to prevent future developers from failing because of prematurely introduced constraints, we will assign a higher priority to flexibility than to performance.

4.2.3 Traveling Salesman

The following chapter will give a more detailed introduction in the structure of Traveling Salesman.

As already mentioned, Traveling Salesman is designed following a modular architecture. Every essential or optional type of module is defined by a java interface with basic methods to provide its desired functionality. The most important ones in our project are

- Interface `IDataSet`

  `IDataSet` provides methods to store and receive Openstreetmap map elements. Identification of elements to receive can either be done by unique Openstreetmap ID or by parameters to search for. This, for example, may be the name of the element or simply its distance to a specific point, that has to fall below a cetain value.

  In dependence of its implementation, data storage is done in memory (`MemoryDataSet`), on disc or in a database.

- Interface `IRouter`

  `IRouter` is the interface which has to be implemented by any routing class. Its primary task is triggered by calling the `route(...)` method, which performs routing between a point to start from an one or more destination points. One has to pass an `IDataSet` object specifying routing resources, when calling `route(...)`. The router is to return a `Route` object containing detailed information about the path.

Traveling Salesman provides several implementations of the `IRouter` interface using different routing algorithms. The `MultiTargetDijkstraRouter` can be considered as the default implementation. As implied by its name, it is based on the Dijkstra algorithm which is extended to an A*-algorithm by using the linear distance to the target node as a heuristic. Its implementation, however, seem to be faulty (see chapter 5.6.2 for details).

- Interface `IRoutingMetric`

  As to give the router a measure of routing costs, one has to define a class implementing the `IRoutingMetric` interface, which allows the router to query routing costs of particular routing steps.

  The default implementation (`ShortestRouteMetric`) simply measures distance between start and end of a routing step by summing up the linear distances between nodes on the way. One might feel uncomfortable with this, because it does not privilege any type of street and hence the shortest route is probably not the fastest one. However, this can be overcome with a re-implementation of a routing metric.

To represent Openstreetmap map elements (nodes, ways and relations), Traveling Salesman uses classes provided by Osmosis, which can be considered as a Java based framework for processing of Openstreetmap data. Osmosis employs classes named `Node`, `Way` and `Relation` to represent Openstreetmap map elements accordingly.

5 Developement for Android

5.1 Platform Specific System Design

Based on the abstract system structure as presented in chapter 2.3 a platform specific software structure needs to be defined for the Android OS platform.This structure is shown in figure 5.7. It respects Android specific constructs, patterns and object-types.

The classes marked in orange extend the platform specific Activity-class. In Android

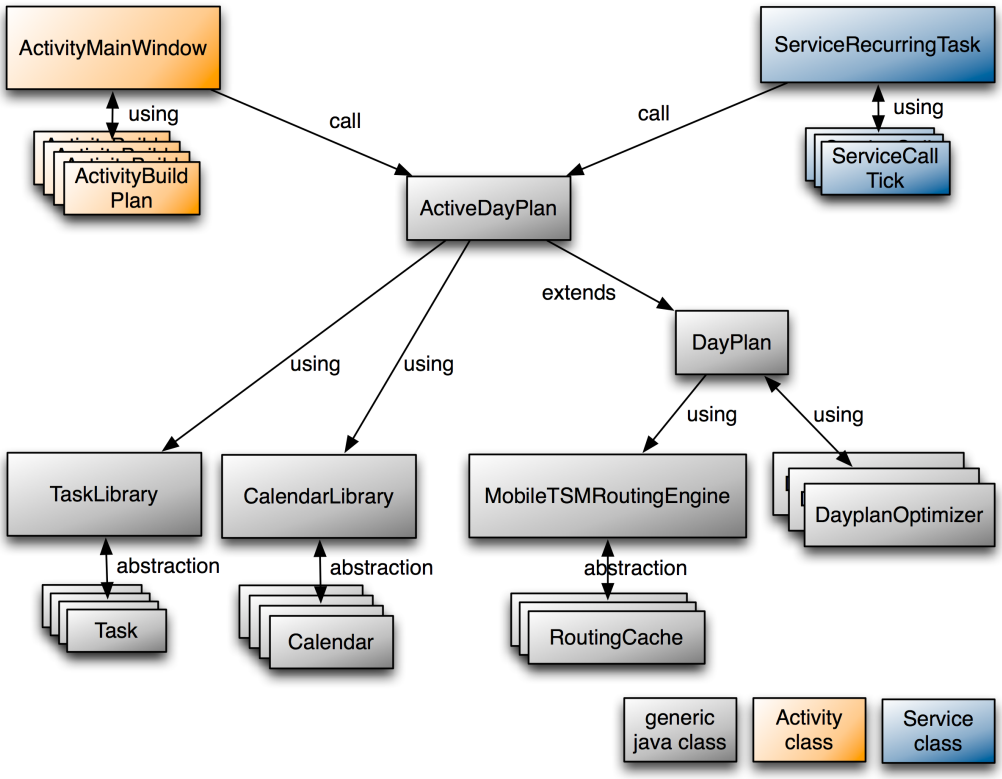

Figure 5.1: Platform specific software structure for Android OS

OS, Activities are used to build the graphical user interface (GUI). The class Activity-MainWindow contains a tabbing element, the other Activities from the com.kangaroo.gui package are wrapped in the individual tabs. A detailed description of the Kangaroo GUI can be found in chapter 5.2

The blue classes extend the Service-class. In Android OS Services are used to execute computations that are independent of the user interface. The ServiceRecurringTask class in com.kangaroo.system implements a background task that is executed over and over independently of user interaction. The other Services shown here are helper classes that

make sure the background task is called when the location of the device in time or space has changed. These system integration classes are explained in the chapter 5.3

All classes kept in grey are generic java classes. These system elements do not directly extend the Android platform, although some of them might use platform specific data types and methods. The classes can be divided into the following groups:

- TaskLibrary: The TaskLibrary and other related classes in the package com.kangaroo.task provide the functionality to store and manage tasks. These classes are described in chapter 5.5
- CalendarLibrary: The classes in com.kangaroo.calendar provide a wrapper for the non public Android calendar API. A description of the classes and the reverse-engineering process can be found in chapter 5.4.
- MobileTSMRoutingEngine and the other classes in the com.mobiletsm.* package provide routing functionality for Kangaroo. The routing is described in chapter 5.6
- DayPlan: The DayPlan and related classes in com.kanaroo implement temporal storage of events and tasks. Also these classes provide the functionality for checking and optimizing of the day plans. These classes can be found in chapter 5.7.1

5.2 User Interface

TODO: insert screen-shots

Basic Interface Structure

The Android user interface is built upon an XML templating engine. The basic interface is designed via XML templates and later on adapted and filled with content via Java methods. Every Activity class inherits from a basic activity class like “TabActivity” or “ListActivity”. The XML templates define a basic element of the according class also (“TabHost”, “ListView”).

Main Window

The main window is just the skeleton with the tablist, which then contains the active window itself. The Activities are added via Intents into the tabbar. An important point, when entering the embedded Activities into the application’s manifest file, is to add the intent filter

```
<category android:name="android.intent.category.EMBED"></category>
```

also. This makes sure, that the Activity respects the boundaries of the main window when activated.

Dayplan

The DayPlan Activity is the basic status interface which shows the current dayplan in a list. It also contains a context menu for manipulating events and an application menu to reload events and invoke the optimizer. The templating for this Activity consists of a template for the ListView and a separate template for the single rows.

In order create the view, the list of events is fetched from the ActiveDayPlan. After that a custom ArrayAdapter is used to populate the single rows of the ListView. The only slight problem is to get the correct object from the ArrayList, when tapping a row in the Activity. Fortunately, when the onCreateContextMenu method is called, it is passed a ContextMenuInfo object as a parameter. This object contains an id, which is the same as the position in the original ArrayList from which the view was populated. Therefore it can be used as an index to identify the correct object in the list. Figure 5.2 shows an example screen.

Figure 5.2: Basic DayPlan with calendar events only

Tasklist

The Tasklist Activity displays all the available tasks. It is a two level list view, which shows the task title in the first level and the task details on expansion of an entry. It also contains a context menu to manipulate single tasks and an application menu to add new tasks. The templating for this Activity is more advanced, as there has to be a template for the ListView itself, the first level and second level of rows.

The TaskList was slightly more difficult to implement as it is a two-level ExpandableListView and therefore naturally more complicated. The most important part is building the entries, as the ListAdapter expects a List of Maps of Strings to Strings for the first level entries (the task titles) and a List of List of Maps of Strings to Strings for the second level entries. The constructor for the adapter (SimpleExpandableListAdapter) then takes these data structures along with String arrays matching the keys of the passed Maps and Int arrays containing the IDs of the elements in the respective XML template for the first or second rows.

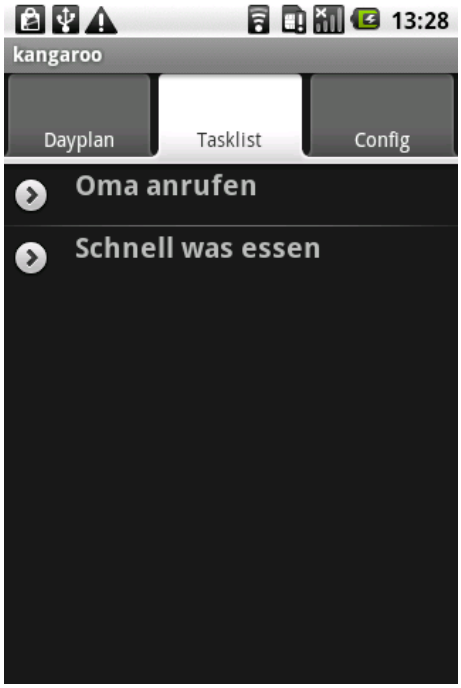

Figure 5.3:

Edit Task

The EditTask Activity (figure 5.4) displays information (title, description, constraints) of a task object in various edit fields. Here the data can be changed and is saved on every keystroke. Therefore the data is available when the Activity is left.

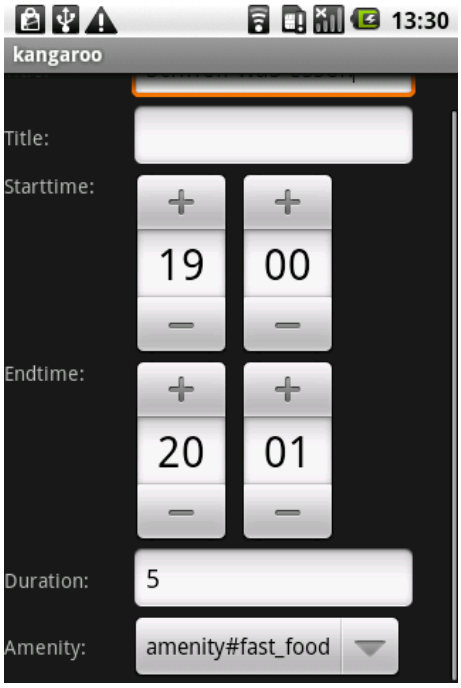

Figure 5.4: Basic interface for editing tasks

Configuration Editor

The Configuration Activity (figure 5.5) provides a way to preset several application settings, such as calendar name to use and time interval for background services to do checking. The values are read from and saved to the key-value store of the kangaroo application. When the application is initially opened, default values are used.

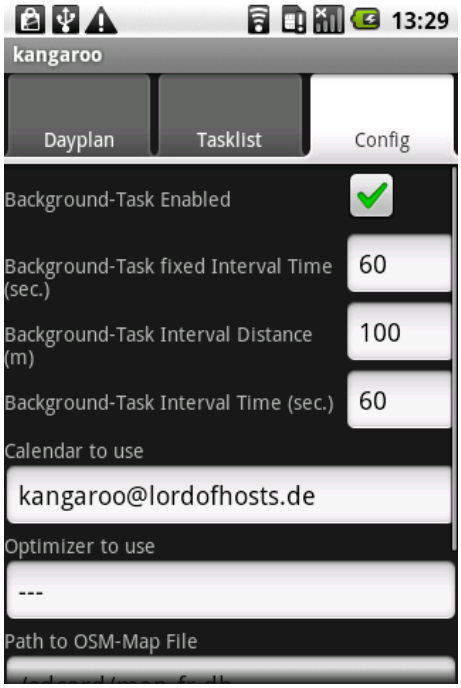

Figure 5.5: Configuration editor

Day Optimizer

The DayOptimizer Activity (figure 5.6) displays a possible dayplan, optimized via various characteristics. The user then has the choice whether to accept the new dayplan, generate a new one or abort. If the dayplan is accepted, it is set as the current dayplan for the application.

Activity Options Menus

Activity Options Menus are often used to be able to reload an Activity or add a new element to it. The menu also needs an own XML template, which is inflated on menu creation. After that, a logic has to be added for every ID (button) of the menu in the method "onOptionsItemSelected".

Context Menus

Several context menus are implemented in the various Activities to edit or delete single objects or provide more information. Context Menus don't have an XML template for itself, since they are all based on the same layout. Only the special method "OnCreateContextMenu" is needed in the Activity as well as the registration of the Context Menu via "registerForContextMenu".

Figure 5.6: DayPlan optimizer dialog

Notifications and alerts

The class UserNotification provides an abstraction of the Android user notifications. The notifications are shown with Kangaroo logo and short message in the status-bar at the top of the screen (see figure **??**). When the user pulls the notification bar down, a longer text is shown, as shown in figure 5.8. Upon a click on the text the Kangaroo main GUI is started. The class provides a simple interface. Other parts of the software can initiate a

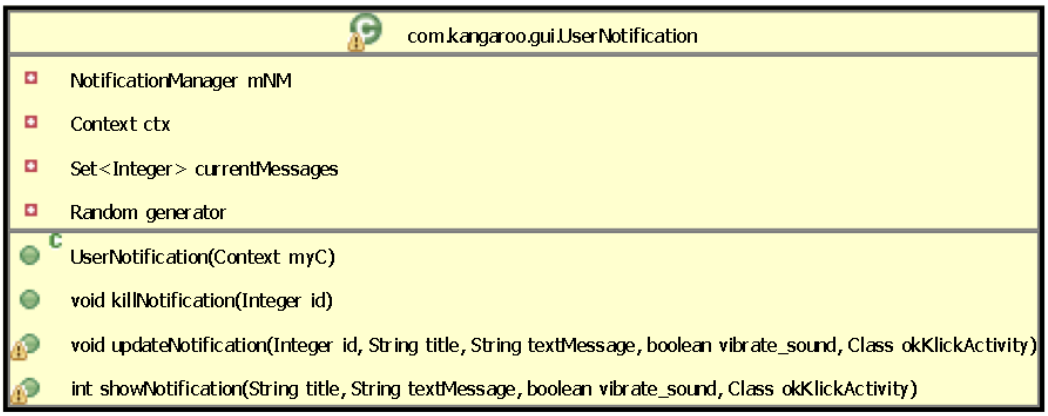

Figure 5.7: Class diagram of com.kangaroo.gui.UserNotification

new notification with the showNotification() method, and are returned an interger value that represents a unique identifier for this notification. The caller can decide wether the user should be alerted with a sound and vibrator alert. With the ID the notification can be deleted or updated with new content.

Figure 5.8: Notifications with longer text

Figure 5.9: Menubar notifications

5.3 Background tasks

TODO: graphic that shows all background-services

TODO: describe graphic, explain serviceCall*, serviceRecurringTask and the Brouadcast-listener

TODO: describe efforts to make the service run un-noticable in the background: semaphore, CPU-wake lock

5.4 Calendar Integration

All data integration on the Android plattform is done via so called "content providers". These providers are the only way to transfer data between applications, since there is no shared space which all applications could use.

However the content provider for the calendar data is working, but not specified in the official developers documentation. That means that the API could change at any time, which has to be considered at the application design.

In order to not be too dependent on the changing API, a wrapper layer was built around it. Now there are only four methods which have to be adapted if the API changes.

- queryEvents
- insertEventToBackend
- updateEventInBackend
- deleteEventFromBackend

The "queryEvents" method can be given a SQL statement as a selection statement and a String array. Values in the array subsequently replace all the "?" in the statement. The method then returns an ArrayList of CalendarEvent objects, which suited the selection.

The "insertEventToBackend" method is used to insert new events via the content provider. It takes a CalendarEvent object, builds an Android ContentValues object from it and inserts this into the backend.

The "updateEventInBackend" method does practically the same, but updates the events in the backend, instead.

At last the "deleteEventFromBackend" method removes the passed CalendarEvent completely from the backend.

These four methods, described above, are the sole fundament of the calendar wrapper library. As they mostly consist of less than ten lines of code, the adaption to a new API can be done extremely easily.

5.5 Introduction of Task management

TODO: android does not have a task handling/storage

TODO: introduce our task class, explain why constraint-architecture was necessary

TODO: discuss task serialization

TODO: storage in calendar, possible different solutions

5.6 Routing

5.6.1 Routing interface (Kangaroo routing framework)

The interface between the main software core and the routing engine has to be designed independently of any specific routing engine, maintaining an abstraction level that allows to exchange the routing backend easily without any impact on the main software part. We will give an overview of the routing interface, which was designed to be compliant with this

requirements. The routing interface as a whole will be called *Kangaroo routing framework*.

The Kangaroo routing framework is designed independently from Android and only uses Standard Java compliant classes. Thus it is qualified to be used outside of the Android system. This was introduced to enable easy debugging without being confined by the Android SDK.

Structure of the Kangaroo routing framework

There is one Java interface and three Java classes that can be considered as the main members of the Kangaroo routing framework.

- Interface `RoutingEngine`

  The `RoutingEngine` interface is the main interface between the routing engine and the module that is using its routing service. It provides methods to find a route between locations and to find street nodes and Points Of Interest near a location. It also provides methods to control the routing cache[1] which is intended to be included in every routing engine compatible with the Kangaroo routing engine.

- Class `Place`

  This class is an abstract representation of a geographic location. It imperatively consists of a pair of geographical coordinates (latitude and longitude) and may in addition contain a name or a reference to Openstreetmap elements associated with it.

- Class `RouteParameter`

  An object of this class is returned by the routing engine when having searched the map for a route between two locations. It has methods to return the length of the route and the time it will take to travel the route. Additionally, it may contain an object representing the route itself.

- Class `GeoConstraints`

  This class is used to define geographical constraints for searches on the map. This provides an interesting feature if for example one is looking for the nearest Point Of Interest of a special type not simply measured from any given location but also accounting for a location that has to be visited after this Point Of Interest.

5.6.2 Routing engine (MobileTSM)

As we already showed, there was no way to get around at least adapting an existing routing engine. As Traveling Salesman is the one to use as a basis, our version of Traveling Salesman implementing the routing interface defined within the Kangaroo routing Framework is called *MobileTSM*. The folling will summarize its structure, explain the most important changes in contrast to the original version of Traveling Salesman and give reasons for the adaptions.

[1] the routing cache will be described in detail in chapter 5.6.2

Major problems arising with decision for Traveling Salesman

Coming up with the decision for using Traveling Salesman, there were several issues to work on:

1. **Data storage**

   The data storage modules (data sets in Traveling Salesman's terminology) that come with the project can be considered unoptimized or even bad with respect to efficency and performance as needed for a mobile application. There are several different types of data set modules with different approaches of storing and organizing data.

   Generally, but on Android's mobile platform in particular, there are two contradictory goals for the data storage architecture. Since Android restricts the heap space for an application to (at this time) 16 megabytes, the map area that can be cached in memory, is limited to several square kilometers, depending of course on data density. Thus one is forced to filter map elements before loading into memory. As not only memory is expensive, but also data access to a flash card or a database, a compromise has to be found between loading the whole map into memory and accessing the persistent map database for every single element requested by the routing engine.

2. **Routing data**

   By default, Traveling Salesman performs every routing operation on the original set of data not dropping elements that do not affect the result of the operation. An example for elements of this type are nodes, that geographically shape the curvature of a street but do not introduce a junction. A node of this type will be called *intermediate way node* and is dispensable for routing operations unless it is the nearest node to the starting or destination point of a route.

   Both memory space occupied by the map and time consumption of a routing operation can be reduced by ignoring intermediate way nodes. Nethertheless one cannot dispense with intermediate way nodes close to starting and destination point. This implies an individual routing data set for every routing operation. At first sight this induviduality runs contrary to Traveling Salesman's architecture when aiming at keeping compatibility. We will show a way to individualize routing data while maintaining full compatibility.

3. **Searching algorithm**

   When given two pairs of geographical coordinates and an order to find a route between these two, one has to find the nearest street nodes to starting and destination locations prior to the actual routing operation. This is because non-trivial routing operations can only be performed on a routing graph between vertices. Thus one has to translate from a geographical location to the vertex (node) that best matches it. This can be considered the street node with minimal distance to the given location.

   A naive ansatz to finding the nearest street node to a geoprahical location would be to iterate over all street nodes, calculate the distance between the location and each street node and select the one with minimal distance. Exactly this is done by Traveling Salesman resulting in an extremly time consuming routing process.

Approaches for major problems arising with decision for Traveling Salesman

As just explained in the previous section, three main issues have to be handled: reduce and individualize routing data while maintaining compatibility to Traveling Salesman, find an efficient data storage architecture and improve the search algorithm for nearest nodes. The next sections will outline the abstract framework designed to approach these problems.

1. **Reducing and individualizing of routing data**

   The basic idea is to introduce a module, an abstract Java class named `MobileDataSetProvider`, that can be queried for a specialized set of routing data which is compatible with Traveling Salesman and optimized for an individual routing operation. Any Traveling Salesman router may then be called with a reference to this individual data set. Anyhow, a router can perform another routing operation with different parameters on the same data set, but since the majority of intermediate way nodes will not be included in the data set, it will probably not find the best route or might even fail.

   One can also think of adding functionality to recycle an individualized data set. Instead of creating a new one for each routing operation, an old data set may be updated to be as qualified as if it was created specifically for the new parameters. One may be tempted to believe this is the only way to significantly gain time resources, but we will show that in fact our implementation exploits major improvements without this feature.

   When dropping intermediate way nodes one has to be aware that Traveling Salesman calculates the distance of a route by hopping from node to node summing up their linear distances. To revise the error induced by missing intermediate way nodes, an extra property has to be added to ways giving the real distance between two way nodes. That is the main reason for introducing a derivate of Osmosis' `Way` class, called `MobileWay` and a derivate of Traveling Salesman's `WayNode` class, called `MobileWayNode`

2. **Data storage architecture**

   The data set provider itself reads data from a suitable data source (for example from database or a file stream). Since even individualized data sets will have a large intersection, data is cached in memory to a great extent to avoid reloading of data already read due to a prior routing query.

   As an abstraction of the data access to a routing data sources, a Java interface (called `RoutingDataAdapter`) is introduced. It provides methods to read map elements from a given data source.

   To become more tangible, the abstract `RoutingDBAdapter` class as a derivation of the `RoutingDataAdapter` class will be used to access a database as source for routing data. A further derivation of the `RoutingDBAdapter` class, called `RoutingAndroidSQLiteAdapter`, finally implements the access to the Android specific SQLite database. For a description of this database structure and its creation, look at section 5.6.3

With this type of abstraction it is still possible to exchange the SQLite database by another type of database or even not to use any database system at all but for example a proprietary file format. Nevertheless the SQLite database of Android seemed to be the most promising one, since one can expect of it to be highly optimized while maintaining a great degree of flexibility. In deed, any of our tests gave reason to be in doubt about this expectation.

3. **Search algorithm for nearest nodes**

   As already demonstrated, the search for the nearest street node is an integral part of a routing operation. Without any assumption about the chronological and geographical pattern of routing operations that will be performed, one has to improve the data structure storing the nodes to be qualified for fast random access. Anyhow, one can think of another way to speed up the search when analyzing the average use.

   Besides routing operations performed in advance[2], the routing engine will also be triggered with routing operations using the current position. Since the current position is likely to change slowly in contrast to a random access, performance may be improved by caching street nodes inside a given radius around the last query.

4. **Caching of calculated routes**

   Routing operations may and will in principle have random parameters based on calendar items and the current position. Anyhow, it is highly probable that antecedent routing operations recur after a while. If one can make sure that the operation's parameter exactly match, there is no sense to recalculate the route[3]. It may rather reuse the result of the respecting routing operation.

   MobileTSM introduces a routing cache that efficiently stores results of antecedent routing operations by using a kind of double hashing. When triggering a new routing operation, the routing cache will be checked in advance and only if there is no matching result of an antecedent routing operation, the routing engine will actually perform the routing operation.

Structure of MobileTSM

- Class `MobileTSMRoutingEngine`

  The class `MobileRoutingEngine` is the *MobileTSM* implementation of the `RoutingEngine` introduced with the Kangaroo routing framework. It provides the routing services used by the main application, including the previously explained routing cache.

- Class `Vehicle`

  The abstract class Vehicle implements Traveling Salesman's `IVehicle` interface which is thought to define and constrain the usage of streets and ways. These constraints will be considered by the routing engine when being passed as a parameter of a routing operation. It also adds the possibility to specify a maximum speed. The routing engine will account for this maximum speed when calculation the duration of travel of a route.

[2] routing operations that are only based on calendar items and that are independent from current position
[3] Note that this assumes restriction to static routing data

There a several derivates of this class defining different types of vehicles, including appropriate maximum speeds (for example `AllStreetVehicle`).

- Class `MobileTSMRouteParameter`

  This class is a direct derivate of the abstract `RouteParameter` class. It includes the algorithm to calculate the duration of travel of a Traveling Salesman route based upon the route and the used vehicle.

- Class `POICode`

  This class is used to represent one type of a Point Of Interest. It also provides a mapping between possible tags of Openstreetmap and an unique integer value.

- Classes `MobileNode`, `MobileWay`, `MobileWayNode`

  The need and sense of introducing the derivated classes `MobileWay` and `MobileWayNode` was already pointed out and won't be repeated. The derivated class `MobileNode` is introduced to be able to associate a reference to a nearest street node and a Point Of Interest (represented by an instance of `POICode`) to the node.

- Class `MobileMultiTargetDijkstraRouter`

  As already stated in chapter 4.2.3, Traveling Salesman's router class `MultiTargetDijkstraRouter` seems to be implemented incorrectly. It actually is supposed to be a Dijkstra algorithm extended to an A*-algorithm. This extension is characterized by the use of a heuristic for determining the next node to visit. In Traveling Salesman's original implementation, this heuristic and the data structure holding the nodes to visit is not realized correctly, often yielding not the shortest path as a result. These implementation errors are fixed in the `MobileMultiTargetDijkstraRouter` router class.

- Class `MobileRoutingMetric`

  As explained above, dropping intermediate way nodes results in the need for an adapted measure of distances on a way. Since the routing metric (`IRoutingMetric`) is a self-contained module, it can easily exchanged by a metric (`MobileRoutingMetric`) using the corrected distance measure.

  Like Traveling Salesman's default implementation `ShortestRouteMetric` also the new implementation does not privilege any type of street. The undesirable effect of finding the shortest but not definitely the fastest route still persists and may be abolished as part of a future extension.

Tests and benchmarks for MobileTSM

In order to test the performance and reliability of the MobileTSM routing engine, we definied a simple test case that tries to cover all routing tasks the routing engine may be faced with when being in "all-day" use.

Consider the following calendar items

1. an appointment fixed at a specific position in time and space (latitude/longitude)
2. a number of variable tasks each based on a special type of amenity (e.g. post box, cash point)

Starting at a point that is in time and space far from the appointment in (1), we make some movements in space which are not necessarily straight into the appointments direction, while time is increasing linearly. At a certain point, the movement turns into a straight movement towards the appointments meeting point. Note that for the movement a single type of vehicle is assumed and it is restricted to an area in space, that is completely covered by the given map file.

Given this movement, the routing engine will be used to periodically perform a set of tasks to give answers to the following questions

1. Is there any need to start moving straight to the next appointment? Is there even no way to get there in time anymore?
   Starting from the current position in space, the fastest route to the next[4] appointments position in space is calculated. The routing engine will account for the restrictions that are given by the specified vehicle. The time needed to follow this route with the given vehicle is compared to the time that is left until the next appointments meeting time.

2. Is there any way to handle one or more variable task between the current position in time and space and the one given by the appointment in (1) without running the risk of missing this appointment?
   A variable task is selected. Starting from the current position in space, the map is searched for the nearest amenities needed to fullfil the selected task. Given this list of amenities, every[5] single amenity is selected and the following calculations are performed: the fastest route from the current position in space to the selected amenity and the fastest route from this amenity to the next appointments position in space is calculated. Using the sum of the time needed to follow these consecutive routes with the given vehicle and the time needed to fullfil the task is compared to the time that is left until the next appointments meeting time.

In order to ensure a reproducible test case, there are some parameters to fix. These are

- the map file that containing routing data
- the vehicle that is used and hence the infratructure that can be used to get to the destinations
- the position in space and time of the appointment in (1), namely time, latitude and longitude
- the number and type of variable tasks (including the type of amenity they are based on)
- the parameters of the movement previous to the appointment in (1), particularly the starting point in time and space of the movement

5.6.3 Routing database

Android comes with an ready-to-use implementation of a SQLite database system which is the one to use in our project to store routing data (cf. chapter 5.6.2). It seems reasonable

[4] the one which is in time the next looking from the current time
[5] a more or less intelligent filter should be applied to reduce the list to a minimal number of amenities

to include all routing data in one database file and to preprocess it in a way that keeps operation expense on the mobile device at a minimum. The follow sections will give details on the database structure and the tools used to convert an Openstreetmap XML to such a database file, including the just mentioned preprocessing.

Significant map elements for routing data

The Openstreetmap XML file consists of a set of node, a set of ways and a (for this project dispensable) set of relations. Although in principle every node and way has the same "weight", significance for our application may vary over a huge scale. This significance depends on the attributes of the according map element. Three types of map elements are of high significance:

- selected ways

  As Openstreetmap also uses ways to map buildings or areas of a special type, public stairways or almost impassable trails, one is forced to drop every way that cannot be considered to be a "standard street"[6]. This selection is based on the tags associated with a way.

- selected street nodes

  The nodes spanning up a way that wasn't dropped are considered to be significant to our application.

  Furthermore, street nodes are (as already mentioned several times) split into two categories. *Intermediate way nodes* may be dropped without any impact on a routing operation traversing the way, whereas *essential street nodes* cannot (cf. chapter 5.6.2 for details).

- selected POI[7] nodes

  Nodes that are tagged as an "amenity" or a "shop" of any type are considered to be significant to our application.

Note that dropping map elements needs to be done with care to yield a coherent reduced map, since one has to be aware of reciprocal dependencies between map elements.

Structure of routing database

According to the structural demands the database basically consists of four tables:

- `ways_0`

  Every row in this table represents a way.

[6] we assume the user is not willing to travel such almost impassable trails as a pedestrian
[7] Point Of Interest

column	data type	description
id	integer primary key	unique way id (= OSM id)
name	text not null	name of this way (= OSM 'name' tag)
highway	text not null	= OSM 'highway' tag
maxspeed	integer not null	maximum speed in km/h (= OSM 'maxspeed' tag)
tags	text not null	serialized OSM tags
flags	integer not null	*unused up to now*
waynodes	text not null	serialized list of way nodes
waynodes_red	text not null	serialized list of reduced way nodes

- `street_nodes_0`

  Every row in this table represents a street node independently of wether it is an intermediate or an essential way node.

column	data type	description
id	integer primary key	unique node id (= OSM id)
lat	real not null	latitude of node
lon	real not null	longitude of node
tags	text not null	serialized OSM tags
ways	text not null	serialized list of ways connected to this node
type	integer not null	type of street node (intermediate/essential)

- `poi_nodes_0`

  Every row in this table represents a Point Of Interest.

column	data type	description
id	integer primary key	unique node id (= OSM id)
lat	real not null	latitude of node
lon	real not null	longitude of node
poicode	integer not null	type code of Point Of Interest
tags	text not null	serialized OSM tags
nst	integer not null	id of nearest street node

- `index`

  At the current stage of the project, this table does not contain any crucial data. It is rather intended to be used in future when the map area is split into disjunctive tiles. It then may contain a kind of table of contents stored as key-value-pairs.

column	data type	description
id	integer primary key	unique id of key-value-pair
key	text not null	unique key
value	text not null	value of key

Note: to be compliant with Android's SQLite database, the database has to contain a table named `android_metadata` with a specification of the used locale.

Creation of routing database

As Openstreetmap is the origin of the routing data to be used, one needs a converter reading the Openstreetmap XML file and writing an Android compliant SQLite .db database file. This conversion is done by a class called `MobileTSMDatabaseWriter`.

It automatically selects map elements of desired significance and outputs an reduced, coherent map stored in a database file of the previously explained format. This file can directly by copied to the mobile device to be used as a routing data source.

5.7 Dayplan

5.7.1 Consistency and compliance checks of a day plan

Having set up the basic routing toolkit needed to construct a calendar being more intelligent than a standard one, this chapter wants to give details on this additional "intelligence". A calendar in this context is defined to be a list of events and tasks. A calendar is broken into pieces by default covering one day. Although it may in principle cover any period of time, it will be called *day plan*.

Representation of a day plan (`DayPlan` class)

The `DayPlan` class is a container for events and tasks in the previously explained sense including functionality to operate on its content. Its core component is an algorithm to check whether any arbitrary event in the day plan is compliant with the assumption of "standing" at a fixed but arbitrary location at a fixed but arbitrary time[8]. This compliance furthermore depends on the (estimated) existence of the possibility to use a given vehicle to reach the considered event in time.

A compliance check actually tries to find a route starting from the given location to the location of the considered event and returns the time left between the specified time and the start time of the considered event substracted by the estimated time needed to travers the calculated route. Since location and time to start and the event to consider as a target event are free parameters, this algorithm can be used in many ways to be the basis of every use-case.

TODO: where do events and tasks come from?

Compliance with current position and time

One question our application is expected to give answer to is *"How much time is left until I have to get going towards my next event to be there in time?"*.

The answer is given by passing the current location (probably the last known GPS location) and the current time to the appropriate method of a `DayPlan` instance. By definition, the result is the answer to the question, with a negative value indicating an incompliance between the current position and the next event. Thus it can probably not be reached in time.

TODO: (Alex) some words on notifications

[8] the considered event of course has to have a fixed location and start time

Consistency check

As a day plan probably contains more than one event, it bears potential for inconsistencies in the sense that at least one event cannot be reached in time[9] assuming one adheres to the previous event as fixed in the day plan. The user probably longs to be informed about such inconsistencies.

A consistency check of a day plan is performed by iteratively checking compliance between one event and its preceeding event and returning a list of conflicts. A conflict can either simply be an incompliance between two subsequent events or any other conflict, including an incapability of finding a route or a temporal overlap between two events.

5.7.2 Optimization of a day plan

Tasks may in principle be considered as events with several parameters, including location and start times, not being fixed. The optimization process of a day plan is defined to try to fix the parameters of the given variable tasks in a desirable way maintaining the day plan's consistency. A task with successfully fixed parameters becomes an event and is thus deleted from the task list and the corresponding event is added to the event list. The task is said to be *fixed as an event* or *executed* within the scope the event.

A task not only has variable parameters to fix but can also have a set of constraints of different types. Task constraints are represented by classes implementing the `TaskConstraintInterface`. Possible constraints may apply to the date, the day time, the location or several other parameters (see table 5.1 for a complete list task constraints) of the task. The parameters of a task to be fixed as an event have to obey every constraint in the task's constraint set.

The `DayPlan` class does not perform the optimization itself but allows to set an optimizer that will be used when an optimization process is triggered. An actual optimizer is an implementation of the `DayPlanOptimizer` interface.

name	description
`TaskConstraintDate`	specifies an end and/or a start date
`TaskConstraintDayTime`	specifies an end and/or a start day time
`TaskConstraintDuration`	specifies the time needed to execute the task
`TaskConstraintLocation`	specifies a location the task has to be executed at
`TaskConstraintPOI`	specifies a type of a Point Of Interest the task needs to be executed

Table 5.1: List of currently implemented task constraints

Approaches to optimization

"Optimization" implies the need for a measure of some kind of quality. Optimization then is the process of finding the solution of highest quality. The most desirable way of optimizing a day plan is to input a day plan (consisting of events and tasks) and a measure of quality (metric) into a black box algorithm returning the best day plan possible maintaining day plan consistency. Since this is probably one of the most challanging problems

[9] using one single specified vehicle

in computer science, we are forced to make strong simplifications and accept the fact that the solution is likely to be not the optimal one.

Our basic approach to this problem is to use a greedy algorithm recursively trying to fix tasks between the events of a day plan, implemented in the class `GreedyTaskInsertionOptimizer`. The optimization is started specifying a location, a time and a vehicle where the location and the time are used as a basis (intent of it to become clear in the next clause). The algorithm iterates over the list of tasks in the order given by a `TaskPriorityComparator`. For every task, it consecutively checks compatibility between the constraints associated with the task and the parameters it is potentially fixed with. These potential parameters are determined based on the basis location, time and the task's constraints.

Given one task within the iteration over all tasks, parameters are tried to be fixed by processing the folling sequence:

1. check wether the task's duration[10] is less than the time left until the next event starts.
2. If so, check whether the time given as a basis is compatible with the task's date and day time constraints. In other words, check whether a task can be executed at this point of time.
3. If so, find a location where the task can be executed. In dependence of its constraints, this may either be completely undefinite, or any Point Of Interest of a definite type or even a definite location. Select the location that is compatible with the task's constraints and that minimizes the sum of the linear distances from it to the basis location and the location of the next event[11]. Find a route from the basis location to the potential task execution location and a route from there to the next event's location. Check whether both routes exists and whether the time needed to travel both routes plus the duration of the task is less than the time left until the next event starts.
4. If so, fix the task.

The first task that is fixed as an event forces the optimization process to restart recursively with the recently fixed end time and location of the task as its new basis.

Note: Since the task's date and day time constraints are only required to be compatible with the basis time, the task is neither guaranteed to be compatible with its actual start time nor with its duration.

Implementations of a `TaskPriorityComparator`

Although the greedy algorithm stays the same, the way to influence the optimization process is to modify the implementation of the `TaskPriorityComparator` which is therefore designed as an interface. By prefering one task to another one in this context, one can in principle force the algorithm to aim for different optimization goals.

Two implementations of a `TaskPriorityComparator` already exist:

[10] specified by its constraints

[11] this effectively considers locations lying on an ellipse with the next event's location and the basis location as its focal points to be equivalent

- `SimpleTaskPriorityComparator`

  The `SimpleTaskPriorityComparator` prefers tasks with more date or day time constraints. Only the number of constraints is considered, details of the constraints are disregarded. The idea is to fix demanding tasks prior to others.

- `UrgencyTaskPriorityComparator`

  The `UrgencyTaskPriorityComparator` prefers tasks of high urgency. High urgency is considered to be given by an early execution end date and/or time in turn given by the task's constraints. The idea is to fix as many tasks as possible.

Other implementations may for example prefer tasks close to the current position to reduce the total distance that has to be traveled.

5.8 Persistent Storage

For persistent database storage, the Android API contains support for SQLite databases. Any application can create an SQLite object, which is stored on the device under "/data/-data/package_name/databases"

Unfortunately, up to Android version 2.1, the installed support for SQLite is 3.5.9. This means that it is not possible to benefit from the support for R^*-Trees, which was included in SQLite 3.6. This would have meant a very comfortable way of storing location data and retrieving positions in vicinity of a given point.

TODO: shortly describe the key-value store and how to use it

TODO: mention that there is also the possibility to store data directly on the SD-Card

6 Conclusion and future Work

TODO: describe what was accomplished in the project - gained knowledge for the devs - list reusable sytem parts - shortly describe functionality of the application

6.1 What to do differently

Milestones

Unit tests

Early in the development process we decided to not do unit tests. At the time there were several reasons for that. As the routing engine and the calendar integration were the first things to be developed, they were crucial in the decision process. Unit testing on the Android platform seemed rather complicated, especially for code using the content provider backend. In a discussion with our advisor, we decided that it would be too time consuming, considering how much time already had passed for specification and platform choosing. Therefore we decided to concentrate the time left on actual development.

However, we had several situation in the development process, where unit testing would have been useful. Many problems only became apparent when certain elements were integrated together. Unit tests could have revealed these much earlier. Especially as we developed on our own much more often than anticipated. This accounted for more problems whilst integrating, which could also have been minimized by unit testing.

Bug tracker

Although a good bug tracker is provided by github, we didn't use it until the very end of the project. In retrospect it would have been better to utilize the bug tracker right from the start. This would have forced us to look deeper into bugs and better document them, if similar bugs appeared later.

Especially in combination with unit tests the bug tracker could have improved the development process significantly. A good approach, for example, would have been, to create a unit test for every occurring bug. This means that there is an immediate way to test if the bug was fixed and that it would not be introduced again later.

TODO: describe what should be done differently next time - shorter development intervals, more milestones

TODO: describe possible followup projects: - use system parts to build other mobile routing related application - refine this application, make it market-ready - improve system parts (especially routing / routing-performance)