Kangaroo
a mobile, location-aware task planer

Alexander Gutjahr   Andreas Walz   Daniel Schauenberg

April 15, 2010

Contents

1 Introduction

TODO: write some more introduction to the project

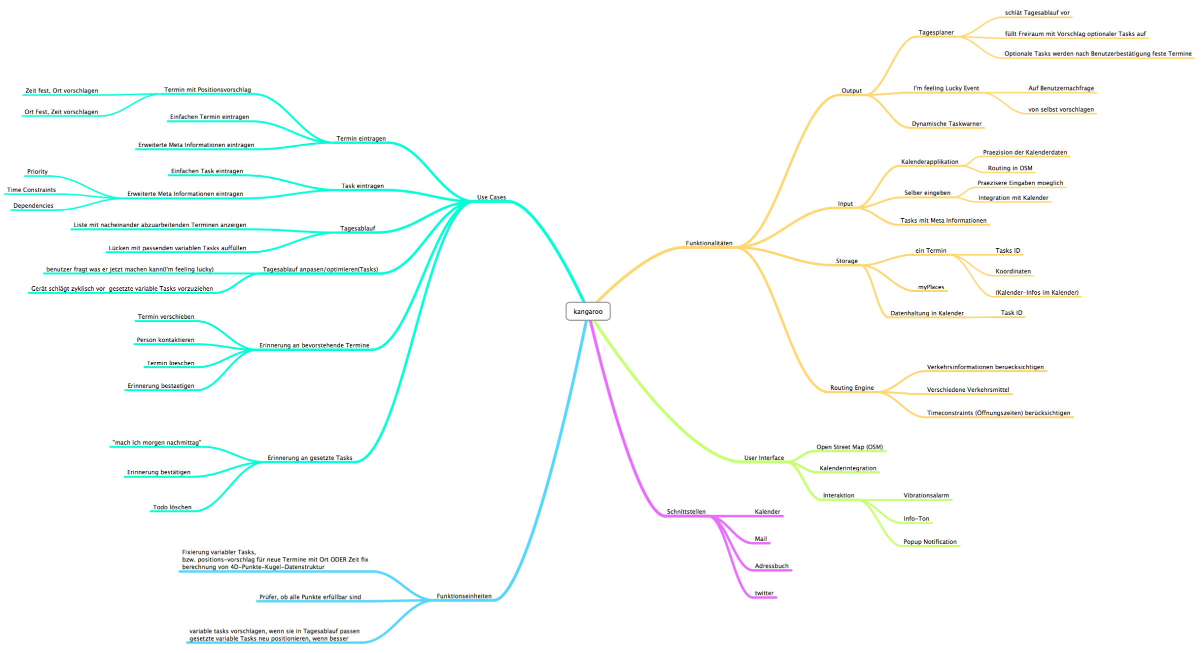
This document contains all data, which was generated while working on the kangaroo project.

2 Definition of project scope

2.1 External requirements

TODO: Introduce process of requirement refinement: external (from klaus) –¿ internal (resulting rev. goals) –¿ appropriate system design and dev. process

TODO: outline the requirements from Klaus



kangaroo
Use Cases
Termin eintragen
Termin mit Positionsvorschlag
Zeit fest, Ort vorschlagen
Ort Fest, Zeit vorschlagen
Einfachen Termin eintragen
Erweiterte Meta Informationen eintragen
Task eintragen
Einfachen Task eintragen
Erweiterte Meta Informationen eintragen
Priority
Time Constraints
Dependencies
Tagesablauf
Liste mit nacheinander abzuarbeitenden Terminen anzeigen
Lücken mit passenden variablen Tasks auffüllen
Tagesablauf anpasen/optimieren(Tasks)
benutzer fragt was er jetzt machen kann(I'm feeling lucky)
Gerät schlägt zyklisch vor gesetzte variable Tasks vorzuziehen
Erinnerung an bevorstehende Termine
Termin verschieben
Person kontaktieren
Termin loeschen
Erinnerung bestaetigen
Erinnerung an gesetzte Tasks
"mach ich morgen nachmittag"
Erinnerung bestätigen
Todo löschen
Funktionseinheiten
Fixierung variabler Tasks, bzw. positions-vorschlag für neue Termine mit Ort ODER Zeit fix berechnung von 4D-Punkte-Kugel-Datenstruktur
Prüfer, ob alle Punkte erfüllbar sind
variable tasks vorschlagen, wenn sie in Tagesablauf passen gesetzte variable Tasks neu positionieren, wenn besser
Funktionalitäten
Output
Tagesplaner
schlät Tagesablauf vor
füllt Freiraum mit Vorschlag optionaler Tasks auf
Optionale Tasks werden nach Benutzerbestätigung feste Termine
I'm feeling Lucky Event
Auf Benutzernachfrage
von selbst vorschlagen
Dynamische Taskwarner
Input
Kalenderapplikation
Praezision der Kalenderdaten
Routing in OSM
Selber eingeben
Praezisere Eingaben moeglich
Integration mit Kalender
Tasks mit Meta Informationen
Storage
ein Termin
Tasks ID
Koordinaten
(Kalender-Infos im Kalender)
myPlaces
Datenhaltung in Kalender
Task ID
Routing Engine
Verkehrsinformationen beruecksichtigen
Verschiedene Verkehrsmittel
Timeconstraints (Öffnungszeiten) berücksichtigen
User Interface
Open Street Map (OSM)
Kalenderintegration
Interaktion
Vibrationsalarm
Info-Ton
Popup Notification
Schnittstellen
Kalender
Mail
Adressbuch
twitter

Figure 2.1: Mindmap with project goals, requirements and elements

2.2 Resulting development goals

TODO: describe deve goals that result from the external requirements

2.3 Abstract System Structure

TODO: refine text

TODO: describe system design and explain how the dev. goals are reflected in the system design

TODO: adapt graphics to reduce difference between initial design and resulting software :)

One important goal for the project was to create the 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 software platform have been made.

The sole basis for the system structure shown in this chapter are the requirements listed in chapter 2.1.

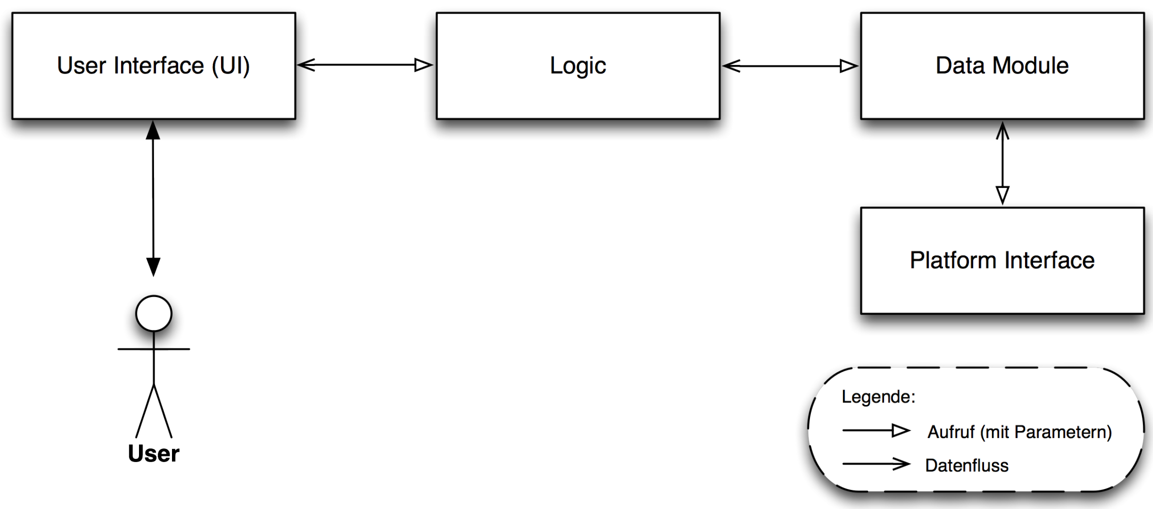


Figure 2.2: High level design of the system components

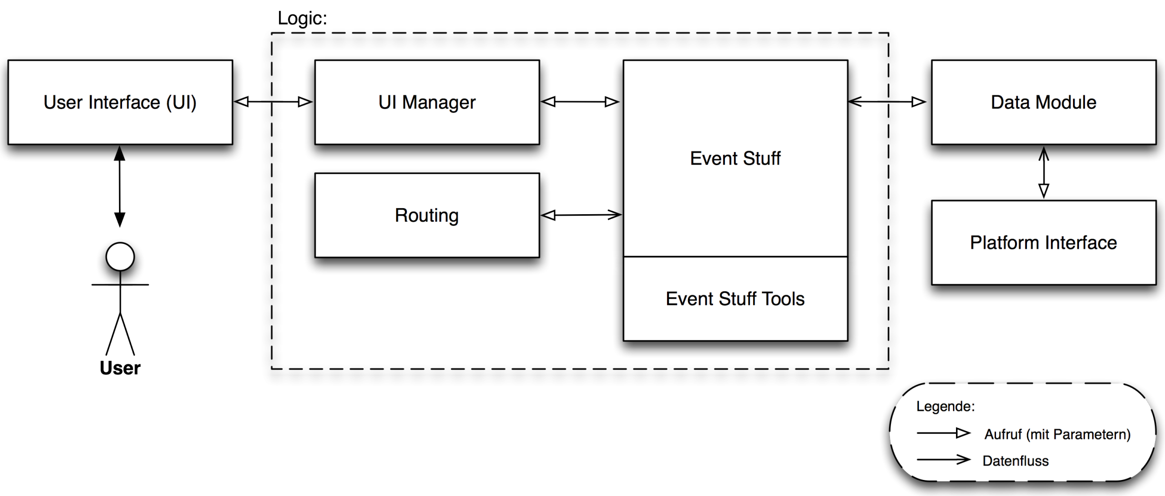


Figure 2.3: Representation of the logical system elements

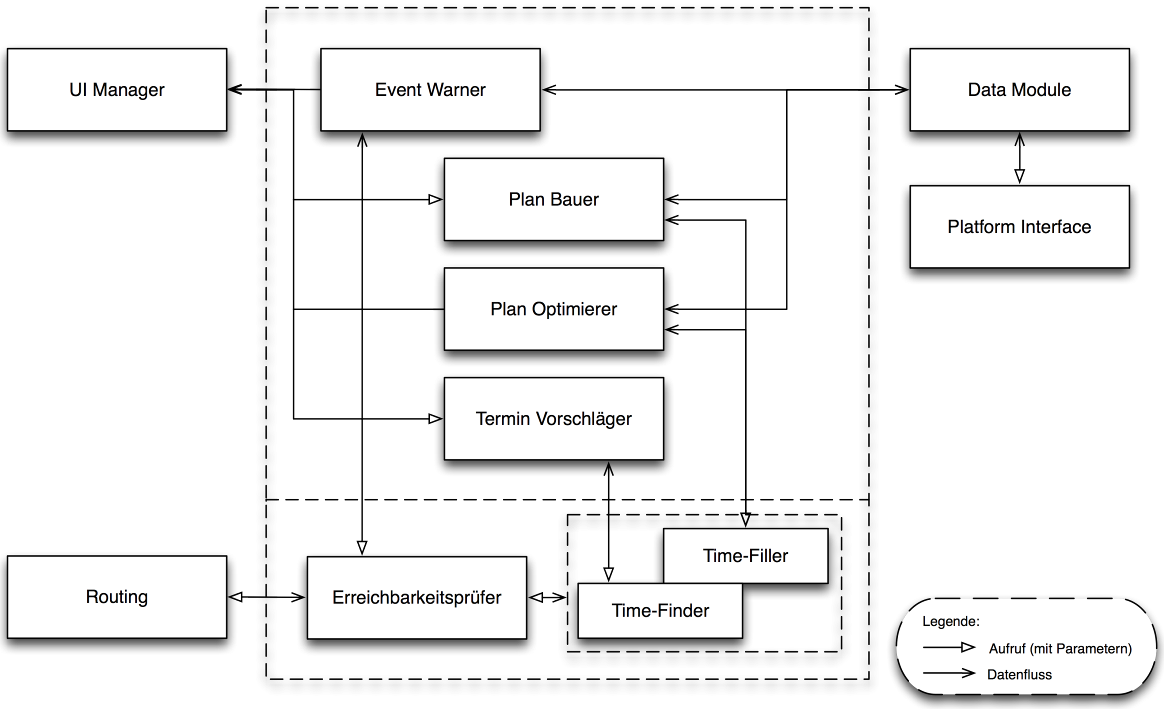


Figure 2.4: Components of the Appointment/Task optimization and verification engine

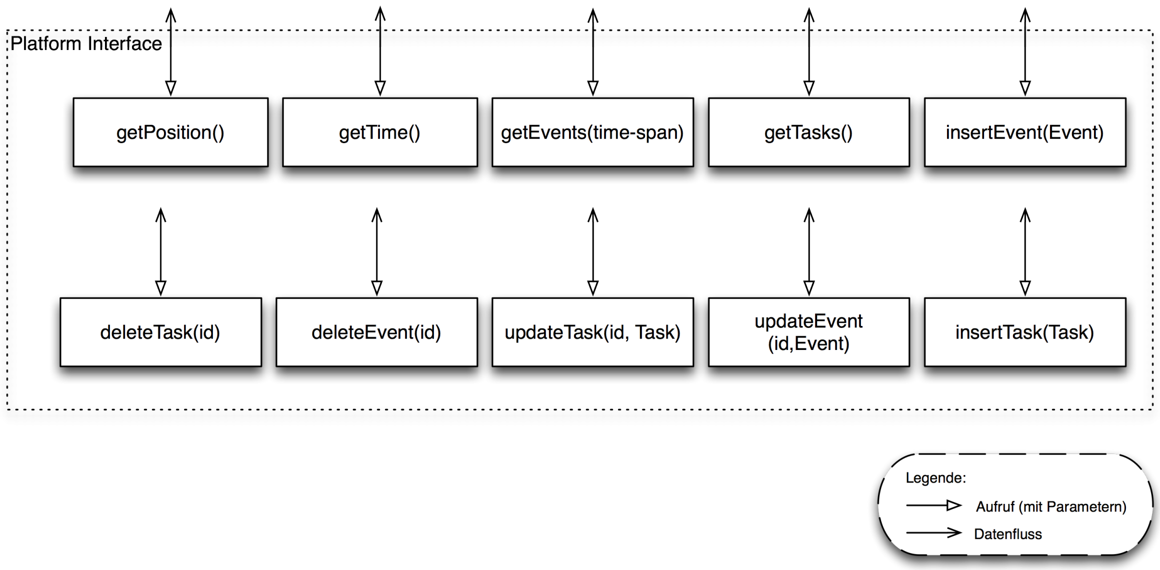


Figure 2.5: Highlevel interface for data storage and system interaction

3 Project plan and development process

3.1 Initial project plan

TODO: write a few sentences to describe the GANTT diagram

Kalenderwoche	42	43	44	45	46	47	48	49	50	51	52	1	2	3	4	5	6	7	8	9	10	11	12	13
Anforderungsanalyse/Ideen sammeln	🟦	🟦	🟦																					
Highlevel Systemdesign				🟪																				
Realisierbarkeit prüfen					🟨	🟨	🟨	🟨	🟨	🟨														
Plattformen evaluieren								🟧	🟧	🟧	🟧													
Detailiertes Systemdesign/Plattform wählen											🟨	🟨												
Implementierung													🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥		
Test / Code-Qualität													🟫	🟫	🟫	🟫	🟫	🟫	🟫	🟫	🟫	🟫		
Demo/Präsentation vorbereiten																							🟩	🟩
Dokumentation	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩

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

3.2 Refined project plan

TODO: insert second GANTT-diagram
TODO: extend task-list, introduce the list as milestones for second GANTT-diagram

- longer platform evaluation phase
- added technology scouting phase for selected platform
- add milestones in implementation phase

At the beginning of calendar week 7 it became obvious, that the number of open ToDos and the amount of time left until the project end after calendar week 13 required a more success-oriented development methodology.
For this purpose the development block (red in GANTT diagram) needs to be divided into smaller tasks or milestones, both for already completed work and for future tasks.

Up to this point, we were able to fulfill the following tasks:

- gather system requirements
- define clean abstract system structur
- choose appropriate OS and platform for the project
- derive a OS specific system structure
- install and get used to Android SDK

- 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

For the remaining 6 weeks of the project the following ToDos are left:

- Determine performance of routing engine, identify bottlenecks and problems for future fixing
- 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)

3.3 Development process

TODO: describe development process: iterative waterfall model with prototyping, why we chose this process, what tools we use (git)

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

TODO: which other OSs didn't we consider:
- Blackberry OS - symbian OS - Maemo(?)

4.2 Routing

TODO: Andi, check that I divided general and android specific part of the routing text correctly. TODO: review and finish this chapter :)

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 and to account for local traffic facilities. This implies the need for a set of data representing these facilites and an engine to operate on the data.

4.2.1 Routing data

From a very abstract and general perspective, this set of 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 and highways). Dynamic facilities are the ones, that have a more or less fixed schedule constraining their usage (for example tramways and public bus service).

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 traffic facilities as there would be a number of carriers with different interfaces to include. However, the user might notably benefit from the abolishment of this restriction. It might be part of future work to include this feature.

As potential sources for static traffic facility 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 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.
  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. 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.

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

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

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.

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 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. 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

[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

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 scouting and therefore the next chapter 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.

Technology scouting - general part

This section will give an overview of the general Technology Scouting process.

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 scouting, 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 scouting 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 scouting 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 a rough 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**

  TODO: explain why not Gosmore

- **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 TSM to read an Openstreetmap XML file stored on the SD card of the mobile device and to output the file's summarized content. Compiling and running this application was possible without any errors but revealed three initial problems:

[6]remember the only alternative at that time was Gosmore

  - 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 discapability 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 4.2.2 will give details).
  - memory usage

- **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**

  TODO: explain why not OSMAndroid

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.

[7] `javax.xml.parsers.SAXParser`
[8] `org.xml.sax.helpers.DefaultHandler`

- 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, ...

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

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.

- 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. 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.

Technology scouting - specific part

This section will give an overview over the Traveling Salesman specific Technology Scouting process.

- changes applied to the XML parser
- test system (mainly map file) used to make first progress with performance

[...]

5 Developement for Android

5.1 Platform Specific System Design

TODO: correct the graphic to reflect most current system design

TODO: describe the graphic

TODO: write why we did the design like that (briefly introduce android SW structure with services, activities etc.)

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

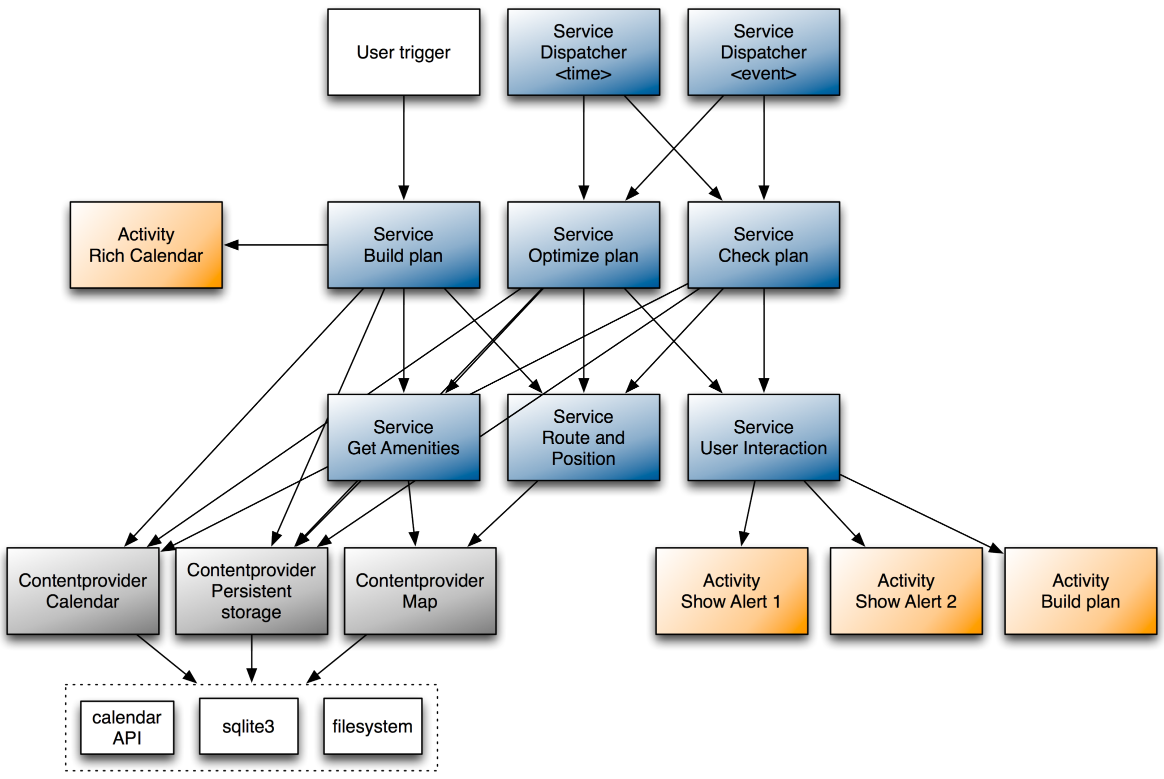


Figure 5.1: Platform specific software structure for Android OS

5.2 Description of system elements

5.2.1 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.

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.

Edit Task

The EditTask Activity 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.

Configuration Editor

The Configuration Activity 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.

Day Optimizer

The DayOptimizer Activity 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".

Notifications and alerts

TODO: describe the notifications the user gets from the background task

5.2.2 Background tasks

TODO: graphic that shows all background-services

TODO: describe graphic, explain serviceCall*, serviceRecurringTask and the Brouadcastlistener

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

5.2.3 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.

TODO: This is important for future development. describe shortly how the API is used, brag about how awesomely we reverse-engineered it. reference the calendar library where our calendar wrapper can be found.

5.2.4 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

5.2.5 Routing engine

TODO: review and finalize this chapter

TODO: Andi: check that I divided the routing chapters correctly (generic vs. android specific)

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.

Classes and interfaces

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 part 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.

- 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 `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 `Limits`

  This class is used to define geographical constraints for searches on the map.

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.

What are the problems?

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 sinle element requested by the routing engine.

   TODO: give values

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.

   TODO: give values

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.

TODO: give values

What is our solution?

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 archtitecture and improve the search algorithm for nearest nodes. The next sections will outline ideas and implementations.

TODO: still to mention:

- filtering intermediate way nodes implies need for an extra field holding distance between street nodes.
- derivates of Traveling Salesman's classes `Node`, `Way` and `WayNode`.

Reducing and individualizing of routing data

The basic idea is to introduce a module, the `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.

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.

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

Search algorithm for nearest nodes

Routing engine of MobileTSM

- Class `MobileTSMRoutingEngine`

  The class `MobileRoutingEngine` is the *MobileTSM* implementation of a `RoutingEngine`.

- Class `Vehicle`

- Class `Limits`

Tests and benchmarks for MobileTSM

In order to test the performance and reliability of the routing engine that is working behind the routing interface, 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[1] 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[2] 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

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

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 be fixed. These are

- The map file that contains the geographical 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.

5.2.6 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

TODO: describe what should be done differently next time - use bug-tracker from the start - shorter development intervals, more milestones - unit-tests to improve code quality

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)