University of Mannheim
Data and Web Science (DWS)

mobEx
Technical Report

Timo Sztyler Michael Jess Christian Bikar
Bernd Opitz Bernd Pfister Florian Knip
Ansgar Scherp

December 19, 2013

Contents

API Application Programming Interface

APK Android application Package File

DTD Document Type Definition

EULA End User License Agreement

FMC Fundamental Modeling Concepts

GoF Gang of Four

GUI Graphical User Interface

GPS Global Positioning System

HCI Human-Computer-Interaction

HTML Hypertext Markup Language

HTTP Hyper Text Transfer Protocol

HSV Hue-Saturation-Value

ID Identifier

JSP JavaServer Pages

JSON JavaScript Object Notation

MVC Model-View-Controller

POI Point Of Interest

RDF Resource Description Framework

REST Representational State Transfer [3]

SCM Source Code Management

SPARQL SPARQL Protocol And RDF Query Language

TAM Technical Architecture Modeling

UML Unified Modeling Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

UI User Interface

UID Unique Identifier

UUID Universally Unique Identifier

W3C World Wide Web Consortium

XML Extensible Markup Language

1. Introduction

1.1. Motivation and Goals

This document describes the technical details of *MobEx*, a nearby search app for `Places` and `Events`. The system consists of two tiers, an Android application as client and a server. The main functionality of *MobEx* is to retrieve `Events` and `Places` based on the user's location from several different data sources. On the server side, duplicates are filtered out and matching entities unitized before sending the data to the client. There it can be either shown on a map or in a result list that can be filtered according to categories (or facets). One distinguishing feature of *MobEx* is the integration of the time dimension. Usually `Events` happen at a certain time or `Places` like shops have opening hours. In *MobEx* `Events` and `Places` can be explored not only in a regional dimension but also in a temporal one. To support this exploration, we added novel UI elements, like a time wheel. Furthermore *MobEx* utilizes a new UI approach to browse through facets and filter the result set. During this project, a user study has been conducted in order to investigate the user behavior and the acceptance of those features. The distinct features on server side cover the implementation of an own data schema on which the different schemas of the data providers are mapped to and an *Entity Resolution* process that eliminates duplicates and merges similar `Events` from different sources into one object. Another goal of the project was to create a hierarchical facet structure that is compatible to those of the data providers, so the data can be inserted into our hierarchy. In this document, all concepts and their functionality will be explained.

1.2. Structure

The document is structured as follows: In the next chapter, the client part of *MobEx* is introduced. First, the overall architecture and the concepts of the user interface is shown before implementation details are explained. The client chapter concludes with suggestions for future work. The third chapter that deals with the server is structured similar to the previous one. First the concept is explained, comprising the overall architecture, data providers and technologies. After that, implementation details (e.g. data querying, *Entity Resolution*) are explained followed by a chapter about future work. Topics concerning both client and server are presented in the fourth chapter. Project management as well as project organization are explained in the first part. Continuous integration concepts and our implementation of those, such as version control or building and releasing software artifacts are shown in the second part.

1.3. Scope of functions

Client

The mobEx client Android application serves as the viewer of the distributed mobEx system, consisting of the client app and the server introduced in Part 3 of this documentation. Both system components communicate via a RESTful interface in order to exchange event[1] and facet[2] data. This part of the documentation deals with the design and implementation of the components required for the following features:

- Browsing events in a list and on a map
- Browsing facets
- Dynamically incorporating batches of events as they are resolved by the server (see Section 3.2.2.4)
- Filtering events by facets, keywords and time
- Choosing between facet browser layouts
- Choosing what data providers to incorporate in the event search
- Gathering usage information for Human-Computer-Interaction (HCI) research purposes

Server

The mobEx server has multiple tasks or functions within the mobEx system. Firstly, it answers to client queries, i.e. the mobEx client application communicates with the mobEx server instead of the data providers. The server in turn queries the data providers, processes the results and forwards them to the client. The server thus takes the role of a mediator. The server-side processing integrates information into a scheme that is powerful enough to represent information about `Events`, `Organizations`, `Persons` and `Places` (Resources). On this scheme, an entity resolution process takes places that eliminates potential duplicates and merges information. The results are then sent to the client as the reply to their request.

[1] Due to the temporal focus of the mobEx application, the term *event* was coined in order to describe what usually is referred to as *resource* on the server-side.

[2] The testers of the first mobEx prototype coined the term *facet* for the browsable categories.

2. Client

2.1. Application Design

This section deals with the design and specification of the mobEx client application. First the application architecture and the components involved are discussed, introducing their roles and basic concepts. Afterwards the Graphical User Interface (GUI) design is explained by means of mockups of the different application screens and widgets. Finally the packet structure is presented, giving an overview of the interdependencies of the application modules.

2.1.1. Software Architecture

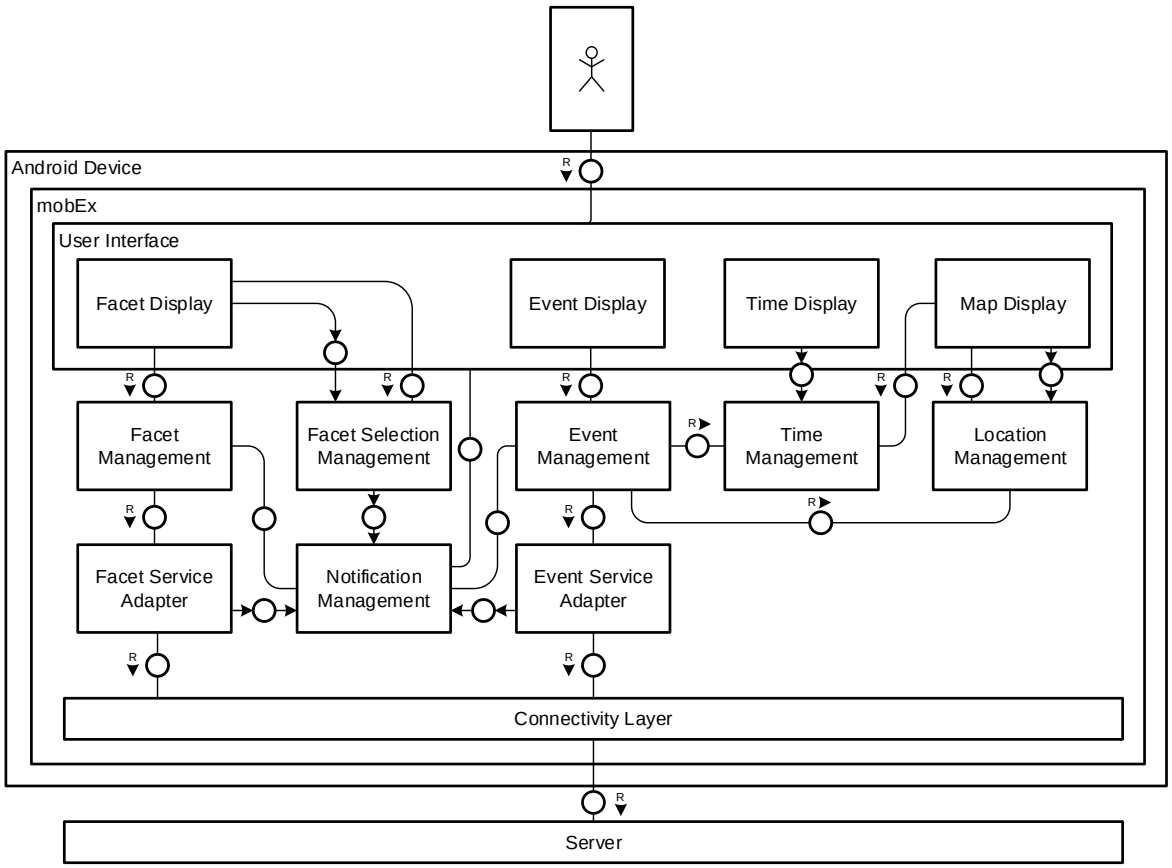


Figure 2.1.: Client application architecture. The FMC TAM block diagram notation is used, which is documented in Appendix A.3.

2.1.1.1. User Interface

Based on the functional scope introduced in the previous section, the user interface consists of four major components: The *facet, event, time* and *map display*. The facet display is used to present the categories the user may choose from. The resulting facet selection serves as a category filter for the event display in which all events such as people, places, organizations and events (in terms of concerts, activities etc.) are listed. Apart from this approach of listing events, the data may also be inspected on a map. This is depicted in the form of a variety of icons reflecting one of each events categories[1]. Finally, those views are complemented by a novel time display, which offers additional temporal filtering of the events.

Section 2.1.2 deals with the user interface specifics in greater detail.

2.1.1.2. Business Logic

The application business logic comprises a number of core controllers and services that connect the basic data services and the user interface. For the inter-component communication an event-based approach is chosen in order to reduce interdependencies to specific implementations. The *notification management* abstracts data sources and the like in terms of *notifications*. Instead of requiring a reference to one of the other components, controllers and GUI components may register themselves with the notification management in order to be notified about specific application events[2].

One of the most important events is the availability of new facet data. Loading that data is triggered once on each application startup. The facet management listens for the corresponding event and creates a new facet service adapter, which loads the data asynchronously from the server via the connectivity layer. As soon as it is finished, the FACETMANAGER is notified about the results using another notification.

The same pattern is applied to the event management, except that there are two additional situations in which a data reload is necessary. Firstly, the user may always trigger a manual reload in order to retrieve more current data. Secondly, whenever the map viewport in the map display is changed, data for the new search location should automatically be loaded. The actual data conversion and connection handling is dealt with by the event service adapter and connectivity layer respectively in this case.

In addition to these core data services, there are three more rather GUI-centric components. The location management keeps track of the current user and the event search location, which are not necessarily the same. When the application is started the search and map location are both initialized to the current user location. Afterwards the user may change the search location any time, while the actual user location is automatically supervised by the location management. This enables the map display to return to the current location whenever necessary.

[1] Each event may occur in several facets, so that one needs to be chosen for the icon.

[2] Not to be confused with *events*, a term used synonymously with the server-side *resource*. This difference in vocabulary originated from the first mobEx implementation, MFacets, which was centered around location-based data. With mobEx, the focus shifted towards temporal data, so that the client team coined the term event to describe what used to be a resource. The server team, however, stuck with the old terms.

The time management, on the other hand, gives access to time-based information which is mainly used to filter the map display for events based by time. Using the time display, this period may be viewed and adjusted so that the map display is able to hide events that are not relevant to the current time selection.

Finally the facet selection management keeps track of the current facet selection. It is important to note that this component is kept separate from the actual facet data management, since the latter is centered around both loading facets and managing facets. In this context, management includes not only keeping track of what has been loaded from the server, but also providing an interface for dynamically adding facets as needed and assigning events to the categories in the tree. In contrast, the facet selection management hand is concerned with the subset of facets that has been chosen by the user for category filtering. In contrast to the FACETMANAGER, it provides a comprehensive interface for managing and querying the selection state of specific facets. Apart from the event management, this is of particular interest for the GUI components in order to distinguish selected from unselected facets. Changes to the facet selection are not directly propagated to the corresponding handler components but also broadcasted via the notification management.

2.1.1.3. Data Services

The data services comprehend the *connectivity layer* and the service-specific *event* and *facet service adapter*. The connectivity layer encapsulates concerns such as the server end point, connection initiation and creating streams for the service adapters. When the connection is established, the service adapters in turn are used to transform the data streams returned by the connectivity layer into client-side data models, such as events or facets. During and after this transformation process the adapters may notify other application components about the progress and results via the notification management.

2.1.2. User Interface

This subsection deals with the user interface design. By the means of mockups the widgets of the display components introduced in the previous subsection and their behavior are explained.

2.1.2.1. Browsing Screen

Figure 2.2 shows a mockup of the first of two application screens, the *browsing screen*. In the lower portion of the mockup you can see the *facet display*, with the *event display* on top (cf. Section 2.1.1.1). As its name suggests, this screen is tailored to browsing the data provided by the server. The action bar at the top border of the screen contains a number of tools for interacting with the application in general and the facet and event display in general. The leftmost option is used to switch between the browsing screen and the map screen. The rightmost button opens the *action overflow*, which replaces the former hardware menu button [10]. The other items (left to right) are the facet and event display specific search, event sorting, facet selection and reload and will be covered in detail in the corresponding sections.

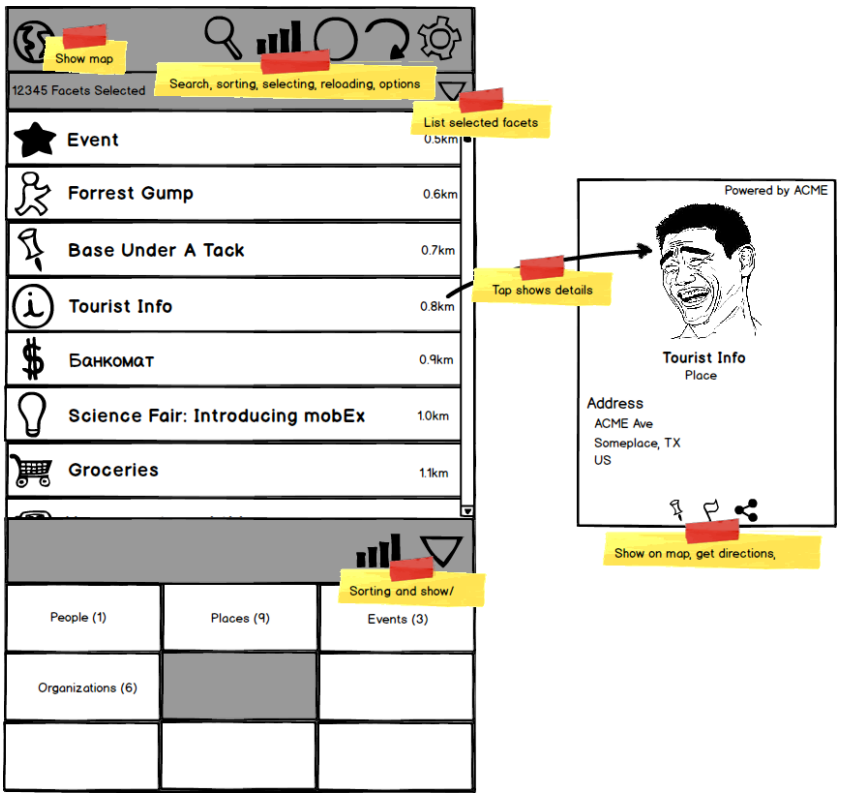


Figure 2.2.: Annotated client event and facet display mockup

Facet Display The facet display presents one level of the facet hierarchy at a time and lets the user navigate through it. The number in brackets behind a facet name indicates the number of events associated with the facet. A simple press on any leaf of the facet tree toggles the leaf selection, while pressing any other node will "open" that node, displaying its children. In order to select a node, the user needs to perform a long press on it, resulting in a selection of the whole facet subtree. In addition, the action bar on the top border of the screen contains a button for toggling all currently displayed facets. For visual feedback, selected and unselected facets should be distinguished by color, so that unselected facets appear in white and selected ones in light gray. The top border of the facet display serves as toolbar and contains two buttons, one for toggling its sorting mode (by facet name, by event count in facet) and one for toggling its collapsed state. When collapsed, the facet toolbar should still be visible at the bottom of the application so that its visibility may be toggled again. The widget toggling the sort order should be hidden, though, since in the collapsed state it serves no purpose and might distract users from the other controls on the screen.

In order to view all selected facets, there is a drawer right below the action bar. To the left the number of currently selected facets is shown. It can be pulled down for a complete list of all chosen facets and enables the user to deselect one of them at a time or all of them at once. In addtion to the pull-down option, pressing the arrow to the right opens the drawer automatically and closes it again when it is open. There are two kinds of facet display available which are described below. After a 50, interactions with the facet display the users are displayed a hint

indicating that they may choose between both kinds using the preferences screen (see Section 2.1.2.3). This hint is shown once more after another 450 interactions.

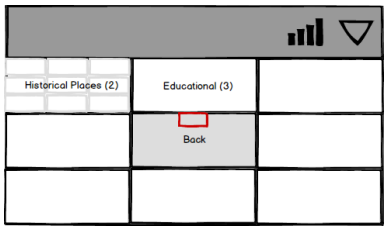


Figure 2.3.: Facet grid with back control

Facet Grid In Figure 2.2, one of two facet displays to implement, the *facet grid*, can be seen. In the grid layout, the facets are displayed in eight out of nine evenly arranged cells, with the ninth cell in the center reserved for a *back navigation*. Figure 2.3 shows this control in detail. Leaves in the facet tree have a plain white background while other nodes display a light gray grid, indicating that there are children below (see top left cell).

When a node in the tree is opened, e.g., the top middle facet in this example, the corresponding position in the back navigation is marked with a colored square. This serves as visualization of the selection path. In addition, a label appears on the navigation indicating that this area now serves as new GUI control for navigating up in the facet tree. If the eight cells do not suffice for displaying all children, the bottom-rightmost facet is replaced with a button that opens an overflow dialog, in which all remaining facets are listed.

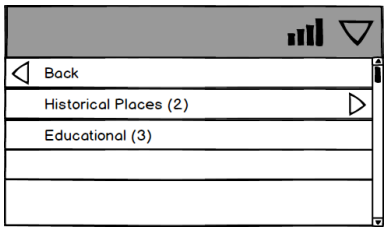


Figure 2.4.: Facet list with back control

Facet List The other type of facet display, the *facet list*, is shown in Figure 2.4. It behaves largely similar to the facet grid, i.e. selecting and navigating works the same way. There are three major differences, though:

- Nodes other than leaves are not marked by a grid in the background but by an arrow to the right, which indicates that further inspection is possible.
- The back navigation is replaced by a special list entry that is always on top, which also has “Back” as its label and features an arrow to the left.
- Instead of the grid, the list may contain arbitrarily many entries, which means that there is no extra overflow dialog.

Event Display The event display lists all selected events according to two filter criteria: The current facet selection and, if applicable, the current filter string. The filter string may be specified by pressing the search icon in the action bar, which will hide all action items and display an input field instead. Only those events that contain the filter string in their name should be displayed. The specifics of selecting and deselecting facets are covered in the previous section. If facets are selected, only those events assigned to at least one of the selected facets should be shown. Having no facets selected is handled as if all facets are selected, such that specifying no category filter displays all events. Both filters can be combined, so that the user may, e.g., first make a rough selection in terms of facets to narrow the search space and then refine the search using a keyword. It is important to note that the time selection is not reflected on this screen due to practical reasons: On the one hand, there is too little room left for a control like the time wheel (see Section 2.1.2.2). This means that a tinier control is required, which would most certainly be far less usable. On the other hand, the temporal dimension makes much more sense on the map display where the user has dozens of events at a glance. In contrast, even large devices can display only around fifteen events at a time, assuming that the facet display is collapsed.

For each event to display a list entry is created with an icon representing one of its facets. If an event has several facets, one needs to be chosen. To the right of the icon the event name and distance to the current search location is displayed. By default, the list is sorted by distance, but switching to alphabetical order and back again is possible by pressing the action item to the right of the search icon.

The event list is loaded each time the application is started and refreshed when the user changes the search location on the map screen. Furthermore the user may trigger reloading the data manually by pressing the reload button to the left of the action overflow in the action bar.

2.1.2.2. Map Screen

A mockup of the map screen is shown in Figure 2.5. As in the browsing screen, an action bar occupies the upper portion of the screen with the action overflow (help and preferences) at the rightmost border. The leftmost control lets the user navigate back to the browsing screen. Clicking the search icon triggers search mode, which removes the action items from the action bar and replaces them with an input field. In this field the user may enter a location name, e.g, a city, and the map viewport will be set to that location. Using the button hovering over the map right below the action overflow users may reset the viewport to their current location. The map surface itself behaves like any application using Google Maps, i.e., panning moves the viewport and pinching zooms in and out. Double-panning, i.e. panning with two fingers at a time, adjusts the viewing angle. All events matching the filter criteria from the browsing screen (keyword and facet filter) are displayed on the map using markers representing one of their facets. If there is a sufficiently large number of map markers next to each other, the events are clustered and a cluster marker is used instead. Tapping this marker displays a list of events. When a list entry is chosen, its details are displayed, much like tapping on a regular map marker. We chose that design so that users to not have to deal with overwhelming amounts of data at a time.

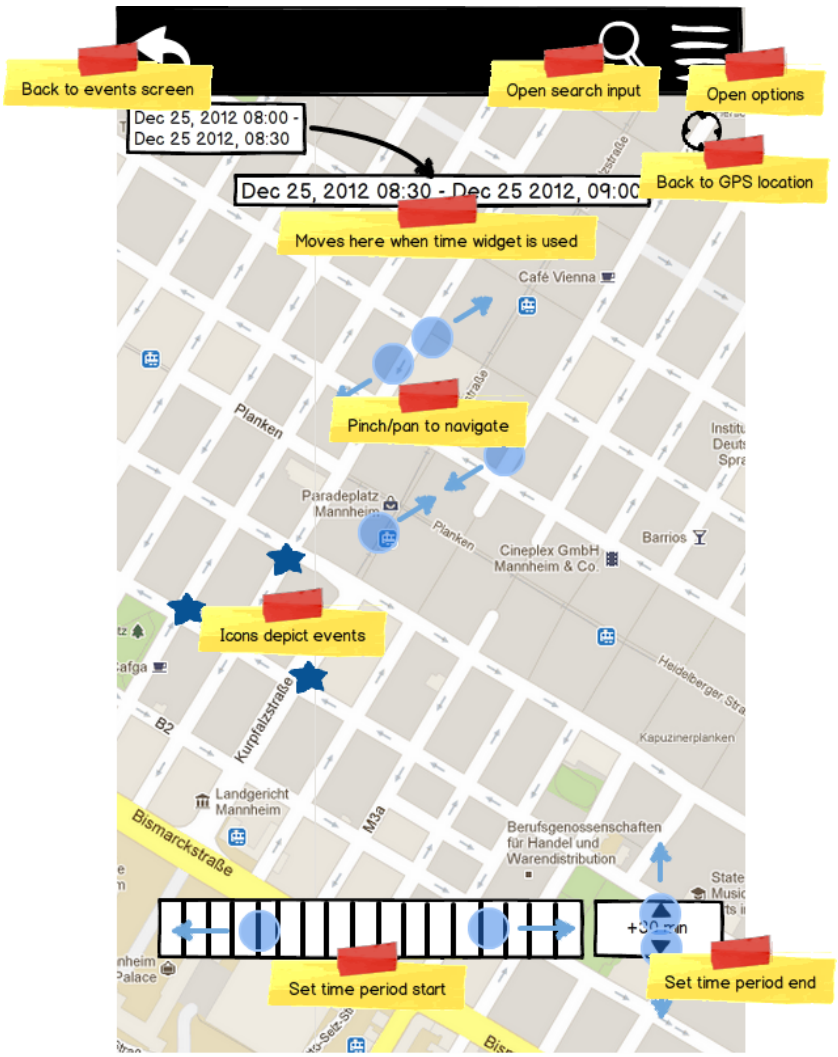


Figure 2.5.: Annotated client map mockup

In order to offer another dimension of data exploration, i.e. browsing in time, the time display is integrated into the map screen. In the top-left corner of the screen, the current time selection is displayed. By tapping on it, a traditional date chooser is opened which lets the user change the current time. Alternatively the novel time wheel at the bottom of the screen may be used. It can be flung and scrolled to the left and right, going backward or forward in time respectively. Both the date chooser and the time wheel only set the start time, even though a time period is used as a filtering window over the events displayed on the map. The period end is chosen relatively to the start using the button to the right of the time wheel. The corresponding widget can be seen at the bottom of Figure 2.5. Tapping it lets the user cycle through all possible values. Events that occur during the period selection as well as those that have no time information attached are displayed opaque. Others are shown increasingly transparent as they happen further away in time.

Early drafts included several solutions to the period selection problem. Even an approach using two time wheels was drafted, but dismissed shortly thereafter. Appendix A.4 contains

a list of the remaining drafts that were considered more intensively. The first two suggest using a single time wheel for adjusting both the period start and end, with two different switch designs for toggling the wheel between those two modes. Finally the event refrained from this solution because it was not sufficiently intuitive and usable.

The third draft is close the the final choice. It already considers the period end as relative to the start, but the difference could be adjusted by swiping up and down. After a prototype test, however, it turned out that one might easily hit the back button of the phone, eliminating this draft as well. For the implementation the idea of a relative period end was kept, but the values could be cycled by tapping instead.

2.1.2.3. Preferences Screen

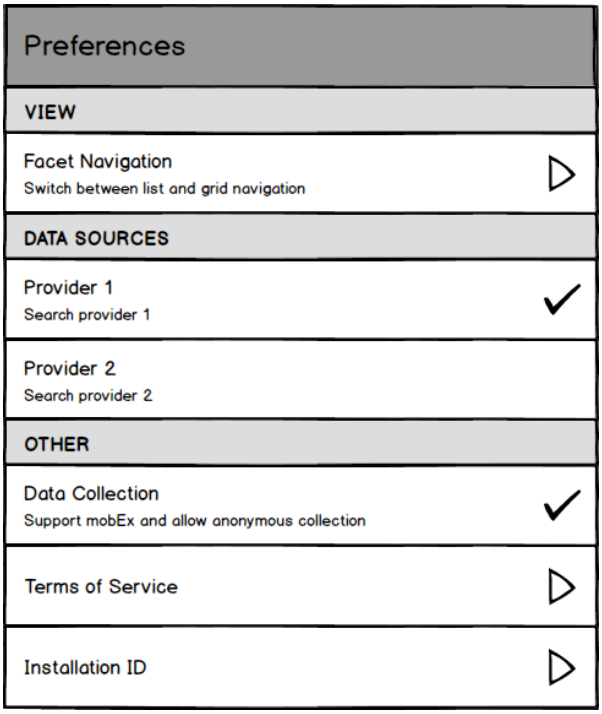


Figure 2.6.: Client preferences screen mockup

For practical reasons an additional component not explicitly mentioned in the architecture is included in the GUI. It has been omitted since it does not play a significant role in the component interaction but has a large number of connections to other components, introducing an unnecessarily high level of detail. The preferences screen, as shown in Figure 2.6, serves several purposes. Firstly, it should let the user allow to choose between the two facet display designs introduced in Section 2.1.2.1. Secondly, it should give them control over the kind of data they are interested in. Due to network traffic restrictions or lengthy provider response times they may wish to disable specific providers. The provider detail text, as depicted in the mockup, can be used to hint the kind of contents a specific provider offers. Thirdly, for legal reasons we are obliged to include the terms of use of several data providers, which can easily be done in the preferences screen. Fourthly the user should be able to grant us the permission

to collect data and also to revoke that permission at any time. Finally, for evaluation purposes of our official testers, we need to acquire their installation Identifiers (IDs), which should also be available as part of the preferences screen.

2.1.3. Package Structure

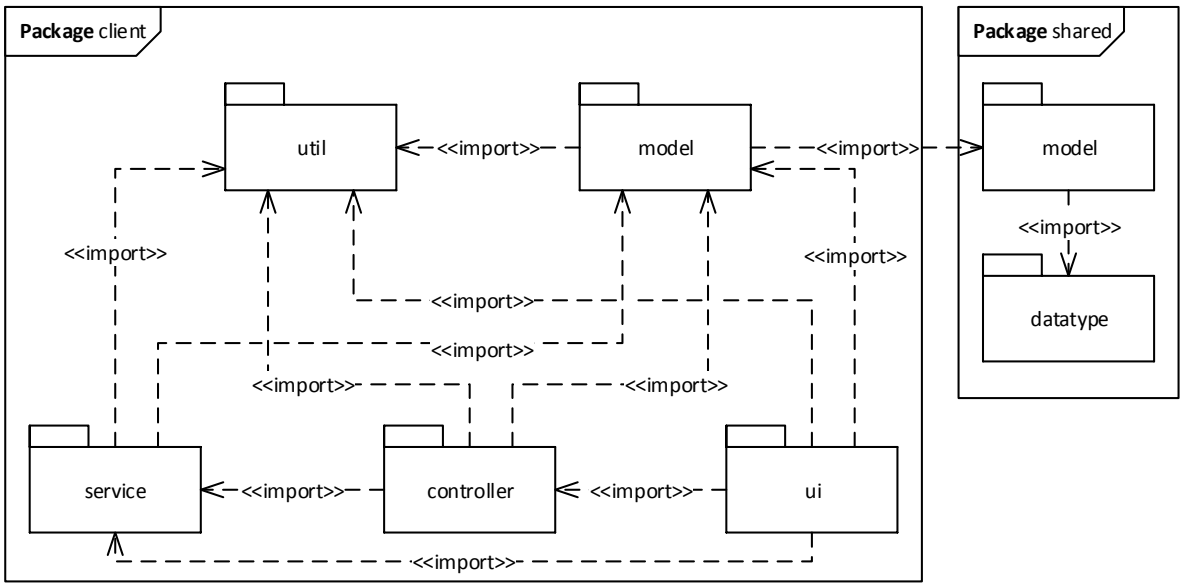


Figure 2.7.: Client package structure and dependencies

The client package structure reflects the components of a typical Model-View-Controller (MVC) application (see Section 3.1.1.2), complemented by a separate utility and a service package. From the implementation point of view it is important to note though that the Android architecture makes it difficult to separate views from controllers. This is partly due to the highly imperative approach of data binding and event handling. For this reason GUI-centric behavior, such as direct event handling, view building and transformation is handled directly by the `view` components. Data-centric behavior, such as the data loading process, is implemented by the `controller` components.

More generally, the `client.model` package contains the client-side implementation of the shared data model, which is the reason why it depends on the `shared.model` package. All of the other components, except for the general-purpose and therefore decoupled utilities, depend on the client-side model definitions in order to load (`service`), manage (`controller`) and display (`ui`) data. The second largely depended-upon component is the collection of utilities in the `utility` package. It defines a variety of general functionality, e.g., for location- and time-based calculations, collections, logging and native resource management. In the `service` package all data-related, asynchronous tasks are collected. The controllers and the user interface depend on it for loading events and facets, assigning the former to the latter and for other time-consuming tasks, such as the map marker clustering. In the package `controller`, the actual management of facets, events and notifications is defined. This includes triggering the

data loading services, incorporating the responses in the currently managed models and notifying other application components about the results. Finally the view implementation of the mobEx client is in the `ui` package, which depends on the data management defined in the `controller` package.

2.2. Implementation

In this section the implementation of the design discussed on the previous pages is explained. Here the focus lies on behavioral and implementation-specific details rather than general concepts. First the underlying data model is introduced. Then third-party dependencies are listed, along with their license-based liabilities. The third subsection deals with the interaction between the client and the server, with an emphasis on the back-end service consumption. In contrast, afterwards the interaction between the client-internal components presented in Section 2.1 is discussed. Finally the GUI details are outlined.

2.2.1. Modeling

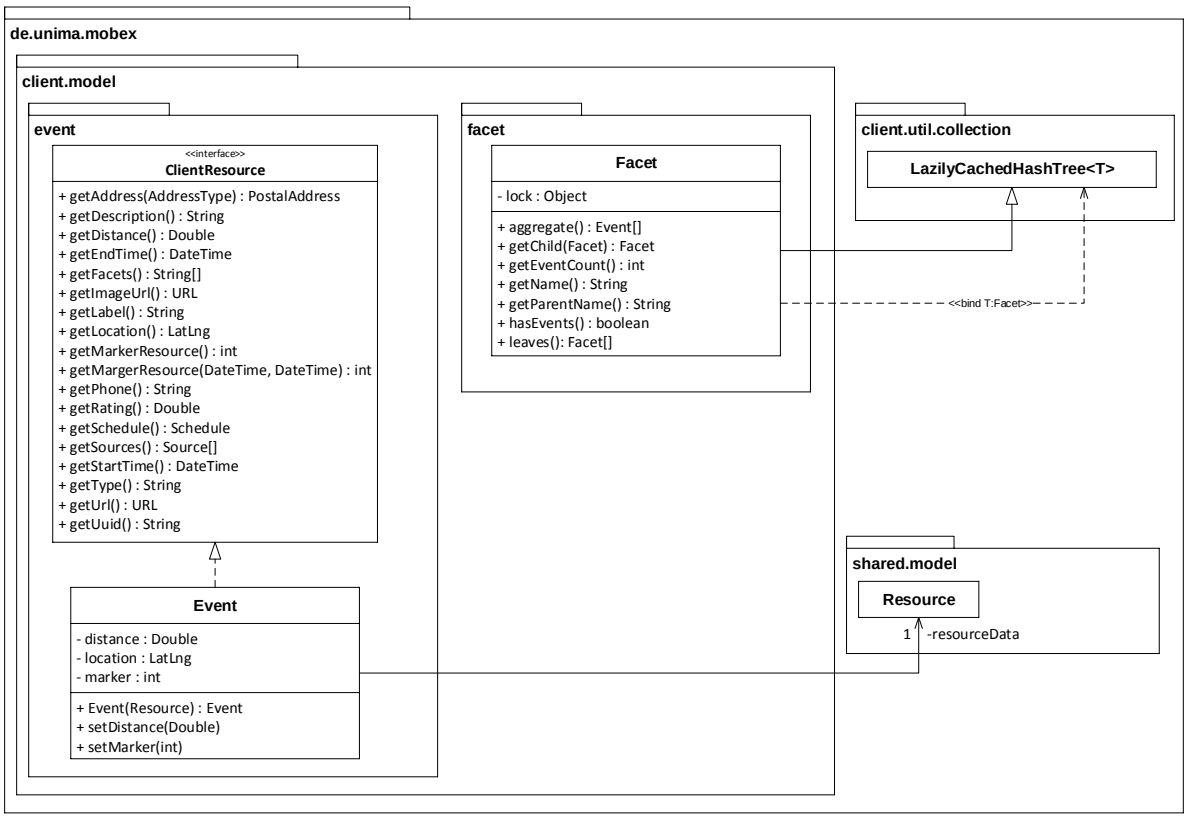


Figure 2.8.: Client data model

The client event model largely resembles the server resource model described in Section 3.2.1.1. In fact for the actual data exchange the server and the client share the classes in the

package `de.unima.mobex.shared`. The main difference between the server-side resource and the client-side event is that the app only requires a largely immutable object, since most data is only read, but never changed. This requirement is reflected by the CLIENTRESOURCE interface, which defines the publicly accessible methods needs to have in order to be displayed in the application. Figure 2.8 contains a Unified Modeling Language (UML)-diagram of this interface and lists its methods. Most of them are self-explanatory, such as *getDescription()*, *getLabel()* and *getLocation()*, so these are going to be excluded from the detailed descriptions on these pages. The less obvious methods are explained below.

Its main implementing class EVENT contains a reference to a RESOURCE object and thus serves as an adapter between the client-side interface and the server-side implementation. This approach was taken due to its superior flexibility compared to inheritance.

*getAddress(*ADDRESSTYPE) – An event may come with a number of places associated with it, each of which may be either a general, birth or death place. This suggests that the distinction between address types is mainly applicable to events in the category *person* and for others birth and death place are not set.

getDistance() – The distance returned by this method is always relative to the search location, i.e. the center of the current map viewport. The distance of an event is only set once, since relocating the viewport triggers a new search, and triggering a new search erases all previous search results.

getFacets() – The data exchange between client and server is separated into two services, the facet and the event service (see Section 2.2.3). This means that whenever events are loaded from the server they cannot be automatically be included in the facet structure as they are loaded. Instead, the facets they should occur in are listed by name and can be accessed using this method. The client then needs to assign them to the actual facet structure using this information.

getLocation() – The Google Maps library expresses the location of an event in terms of LATLNG pairs. This value is constructed by the EVENT class upon construction based on the latitude and longitude values of the underlying RESOURCE object.

getMarkerResource() – This method was overloaded for time-based filtering. Generally the marker resource ID is obtained by the version without parameters, which is the case when an event does not have time information attached to it. If there is time information available though, the other version is used and the resource should return a reference to a less opaque marker the further it is away in time from the specified start and end date.

getType() – The type of an event always refers to its top-level category, i.e. *person, place, event* or *organization*. It is used as a fall-back option in case one of the facets it specifies is not part of the current facet structure. In this situation, the facet will be created as a child of the corresponding top-level category.

For the facet structure a simple tree structure based on HASHSETs was implemented and later optimized using lazy property caching for performance reasons. In this section the purpose of the rather specific operations of the FACET class are highlighted, since the inherited methods are typical for regular (non-poly-)trees. Since the tree implementation is separated from the data it encapsulates, FACETs represent the tree *structure* and the *data* of each node is represented by a FACETDESCRIPTION instance attached to it.

`aggregate()` – This method provides access to all events assigned to facet subtree, represented by a specific node. It recursively gathers all events assigned to each individual subcategory and returns them as a set, since each event may be assigned to a number of different facets.

`getChild(Facet)` – It may appear strange to look for a specific facet in a facet tree using a specific facet, but this is due to how facet identity is defined in the application. In facet the FACET parameter serves as a template checking each of the facets in the subtree for equality. In the tree implementation used two nodes are equal whenever their contents, i.e. the underlying FACETDESCRIPTIONS, are equal, which is the case when their names match. That way this method may be used to obtain a facet which replaces the specified one due to deserialization, e.g., when the facet tree is reloaded from the server.

`leaves()` – This method returns all leaves in the subtree, which is mainly used by the facet selection management since non-leaf nodes are never directly selected. Instead, they become part of a selection, because they are part of a selected path, i.e. the path from the root facet to each individual child.

2.2.2. External Libraries

Library	Version	Developer	License
ActionBarSherlock	4.2[1]	Jake Wharton	Apache License, 2.0
Google Analytics Library	3.0.0[2]	Google Inc.	Apache License, 2.0
Android Support Library	4[3]	Google Inc.	Apache License, 2.0
Easing Interpolator	1.0[4]	Robert Penner	Apache License, 2.0
Gson	2.2	Google Inc.	Apache License, 2.0
Joda Time	2.1	Joda	Apache License, 2.0
SimpleViewPager	0.1	Michael Jess[5]	Apache License, 2.0
Wheel Widget	1.0[6]	Alessandro Crugnola	Apache License, 2.0

Table 2.1.: Overview of the external libraries

ActionBarSherlock

The ActionBar is an Android User Interface component that was introduced with Android version 3.0. It offers a consistent view element on top of each screen for navigating within

[1] `http://actionbarsherlock.com/`
[2] `https://developers.google.com/analytics/devguides/collection/android/resources/`
[3] `http://developer.android.com/tools/support-library/setup.html`
[4] The source code is freely distributed on the web and has been packaged under the assumed version number 1.0 for mobEx.
[5] This component was developed specifically for mobEx but distributed separately due to its high reusability.
[6] `https://github.com/sephiroth74/AndroidWheel`

the app, indicating the user's location in the app and making key functions easily accessible. ActionBarSherlock is a library that offers this feature for versions prior to 3.0.

Google Analytics Library

This library provides methods for the communication with Google Analytics. Google Analytics is a free service that helps to analyze user traffic and can be customized to capture individual events. The data can be examined via a web application. As long as a user has not opted out of logging and transferring the data to Google Analytics, every activity such as a click on a button or the interaction with the time wheel results in an event that is logged. We used the logging to get insights about how mobEx is used during a field study.

Android Support Library

ActionBarSherlock requires the Android Support Library to work. The purpose of this library is to provide backwards compatible features and APIs to an Android application. This way developers can target also older versions of Android while still using some of the features of newer versions.

Easing Interpolator

Interpolators are used to encapsulate the animation timing of GUI animations. For those purposes a number of interpolators was published on the Internet freely available for use on Android. The facet selection pull-down menu uses an easing interpolator of this collection for its automatic opening mode, i.e. the animation when the open button is pressed. This interpolator is characterized by a swift start and a smooth animation end. In the first application version (MFacets) this component used to be included as a source code file along with a large number of others, but unused interpolators without any hint about their origin. In mobEx the superfluous components were removed and the easing interpolator was moved to a separate library.

GSON

Gson is an easy-to-use (de-)serialization tool for the JSON format. Its main advantage over its competitors is that it uses reflection to automatically map JSON attributes to Java object attributes and the other way around. In mobEx it is used for deserializing the facet tree without the need for a cluttered parser class or any explicit annotation-based mapping.

JodaTime

Not only is the default Java date- and time-centric functionality cumbersome to use, it also lacks support for easily computing differences between points in time and shifting those values by a given amount of, e.g., minutes or days. This is the great strength of the JodaTime framework, which is the reason why it is used in the mobEx time management where adding arbitrary values to the time selection and computing the difference between points in time

is crucial. By removing a lot of the regular Java date and time complexity from the client JodaTime makes the corresponding code a lot more error proof and easier to maintain.

SimpleViewPager

For the tutorial class an easy-to-implement view pager was required. Unfortunately standard Android does not come with a component that supports a declarative approach of designing static view pagers, as it is necessary for the user tutorial. For this reason a lightweight library was written that extracts the direct children from the XML view layout and registers them as separate pages, hence enabling declarative pager designs. Due to its high reusability and general applicability it has been release separate from mobEx for the Android development community to use.

Wheel Widget

The Wheel Widget is an view element that emulates a realistic wheel which can be spinned. We use it on the map screen to provide an easy possibility for users to move back and forth in time.

2.2.3. Data Querying

In order to be able to display data to the user, the client has to query the server for both an initial taxonomy (or *facets*) in which events, such as Point Of Interests (POIs), are organized and the events themselves. The following section deals with the process of facet and event querying, the query triggers and the further technical processing of the result data.

2.2.3.1. Facet Querying

Since the taxonomy under which events are categorized is likely to change at the same pace as the provider databases themselves, hard-coding or otherwise distributing a fixed taxonomy with the client application releases would not have been a reasonable approach. Instead the existing taxonomy services offered and maintained by the data providers are leveraged, providing the client with a large and solid basic categorization for most of the events. These taxonomy services are not consumed directly by the client however but encapsulated by the REST facet structure service (see Section 3.2.3.2), for centralized translation, integration and caching. This service is only called once on application startup in order to guarantee a sufficiently up-to-date set of categories for the subsequent application use.

The raw facet tree from the server remains largely untouched as soon as it was loaded. There are two exceptions to this, however:

- The FACETMANAGER will add a new facet to the taxonomy directly under the top-level category of an event (one of *EventIn terms of concerts etc.*, *Person*, *Place* or *Organization*) for each facet the event should be assigned to but which is not yet part of the tree. For instance, if a person should be assigned to the *Actors* category but there is none in

the facet tree, a new facet called *Various Actors* will be created for that category under *Person*.

- For usability reasons events will never be assigned to facet tree nodes other than leaves. This is mainly due to the fact that a single node within the facet tree cannot be selected without also selecting all of its children. Instead, if for instance an event should be assigned directly to the top-level facet *Person*, the FACETMANAGER would create the new leaf *Various Persons* under *Person* and assign the event to the new category. For a more detailed description of the assignment process, see Section 2.2.4.5.

2.2.3.2. Resource Querying

The client-server interface for location-based data access was designed with the necessity of entity resolution and short user response times in mind. Since many providers are queried by the server at the same time, it is likely that the data overlaps to some extent. Merging the information of duplicates is only possible when all data is available. On the other hand, there is a considerable provider response time variance and the user would have to wait unacceptably long. For that reason the interface was designed for making results available as soon as possible, sending batches of events as they become available rather than waiting for the response of all providers.

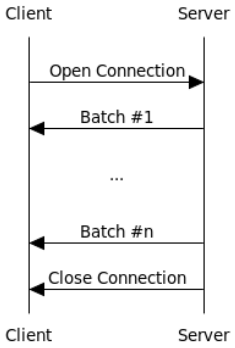


Figure 2.9.: Client-server-communication

Figure 2.9 illustrates the implementation of the client-server-interface. Whenever the client application detects the need for a data update, for instance, due to map activity or user movement, a new RESTRESOURCEPROVIDER (see Section 3.2.3.1) request is created, opening a connection to the server, which in turn queries the data providers. For a detailed list of the request parameters, please refer to Section 3.2.3.1. As each of the providers responds, batches of intermediary results are returned, consisting of *insertions* and *deletions*. Insertions typically refer to new events that were found, while deletions should be removed from the client because the entity resolution made them obsolete.

The EVENTMANAGER[3] keeps track of all of the deletions in a deletion set. Events in the deletion set, identified by a UID, will

[3]Due to the time-sensitive nature of the data the application deals with all events are commonly called *events*, hence the name of this component.

1. be removed from the currently managed events if they were part of it when they where added

2. never be added again, even if another (later) response specifically includes it in its list of insertions.

For technical reasons the latter step is of particular importance. It is possible that a batch of events is sent to the server and directly afterwards part of the entities is resolved, resulting in another batch of events. If due to some delay the second batch arrives prior to the first, nothing would be removed because the events the deletions refer to did not yet arrive. Only when the delayed batch arrives they would be added, resulting in displaying invalid information. This issue can be clarified by the following example: The server concurrently performs entity resolution on two sets of events after receiving new data from a provider. The first thread identifies a duplicate, sending the client the instruction to add a new event containing the merged information. At the same time the second thread discovers that this new merged event is a duplicate, too. This will result in sending the instruction to remove it. Now it is not certain which of both instructions would reach the client first, due to the highly concurrent nature of the entity resolution. If the insertion instruction arrives after the deletion instruction, this can be resolved because the client keeps track of events to remove in the deletion set. In this case the client would check if it was told to delete the event before and not insert it at all.

Finally, when the set of managed events was adjusted by the insertions and deletions in the currently processed batch, an EventAssignmentService is used to concurrently assign the events to the facets they specify. As described in the previous subsection, this may lead to new facets being added to the taxonomy. The assignment process is described in detail in Section 2.2.4.5.

2.2.4. Component Interaction

In this section, the component interaction is described using the example of starting the application, which triggers an almost complete round trip through the application. Please note that most messages in the sequence diagrams in this section represent method calls, but those without the parameter part enclosed in round brackets do not. Also, most actors refer to actual classes, but those not written in camel case do not. This was done in order to reduce the level of detail in the diagrams.

2.2.4.1. Application Start

The sequence diagram in Figure 2.10 illustrates the basic process of starting the application. First the application component MobEx is instantiated by the Android framework. It creates the main controllers of the application: The FacetManager, EventManager and EventEmitter (notification management in the architecture diagram). In addition it implements the SharedApplicationState interface, thus serving as the location management discussed in Section 2.1.1.2. Finally it also grants access to the application preferences, which are used to control which kind of facet navigation to display (see Sections 2.1.2.1 and 2.2.5.2) and

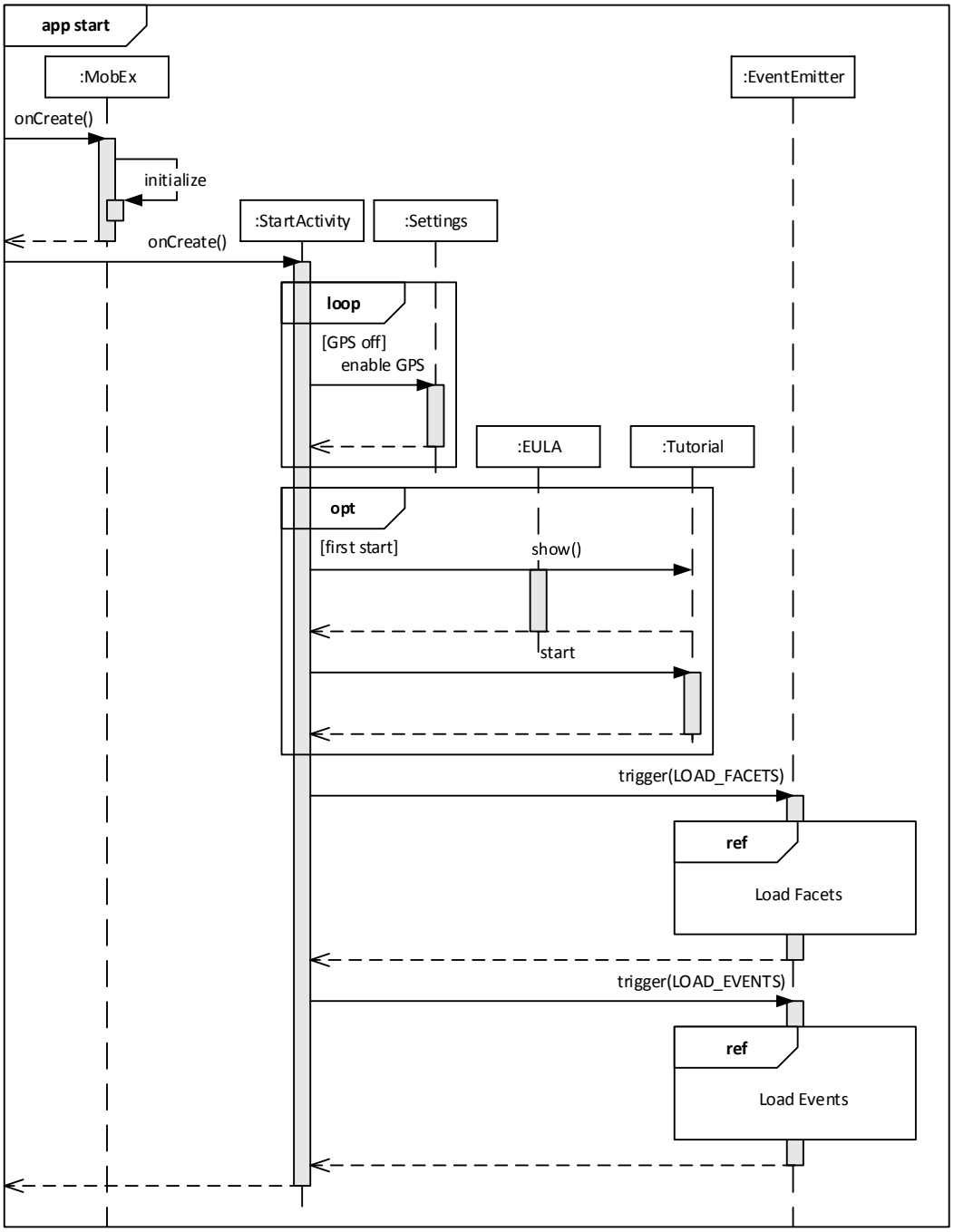


Figure 2.10.: Sequence diagram: Application start

what providers to query for events. Originally in MFacets[13], the mobEx predecessor, the application component did not exist and the managers were implemented as singletons. This generally is a bad design approach since singletons introduce a global state in the form of static variables, hampering application testability. For this reason the follow-up team refrained from this approach. For future development, this decision should be taken into account.

When the application initialization is finished, the Android framework creates the application main ACTIVITY, the STARTACTIVITY. It checks if the location service was enabled by the user, since it may be turned off in the settings. If it is not enabled, a dialog is displayed, offering the user to open the Android preferences for changing the settings. The application is closed when the user refuses to do so.

If the application is started for the first time, both the End User License Agreement (EULA) and the tutorial are displayed. The former asks the users to grant their permission for anonymous data collection, while the latter seeks to familiarize the user with the main application features. Afterwards a flag is set in the preferences indicating that the application was started before, skipping the EULA and tutorial in future usages.

As soon as the EULA and tutorial are dismissed or if they were not displayed at all, two events are published using the EVENTEMITTER: One triggering the facet loading, and another triggering the event loading process.

2.2.4.2. Notification Management

In mobEx, the notification component discussed in Section 2.1.1.2 is implemented by the EVENTEMITTER class. It allows registering listeners to a specific type of event with a specific type of data associated with that event. Usually if generic data is associated with an event, the listeners would implement a method in the form of *on(*`OBJECT data`), requiring them to cast the data depending on the event type. For instance, assuming class `A` expects an event to occur with an instance of `B` attached to it as `data`, it would need check its type and cast it. This must be done for each listener expecting data, multiplying the corresponding type-checking code. By parameterizing the listener interface with the expected data type, this behavior could be moved to the notification management. The NOTIFICATION class represents a single notification to a single listener and casts the argument as required. If casting fails, the notification is not delivered and a warning message is logged.

Furthermore the EVENTEMITTER uses an Android HANDLER to post the NOTIFICATION to the message queue of the GUI thread. This has a great advantage over the regular observer pattern in that it decouples the actual notification from the loop in which all handlers are notified. In the traditional approach, an exception from one of the handlers would force the loop to exit unless handled properly. Unfortunately the EVENTEMITTER deals with generic events and data and thus lacks the information to deal with individual types of failure. Even if it could, this would introduce dependencies to specific listeners, which reduces the overall system maintainability. Furthermore by calling listeners directly, execution control is also passed to them. If it takes a listener five minutes to return, then the component calling the notification management must wait that time until it can proceed.

2.2.4.3. Loading Facets

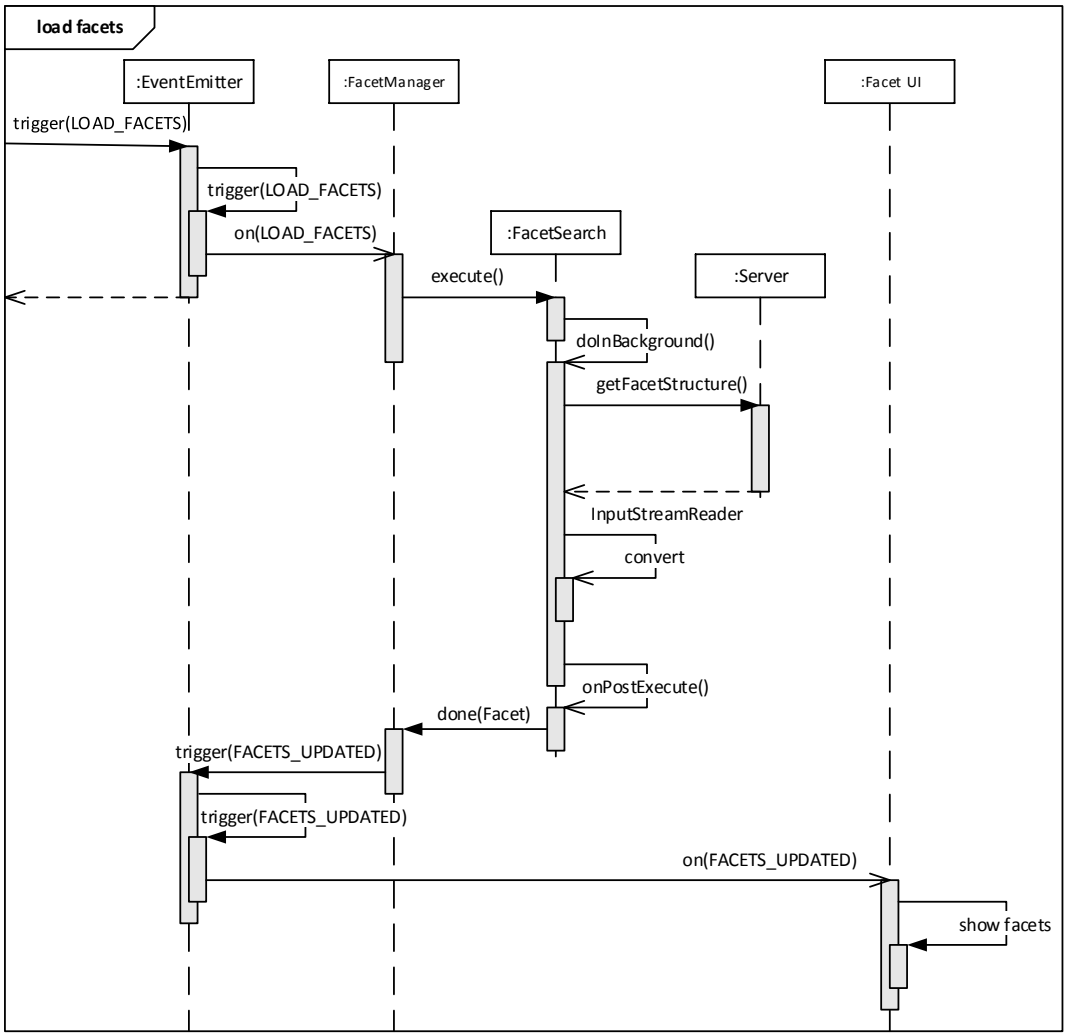


Figure 2.11.: Sequence diagram: Loading facets

The facet loading process can only be triggered by a specific notification, the APPLICATIONEVENT.SHOULD_LOAD_FACETS event. Figure 2.11 contains a description in the form of a sequence diagram. As described in the previous section, the FACETMANAGER is notified asynchronously via its *on(ApplicationEvent, Object)*[4] method. It will then create a new FACETSEARCH instance, which in turn creates a new SERVER instance, representing the server-side web services. When the stream representing the facet end point is available, the JSON data transferred is converted into the client-side facet tree structure. Finally the control flow returns to the GUI thread and passes the results back to the FACETMANAGER via its *done(Facet)* method. The FACETMANAGER updates its internal data structure and triggers the FACETS_UPDATES notification, which updates the facet GUI components.

[4]The FACETMANAGER does not expect data with events, so it implements the generic OBJECT interface.

2.2.4.4. Loading Events

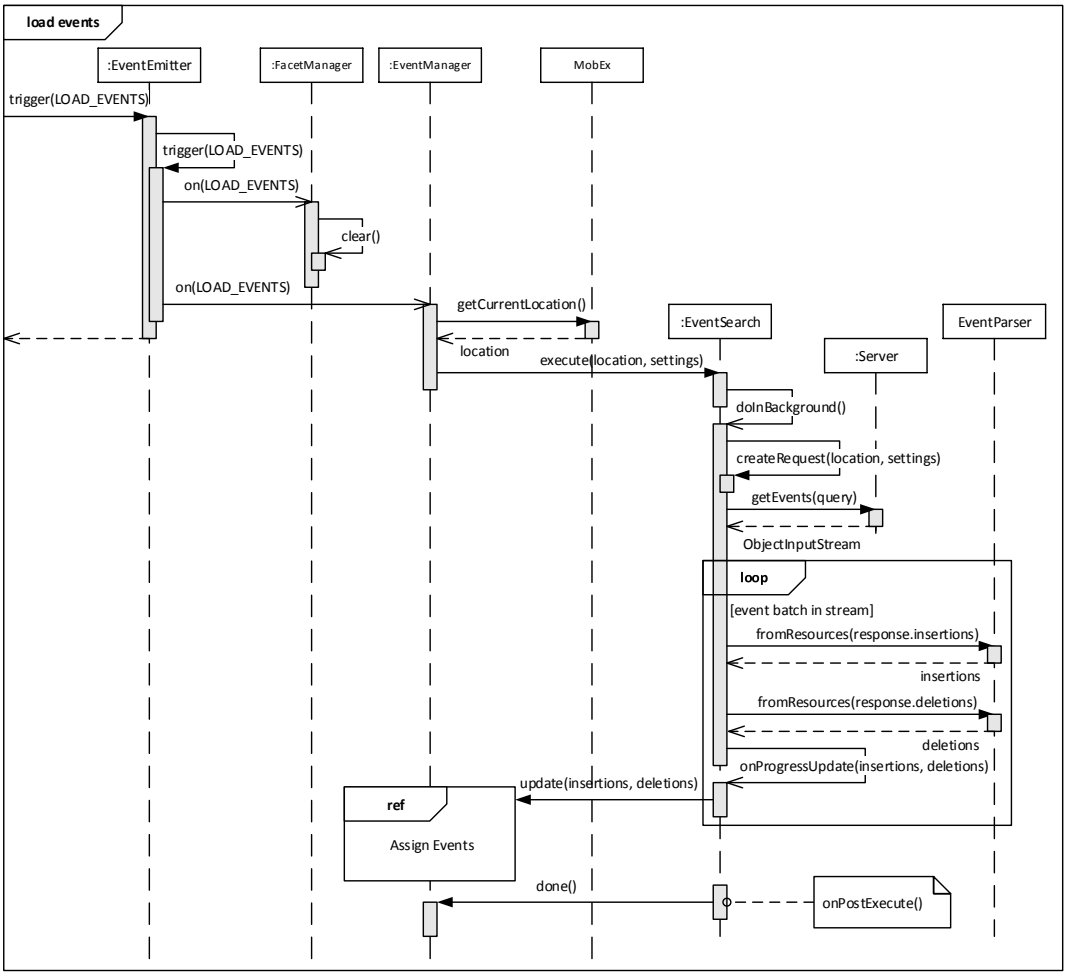


Figure 2.12.: Sequence diagram: Loading events

In contrast to the facet loading process, the event loading process shown in Figure 2.12 can be triggered via the SHOULD_LOAD_EVENTS notification in a number of situations:

- When the map viewport changes
- When the application is resumed or started
- When the provider preferences are changed
- When the user triggers a manual data reload

Both the FACETMANAGER and EVENTMANAGER listen for this kind of notification. The FACETMANAGER clears the facet tree of the currently assigned events, resetting the number of events assigned to each facet to zero. The EVENTMANAGER on the other hand obtains the current location and preferences from the SHAREDAPPLICATIONSTATE implemented by MOBEX

and starts a new EventSearch with that information. Much like the FacetSearch described in the previous section, the actual data stream is obtained by an instance of Server, which deals with the connection details. The main difference here is that while the FacetSearch accepts a single JSON representation, the EventSearch loops over the stream. For each batch of events that is transmit via it, a new set of event insertions and deletions is created using an EventParser. The parser is not only responsible for transforming the resource data into the client-side data model, but also calculates and sets the distance and the map icon for each event returned. When one batch of events was received that way, an update is sent to the EventManager class using the *onProgressUpdate()* method on the GUI thread. This triggers the assignment process described in the next section. In additon, as soon as all event batches are loaded, the *done()* method of the EventManger is called, triggering another notification that the data loading process finished. This notification is used to stop the loading indicator animation in the GUI.

2.2.4.5. Assigning Events

For each batch of events transmitted from the server to the client, which is consisting of a number of *insertions* and *deletions*, an instance of the update process depicted in Figure 2.13 is started. The actual assignment process is outsourced into a concurrent component, the EventAssignmentService. This is necessary because the facet and event loading process are started concurrently and the EventManager cannot assume that the facet structure already exists when the first batches of event data arrive. For that reason the EventAssignmentService running in the background acquires a lock on the FacetManager using *wait()*, which is only released when the facet structure was loaded from the server. This lock prevents calls to the *doWork()* method before the FacetManager is capable of assigning events to facets.

Independently of the EventAssignmentService waiting for the FacetManager, the EventManager may always receive batches of events for processing. First the currently managed *deletion set* is extended by the deletions added in the current *update()* call. Then, correspondingly, the currently managed events are extended by the insertions defined in the currently processed batch. Since the server may always update the information associated with any event, this may also lead to replacements in rather than additions to the current data set.

Regardless of the events managed by the EventManager, some that should be deleted may still be assigned to facets. For that reason the FacetManager is told to unassign (remove) those from the facet tree. Also, due to, e.g., network latency, the client may receive insertions that are already in the deletion set. For that reason all events that are in the deletion set are stripped from the insertions. The remaining items are pushed onto the blocking queue of the EventAssignmentService.

As soon as the facet structure is available, releasing the lock preventing the EventAssignmentService from processing the event batches, it passes each single event to the FacetManager. In its *assign(*Event) method, it iterates over the facet names attached to the event in order to assign it to all of them. If the facet already exists, the event is just added to it, unless it is not a leaf. This is due to usability requirements discussed in Section 2.2.5.2. If the facet was not a leaf, a new leaf will be created under it with the same name, prefixed

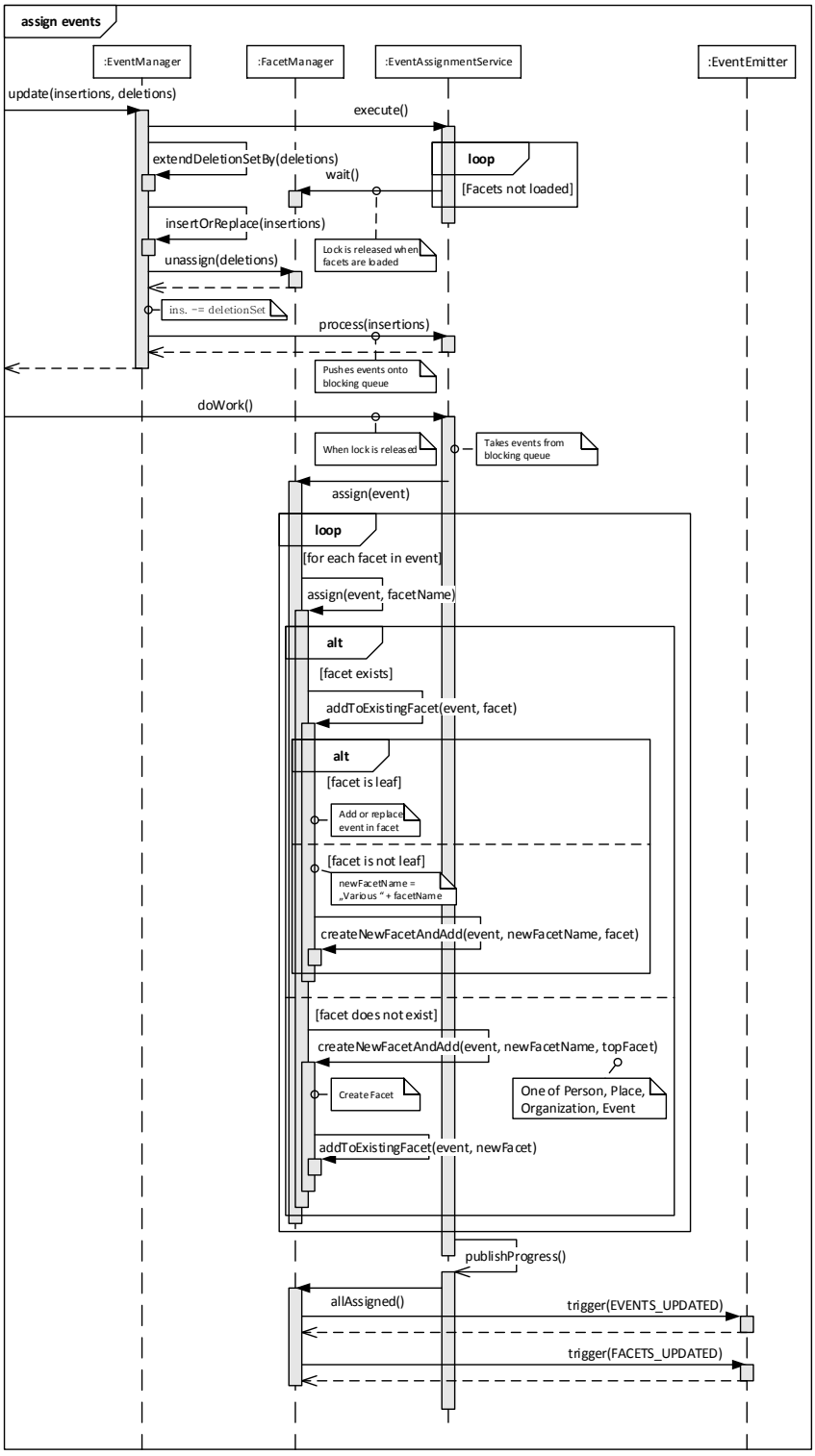


Figure 2.13.: Sequence diagram: Assigning events

by the facet name *Various*. For instance, if an event should be assigned to the *Actors* facet but Actors is not a leaf in the facet tree, a new leaf called *Various Actors* would be created under it and the event would be assigned to the new child.

If the facet the event should be assigned to does not exist in the facet tree at all, it must be created as well. In that case the `type` attribute of the event is leveraged, which refers to one of *Event*, *Organization*, *Person* or *Place*. The new facet will be created and placed under that top-level category. For instance, an event of type Place should be assigned to the facet *Dome*, but Dome does not yet exist in the tree. The issue is solved by creating a new facet called Dome directly under Place and assigning the event to it. That way the facet structure may be extended dynamically on the client depending on the event data that is sent to it.

2.2.5. User Interface

In this section the implementation details of the GUI components will be discussed. The first subsection deals with the implementation of the components bundled in the FacetActivity class, while the second focuses on the MapActivity behavior. For a general description of the goals and specification of the components, please refer to Section 2.1.2.

2.2.5.1. Event Display

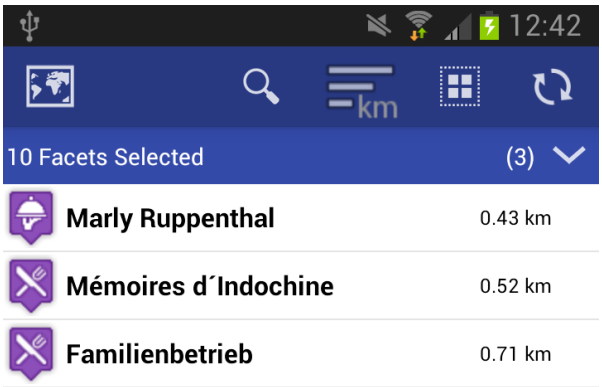


Figure 2.14.: Screenshot of the event list

Figure 2.14 shows an implementation screenshot of the event display discussed in Section 2.1.2.1. As can be seen when comparing the design mockup and the screenshot, the latter is an exact implementation of the drafts. For this reason the statements and descriptions of the aforementioned section apply. In the list all events are listed, filtered by the current facet selection and an optional query string. The query may be entered by pressing the magnifying glass icon in the action bar on the top corner of the screen, which replaces the icons with a search input. In contrast to the default ArrayFilter implementation of the Android ArrayAdapter class, the filter implemented does not require the list entry to begin with the query. Instead a case-ignoring comparison using String.*contains*(String) is made, requiring the keyword to appear anywhere in the list entry. The distance to the right of the event name is set to a fixed value and computed when parsing the server response (see Section 2.2.4.4). This means that it

will not change when the users move but is based on the location where the event search was triggered from. All other event information is available from the details screen implemented in the EventInspectorActivity, which is opened by tapping on an event on the map or the event list.

2.2.5.2. Facet Display

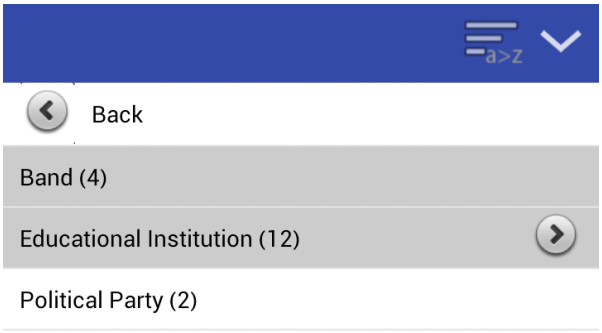


Figure 2.15.: Screenshot of the facet display list layout

Figure 2.15 shows the expanded facet display navigation area with the list layout. The top-leftmost button is used to toggle the facet sorting mode between alphabetic order and order by number of events per facet. The sorting mode is set in the FacetManager for centralized access, i.e. the FacetManager provides the single point of access for facet data and controls its ordering at the same time. This way it is guaranteed that the correct ordering is always applied. The chevron next to the order button is used to expand and collapse the facet navigation area and hiding the facet order button. While hiding the button is not necessary, it was implemented this way because in collapsed state it serves no purpose and could potentially confuse the user, since it has no visible effect. Finally the second button from the right in the action bar shown in Figure 2.14 is used to toggle the selection state of the currently displayed facets. If any facets are selected, it will deselect all of the currently visible facets. Only if none are selected, it will select them.

A list of the current selection is available from the facet drawer, which can be opened by dragging down the handle below the action bar or by pressing the chevron to the right. As discussed in the facet display design section, single facets can be deselected by pressing the red cross next to them. For the sake of usability the first and the last entry of the facet selection list may be used to clear the whole selection.

Since the actual Android SlidingDrawer was deprecated in Application Programming Interface (API) version 17, mobEx comes with a custom implementation of that widget. This component is the only one left largely untouched when MFacets was refactored to form a solid basis for mobEx, the reason being poorly documented but rather complex code. Even though it appeared pretty stable at that time, the animation code seems to fail at times when the drawer is opened slowly and the handle is released half-way.

Implementation-wise, the facet list is a lot simpler than the facet grid showed in Figure 2.17, e.g., in that it does not need to deal with facet overflows (*More Facets...*). Since basically its

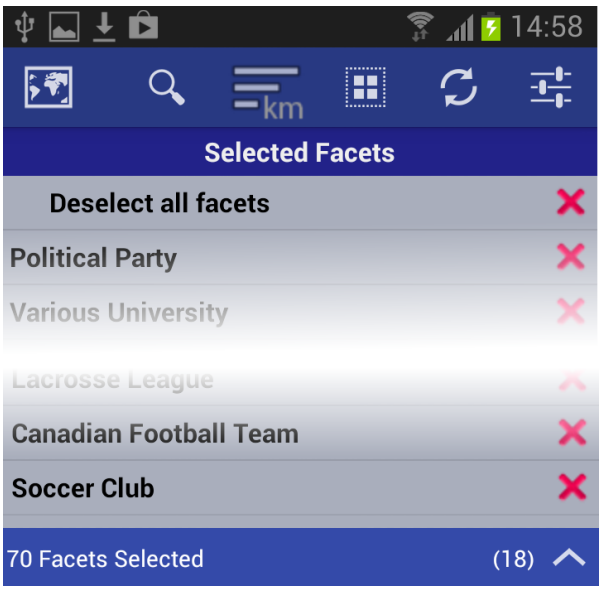


Figure 2.16.: Screenshot of the facet selection drawer

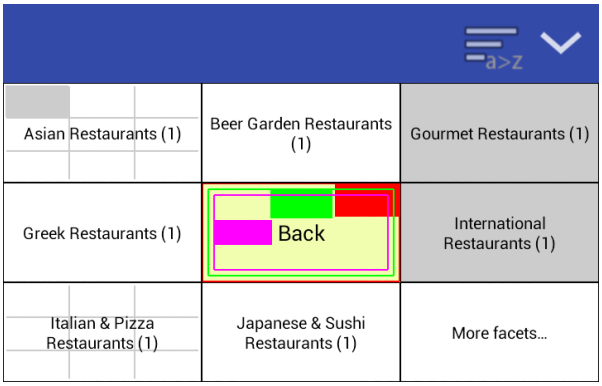


Figure 2.17.: Screenshot of the facet display grid layout

implementation is not different to any Android list view, its internals will not be discussed here.

Generally, the facet navigation area is an implementation of the Gang of Four (GoF) Bridge pattern [15] in order to let the user exchange the implementation easily: The actual interface exposed to its client is defined in the interface FACETNAVIGATION, which exposes methods such as navigting up and down in the facet tree. This interface is implemented by the FACETNAVIGATIONAREA, a base class implementing some basic functionality, such as expanding and collapsing the facet display. For the grid-specific and list-specific behavior it holds a reference to one of its subclasses, the FACETGRID or FACETLIST. This behavior is encapsulated in these implementations and the FACETNAVIGATIONAREA simply delegates method calls such as showing a facet to the grid and list implementations. Since the Bridge pattern excels at separating interfaces from actual implementations, switching to another type of facet display is as easy as exchanging an object reference.

The navigation type the user is confronted with initially is determined randomly when the application is first started. That way the users' preferences could be analyzed by checking if they switched to another type of navigation in the preferences screen.

Furthermore it is important to note that in both facet display layouts no event is ever assigned to a non-leaf facet for usability reasons. In fact clicking a node other than a leaf only opens it, and even long presses only selected all leaves in the selected facet subtree. Generally it is not possible for the user to select a specific node but *not its children*, which means allowing the assignment of events to regular facet nodes either restricts the users' control over the application or increases the complexity: Either the node *and* its children must be selected, e,g, by a long press, or another gesture must be introduced for selecting nodes only. This, however, makes it difficult to highlight the selection state of such a node in the user interface, making the application less intuitive. The solution to this issue is rather simple: Instead of assigning events to non-leaf nodes, a new leaf is created under the node in question. The new facet is called the same as its parent, except that it is prefixed by *Various*. For instance, when an event should be assigned to the facet *Artists* and Artists is not a leaf, a new leaf called *Various Artists* is created as its child and the event is assigned to that leaf. This way the user may always assign the whole subtree including the new leaf as usual using a long press, but also select the contents of the node in question only by selecting its artificial child.

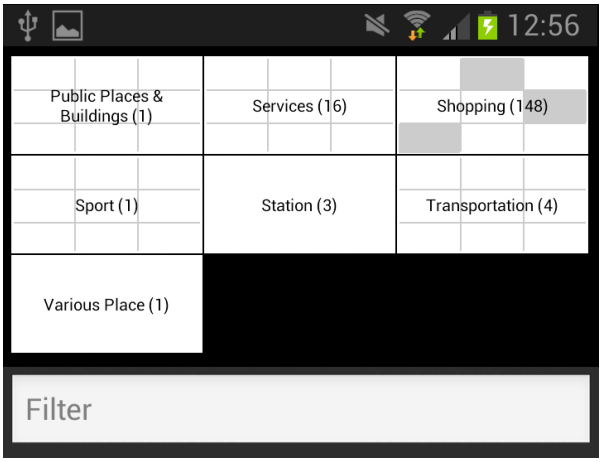


Figure 2.18.: Screenshot of the More Facets dialog

Figure 2.18 shows the implementation of the facet grid overflow, which comes into play when there are more than eight child facets to display. This is necessary because the facet grid layout only has enough room to display a maximum of eight children. In that case the eighth facet is replaced with an artificial *More Facets...* node as shown in Figure 2.17. Pressing this facet opens the facet grid overflow which also contains a text filter, since in some cases there is an excessive number of entries that might otherwise be difficult to handle for the user. However, selecting and opening facets in this view works just like in the regular facet grid. If a non-leaf node is chosen, the dialog will be closed and its children are displayed in the facet grid. Pressing the back button in the middle returns to its parent as usual.

A detail not discussed in the facet grid design section was the appearance of facets with some of their children selected. Non-leaf facets have a grid displayed in their background

indicating that they have children. If one of it is selected, the corresponding grid position is painted in gray, as can be seen in the Figures 2.17 and 2.18. Since the facet ordering is managed centrally in the FACETMANAGER and changes to the ordering trigger a FACETS_UPDATED notification, the highlighting of selected children always stays in sync with the actual child position according to the current ordering. Also, selected children in the overflow, if there is any, are correctly marked in the bottom-right corner, indicating that pressing the facet overflow button is necessary to view the selected children.

Not only is the selection state of children marked according to their position in the parent facet, but so is the current navigation path in the back button in the middle of the facet grid. In MFacets this control used to be restricted to three levels of depth, which was changed in mobEx. With increasing depth levels the inset of the border to draw is increased and a rectangle is drawn in the position corresponding to the parent facet clicked. The colors are generated using the Hue-Saturation-Value (HSV) color model [17], leveraging its cylindrical-coordinate character: For each level of depth, the color hue is shifted by 120 degrees or one third of the spectrum. Additionally, after three levels of depth, this shifted angle is offset by another sixth of the spectrum, resulting in a total of six different colors which will repeat after six levels of navigation.

2.2.5.3. Map Surface

The Map Surface is the part of the user interface where events are visually presented on a map by using map markers. The map allows easy exploration and discovery of places. In the following part, first the corresponding activity is introduced, followed by a detailed illustration of the clustering function that is used to avoid overcrowding of map markers. Figure 2.19 shows a screenshot of the map screen.

2.2.5.3.1. Map Activity The Map Activity is responsible for showing a map together with controls to change the current date. On the map, the events that are currently loaded are shown as map markers. A map marker consists of an icon that visualizes the type of the event and a label that is visible after a click on the marker. The icon of an event is determined by the corresponding facet of the event. The file `facetDrawableMapping.json` contains a mapping between facets and icons. For the map itself, we use the Android Google Maps API v2 that provides a feature rich map and allows easy integration into an Android application. The Map Activity also uses the ActionBar to make functions like setting the viewport of the map to the user's location or reloading events easily accessible.
One characteristic of the map activity is the integration of the time component. A date and time is not only visible at the top right of the map, it can also be changed in two different ways. Either by using the timewheel on the bottom of the map or by using the date picker that appears after a click on the date. The date picker is implemented as a dialog using the standard Android date picker component. The timewheel is an external library providing a view component that shows a scrollable wheel. In addition to the date and time, a timespan can be set via a small button next to the wheel that also shows the timespan that is currently selected. Possible values of the timespan are predefined in the `Timespan`-Enum and range from 0 minutes to seven days. After each change of the time or timespan, the events that are shown on the map behave

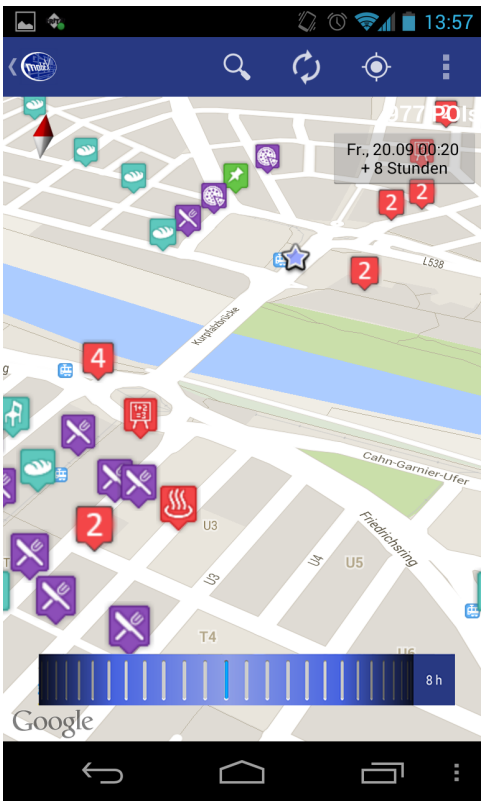


Figure 2.19.: Screenshot of the Map Activity

accordingly, e.g. change their icons or appear/disappear. This behavior is different depending on the type of event. A regular place that has no time information is always visible. An event that has a beginning and ending, like a concert, is shown as a star icon. The opacity of the star increases when changing the time closer to the start time of the event, see Figure 2.20. For locations with opening hours the icon is shown transparent when the location is closed.

2.2.5.3.2. Clustering When showing events and places on the map as so called map markers, soon the problem occurs that far too many markers are placed on the map, lowering the ability to overview and explore places. This situation occurs especially in city centers where many places of interest are situated or if the user does not filter the events in the facet activity prior to switching to the map. To solve this problem, the idea is to group map markers together based on their geographical distance. An alternative approach would be to only show a selection of the markers when too many markers are at one spot. By zooming in more and more markers can be made visible. However, this would require a ranking system that favors one marker over another and would also hide information from the user which is not a desired outcome.
The basic idea of clustering is to group items together based on their distance to each other on the screen. When zooming in, there is more space on the map, so more items can be shown. On the opposite, by zooming out, more items have to be grouped together to avoid overcrowd-

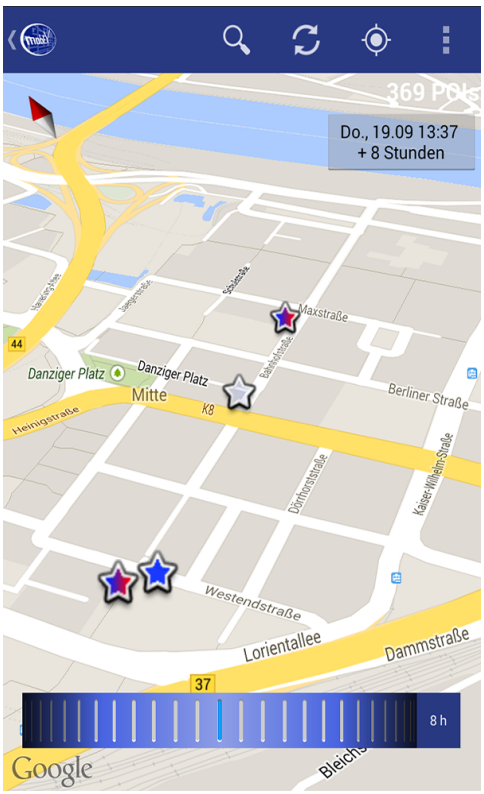


Figure 2.20.: Screenshot of the Map Activity showing events with different icons representing the temporal distance

ing the map. Thus, the clustering process has to be executed each time the user zooms in or out. The main requirement for the clustering function was a reasonable performance. The user should receive immediate results, even if the number of items is high. That is why the function had to be redesigned several times until a good performance had been achieved.
In an early implementation the distance between the items had been calculated as pixels on the screen. The advantage of this solution was the independence of the current map zoom level. The major downside was the performance of this function which was provided by the Google Android Maps API. The clustering process with a realistic number of items took about 5 seconds which was too long. Thus, we had to choose another solution. The two factors determining if two events should form a cluster are firstly, their geographical distance and secondly the current zoom level of the map. The implementation of this process is quite static. For several ranges of zoom levels, we defined threshold distances below which items will be clustered. Those values were defined by trying different thresholds for different zoom levels until a reasonable value was found. For instance, for a zoom level between 12 and 13 every two events that are closer than 14 kilometers from each other will be clustered. The corresponding Java class is *Clusterer.java*. This solution performs better and results are usually returned in less than a second, even when clustering a large amount of items. The algorithm compares all events with each other, resulting in a complexity of $O(log(n^2))$. In a simplified way, the algorithm starts with a stack of unclustered events and an empty list of clustered events. We take the top element from the stack of unclustered Events and compare it to all other events in

that stack. If the distance is close enough so that they would form a cluster, all those nearby events and the initial event itself are grouped to a new `ClusteredResource` object that holds references to all events in that cluster and that is added to the list of clustered events. Moreover those events are removed from the list of unclustered events to avoid further unnecessary comparisons. If an event has such a great distance to all other events so it does not form a cluster, it will be added as single event to the list of clustered resources. The algorithm terminates when the list of unclustered markers is empty. The pseudo code for this algorithm is shown in Algorithm 1.

Data: *unclusteredEvents*: a stack of unclustered events
Data: *threshold*: a threshold based on the current zoom level indicating the distance when items need to be clustered
Result: *clusteredEvents*: a list of clustered events
while *unclusteredEvents is not empty* **do**
  Take the first item from the *unclusteredEvents* Stack and assign it to *e* ;
  Create a new list *cluster* to collect events that need to be clustered ;
  foreach *t In unclusteredEvents* **do**
    distance← CalculateDistance(*e*, *t*);
    if *threshold* > *distance* **then**
      Add *t* to *cluster*;
      Remove *t* from *unclusteredEvents*;
    end
  end
  if *cluster is not empty* **then**
    Create a new ClusteredResource *c* ;
    Add references of *e* and all events in *cluster* to *c* ;
    Add *c* to *clusteredEvents* ;
  else
    Add *e* to *clusteredEvents* ;
  end
end
Return *clusteredEvents* ;

Algorithm 1: Clustering algorithm

2.2.6. Logging

Logging is an essential part of the mobEx application. With the logging functionality we aim to get insights about how mobEx is used and also conducted a quantitative user study. In general, after the user has done an activity like clicking on a button, we captured this action as an event that is logged and sent to a server where all information is held in a database and can be inspected and analyzed. We decided to use Google Analytics for this task as it offers benefits compared to an own implementation or other available tools on the market.

2.2.6.1. Google Analytics

Google Analytics is a free service that helps to analyze user traffic and can be customized to capture individual events. The data can be examined via a web application. As long as a user has not opted out of logging and transferring the data to Google Analytics, every activity such as a click on a button or the interaction with the time wheel results in an event that is logged. In order to use Google Analytics, an account has to be registered and a new Mobile Application Project created whereupon a Tracking ID is shown that is used to identify the application. On the client side, the Google Analytics library needs to be included and a configuration file with the Tracking ID created.

2.2.6.2. Implementation

There are different kinds of data that can be sent to Google Analytics. Usually, every start and stop of an Activity is logged by calling a method of the library in the `onStart()` or `onStop()` method of the activity, see Listing 2.1.

Listing 2.1: Logging the start and stop of an activity

```
1  @Override
2   public void onStart() {
3     super.onStart();
4     EasyTracker.getInstance(this).activityStart(this);
5   }
6
7   @Override
8   public void onStop() {
9     super.onStop();
10    EasyTracker.getInstance(this).activityStop(this);
11  }
```

Furthermore, there are custom events that can be logged. An event consists of four fields that can be used to describe a user's interaction. Those are Category, Action, Label and Value. The field "Value" has the data type `Long`. All other fields are Strings. In mobEx we used the "Category" to indicate on which screen the event takes place. "Action" describes the type of action like a click on a button or an input of text. The "label" then identifies the UI element. The field "value" is rarely used as it makes only sense in some cases like to log the different time intervals in the map screen. An example of logging an event is shown in Listing 2.2.

Listing 2.2: Logging events

```
1  EasyTracker.getTracker().sendEvent("map_ui_action", "wheel_usage", "
       intervall_change", timespan);
```

Another type of logging is to sent custom metrics or dimensions to Google Analytics. In mobEx, we use custom dimensions to capture the User ID and to log if the user uses the list or grid navigation type during a session. Listing 2.3 shows a code snipped that logs a custom dimension. The method requires an ID that needs to be defined in the Google Analytics Web Interface and the value of the dimension.

Listing 2.3: Logging custom dimensions

```
1 EasyTracker.getTracker().setCustomDimension(3  ,facetNavigationType);
```

2.2.6.3. Continue to use Google Analytics

To further use Google Analytics in order to log user data, some steps are required. First, a Google Analytics account has to be registered. In this a account a new application project needs to be created to retrieve a Tracking ID that needs to be pasted in the config file of the app (res/values/analytics.xml). After that, most of the logging works, except of the custom dimension that have to be defined in the web interface. In "Admin -> Custom Definitions", the following custom dimensions have to be defined:

Name	Index	Scope
UserID	2	User
uses_list	3	Session

Table 2.2.: The custom dimensions that have to be defined in Google Analytics

2.2.6.4. Reporting

As already mentioned, the logged data can be accessed via the Webinterface of Google Analytics. Many basic reports that show usage statistics are already available. Figure 2.21 shows a Screenshot of the interface. On the left side, the main menu allows for navigating through the

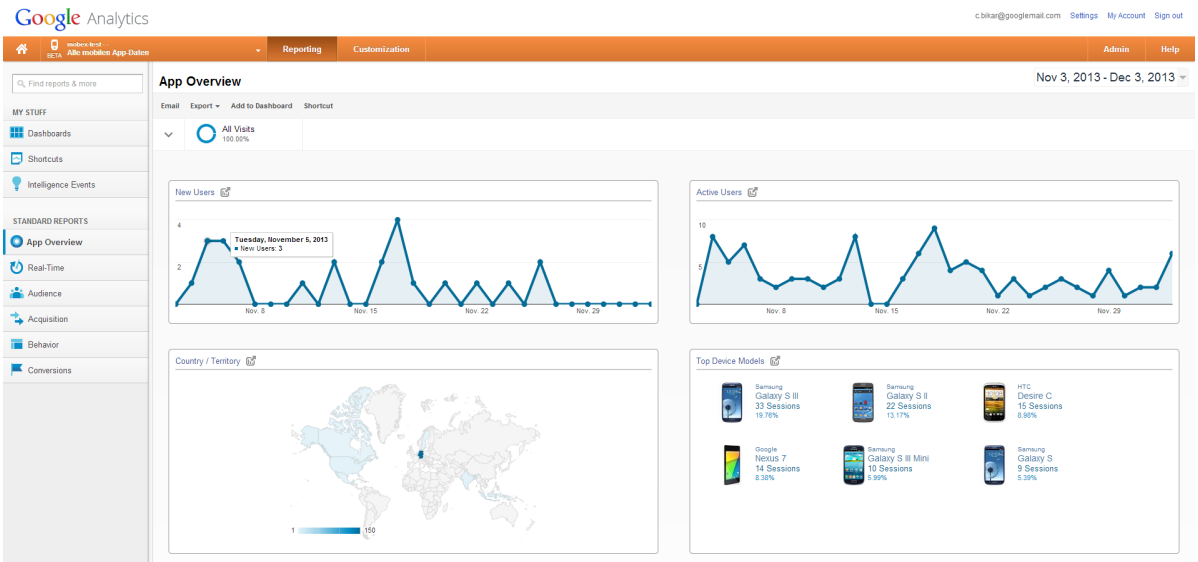


Figure 2.21.: Screenshot showing the main user interface of Google Analytics

different reports. The *App Overview* section provides an overview of all other sections, mainly demographics. *Real-Time* shows statistics about users that use the app right now. *Audience*

gives information about demographics, like location, language or devices. *Acquisition* shows how many new users were acquired and how. *Behavior* gives insights about how the app is used. This is also the most interesting section for the evaluation of mobEx. The *Conversions* section is not relevant in our context. In the *Behavior* section, the most interesting submenus are the *Behavior Flow* showing a flow a subsequent actions that users do and the *Events -> All Events* menu where all custom events are shown. On top of the interface, you find two tabs. The preselected *Reporting* tab and the *Customization* tab, where customized reports can be created. A custom report requires a dimension and a metric. Figure 2.22 shows an example where we defined the User IDs as dimension and the number of sessions as metric.

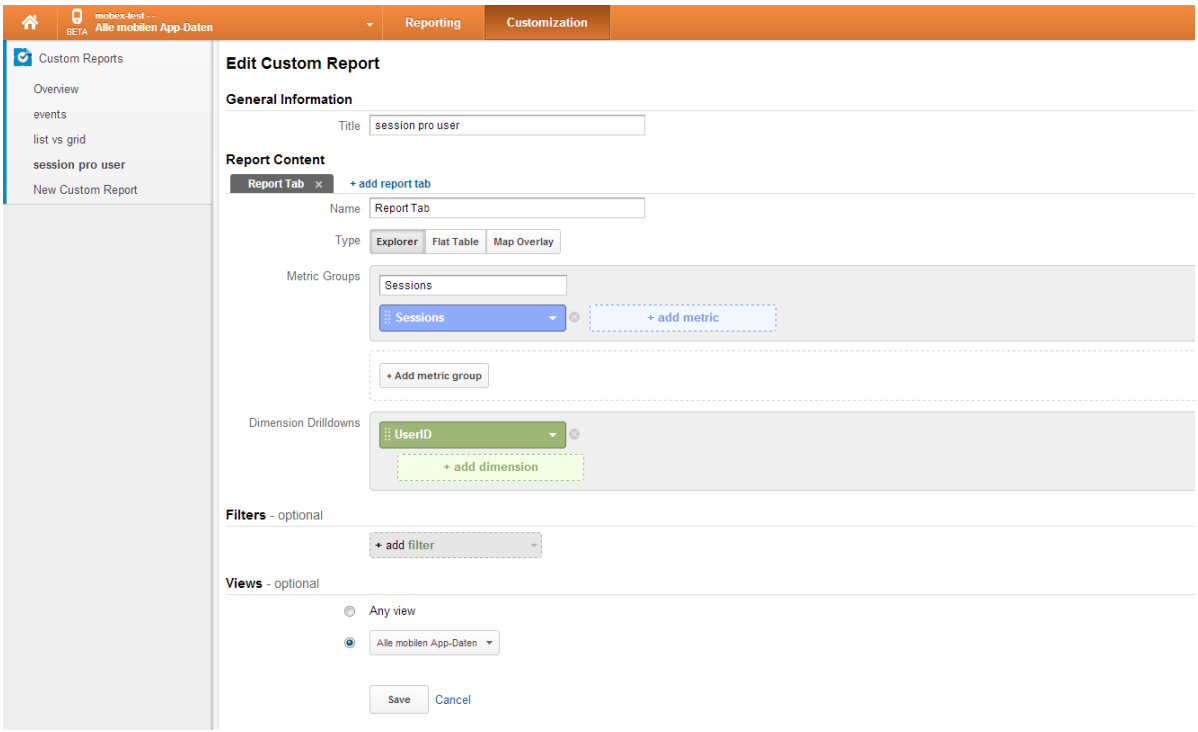


Figure 2.22.: Creating a custom report

The time range and the user segment can be customized in all screens. For the time range, a click on the date on the top right of the screen opens a dialog where a new time range can be set. In order to customize the user segment that should be taken into account, a click on "All visits" opens a screen where a segment can either be selected or newly created. For mobEx we created a custom segment that contained only the participants of the user study. In the respective screen, we chose to filter by "Condition" in the advanced section and filtered by UserID which corresponds to the custom dimension we created. We then selected "is one of" and inserted all the UserIDs, separated by a new line. Figure 2.23 shows this screen and in Table 2.3 all UserIDs are listed.

UserID
da55daf48a281645
7417d84829075721
bbe87d3fb648c5a9
5fc22cfa6d65ee1f
a4bfa95a63b76478
11517e1e1da6f273
6ec0628f4a17573f
5716e139d948bb02
57790fd861103347
8e26c7bc393602e7
16bea79fd2d5805
483092bbc904eeed
752790cfe0688051
9bec6eb91dc1ec17
58ee41bd231c2e81
a0b487fe94a95d32
e621d2a20ea30875
7a119e17e2315f28

Table 2.3.: UserIDs of the user study participants

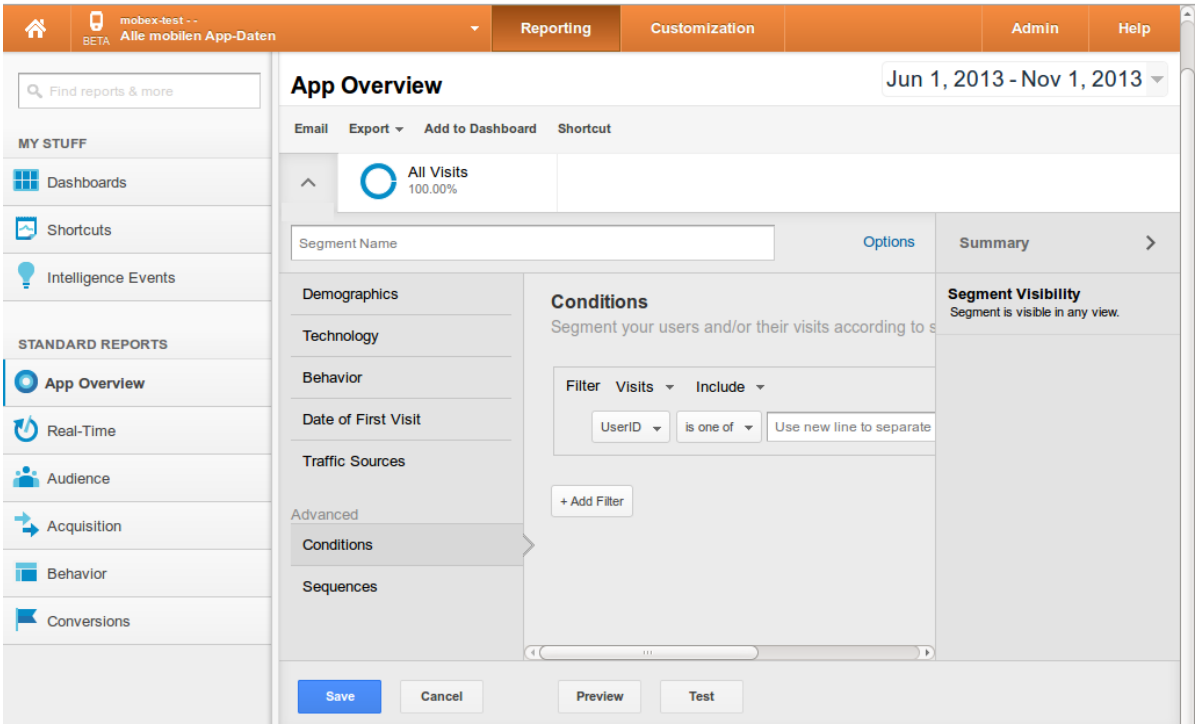


Figure 2.23.: Adding a new user segment

2.3. Future Work

In this section, ideas for possible future work are presented. We collected these ideas during the project, but their realization was not possible due to insufficient time. The concepts only deal with extensions for the client application. For future work of the server, please see Chapter 3.3.

2.3.1. Tablet Layout

Considering the increasing popularity tablet devices, the idea is to design a designated layout or view for tablets. When using the current version of mobEx on a tablet, the UI is just scaled to the size of the screen. This is not optimal, because the space is not used effectively. An approach could be to use the additional space and show map and facets side by side in landscape orientation. This way, users can select facets and see the results immediately on the map. This would also help to understand how facets and the items shown on the map correlate.

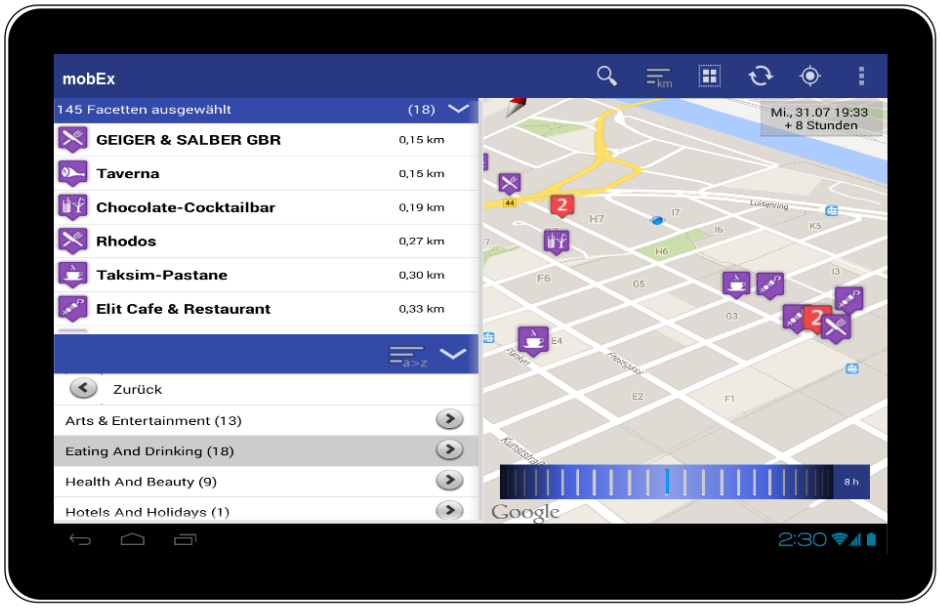


Figure 2.24.: Mockup of a possible layout for tablets

2.3.2. Integration of Social Networks

By integrating social network applications like Facebook or Google+, we would have several opportunities to add location and events that are of higher interest to the individual user. After the user gave us permission to access his or her profile, mobEx would not only query the usual datasources but also check the user's profile for relevant locations that he or his friends have visited or checked in and also for events that the user wants to attend in the future. The additional set of events and locations will be shown on the map and added to suitable facets.

An integration of the additional information with existing data is also possible. For instance, if a friend rated a place on facebook, we can show this rating along with the basic location information we got from Qype or others.

2.3.3. Editing or adding events on the Client

It is often the case that detailed information about an event or a location is wrong or incomplete, e.g. when a location like a bakery moves to another place. Instead of sending a request to the datasource (Qype, Google, etc) itself by using the corresponding webpage, we could provide a suitable interface on the client allowing a direct modification of the event. Those changes are then send to server and further processed. Additionally, a function to add new locations is also possible. New locations are either stored on the mobEx server or sent to datasources that provide an interface for adding new locations.

2.3.4. Time Aspect throughout the Client

In the current version of mobEx, the time aspect is only visible on the map screen. This means that the time can only be changed in the MapActivity and those changes affect only the events that are shown on the map. When switching over to the FacetActivtity, all events are shown, regardless of which time and timespan is selected.
The time aspect, being a primary characteristic of mobEx, could also be visible in the Facet Screen, where events that do not match the given time are hidden. However a first prototype showed that after adding controls to view and change the time to the FacetActivity, the screen is too overcrowded. Thus, the challenge is to integrate the time aspect in a way that is simple and intuitive to the user.

3. Server

3.1. Application Design

This sections deals with the software architecture of the *MobEx* server and describes the layout of the interaction with a client. Furthermore also the local and shared package structure will be explained.

3.1.1. Software Architecture

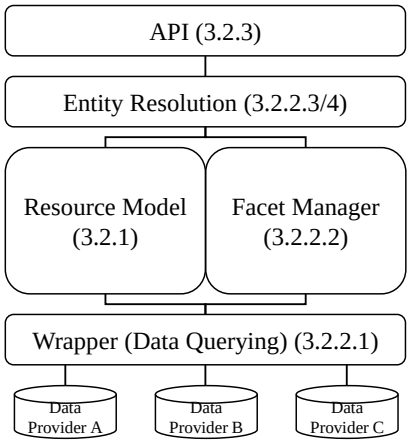


Figure 3.1.: *MobEx* Server Architecture

The architecture of the server comprises the Wrapper (see Section 3.2.2.1), RESOURCE model (see Section 3.2.1.1), FACETMANAGER (see Section 3.2.2.2), and *Entity Resolution* (see Section 3.2.2.4). Each of them represent a layer of the architecture (see Figure 3.1). The Wrapper communicates with all implemented data provider (see Section 3.1.2) and adapts the queried data into the RESOURCE model. The next layer includes the administration of all available RESOURCE objects as well as the allocation of each RESOURCE object to at least one entry in the facet tree (see FACETMANAGER). The layer of the *Entity Resolution* tries to improve the quality of these RESOURCE object by duplicate elimination and merging. All of these data can be requested through the API layer. The components which were mentioned in this section will be explained in detail in Section 3.2. Figure 3.1 illustrates the structure of Section 3.2 (bottom up).

3.1.1.1. Client / Server Communication

There are two types of querys which can appear during the client and server communication, namely *Resource Querying* and *Facet Querying*. The communication process as well as the idea how it was implemented were already described in Section 2.2.3. Therefore, this section describes only the technical part.
When the server receives a HTTP request from a client then the request will processed by the RESTRESOURCEPROVIDER. The RESTRESOURCEPROVIDER forwards the request to an underlying layer and waits until a result is available (see Section 3.2.2). The result is a set of RESOURCE objects which will be serialized and compressed before it is sent. Thus, the result is forwarded

as a binary data stream to the client. The RESTResourceProvider verifies if the client supports the used compression (gzip). Therefore, the client has to indicate in the request if it supports the compression. Without this information the data will send uncompressed which increase the traffic up to 80%. The transfer of the result is incremental and the client has to know the Resource class and its sub-classes (see Sections 3.1.1.2 and 3.2.1).

3.1.1.2. Package Structure

The entire structure of the project (client and server) is based on a *Three-tier* architecture [4] (see Figure 3.2). The server implements the *Logic* and the *Data* layer, the client is the *Presentation* layer. The *Logic* layer is represented by the server package which includes the **API** and **Resolution** package. The *Data* layer is provided by the **Adapter** package. The **Utils** package is used by both layer (see Figure 3.3). The **API** package enables the possibility to answer to requests from the *Presentation* layer. Furthermore, the **API** Package (*Logic* layer) communicates with the data providers through the **Adapter** package (*Data* layer) which implements all communication interfaces to the data providers as well as the transformation methods that store the result of the data providers in the data model provided by the **Shared** package. When the *Data* layer (**Adapter** package) has delivered the result to the *Logic* layer, the data will processed by the classes in the **Resolution** package (see Section 3.2.2.4) which then return it to the **API**. Finally, the *Logic* layer pushes this data to the *Presentation* layer. The **Utils** package contains multiple classes with helper functions which support all other packages. This includes, e.g., geocoding and reverse geocoding functions for the **Adapter** package as well as the calculation of the distance between two coordinates on the earth's surface for the **Resolution** package.

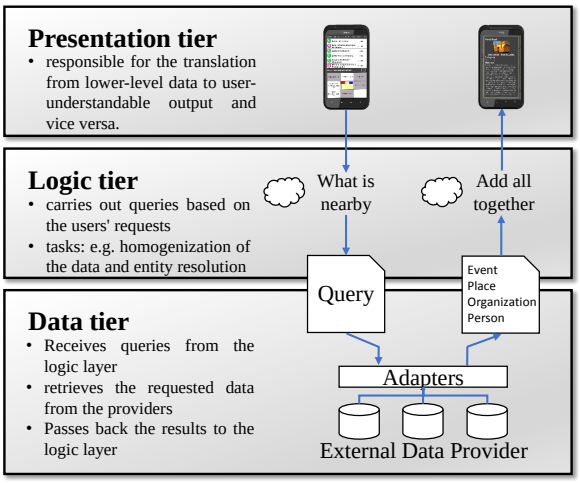


Figure 3.2.: *Three-tier architecture* pattern. The presentation tier comprises the Android client, the *Logic* and *Data* tier are part of the server.

The **Shared** package is used by client and server alike and contains the data model. The data model is used for the homogeneous representation of results on the client and server side.
An overview of all packages at server-side and their dependencies are illustrated in Figure 3.3. For more details regarding the packages of the client see Section 2.1.3.

3.1.2. Data Providers

Neither the server nor the client have their own database as a source for event or place entities. Each time the client requests data for a specific area the server starts requests to several data providers for information. The following sections comprise the information about the particu-

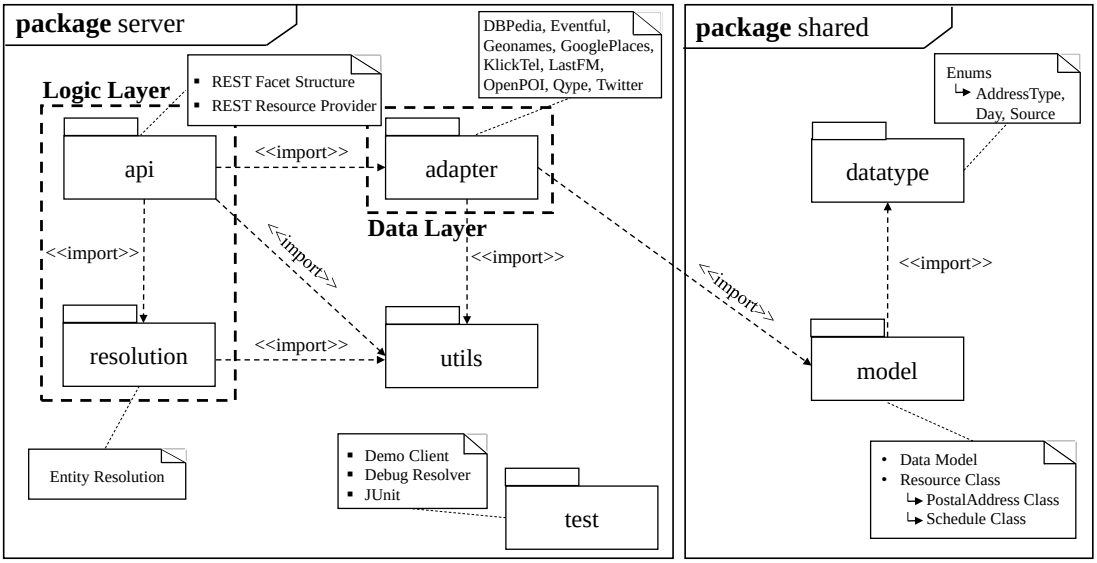


Figure 3.3.: UML Server Package Diagram. Overview of all packages at server-side as well as the data model package used by client and server.

lar data providers such as the used interface (the so called API), the provided kind of data and the restrictions.

3.1.2.1. Qype

Qype provides information about `Places` like restaurants. The provided web interface allows for anybody to add or update information like opening hours. So, it is possible to complete incomplete entries or to correct incorrect once. *Qype* provides data from multiple countries, so it is not restricted to Germany or Europe.

Interface/API/Technology *Qype* provides a REST interface for data querying. The interface allows to query information dependend from the current location. The result is represented as JSON or XML document. Our implementation uses the JSON document. The endpoint of the API is:

```
http://api.qype.com/v1/positions/<lat>,<long>/places.json?
```

The parameters which are used for a query are described in Table 3.1. Further parameters are described in the official documentation[1]. Additionally, each request has to be authorized. The used API key is: *zQXF5VKHyFX9rxTFCgpM7w* . The average response time of this endpoint is 9.3 seconds per request.

Provided Data *Qype* offers categorized information regarding `Places` and `Events` such as cafés, restaurants, and tourist attractions. Nevertheless, they also provide feedback of the community regarding the various place like reviews, rating, and photos.

[1] `http://apidocs.qype.com/start`

Parameters	Description	Required/optional
consumer_key	the used API Key	required
lang	language of the result	optional
per_page	determines the maximum entries per page	optional
radius	determines the requested radius in kilometers. The default value is 0.5	optional

Table 3.1.: Overview of the used parameters at Qype

Restrictions The main restriction of this provider is the limited number of queries per day. The *Basic Account* just allows 200 requests per day. Hence, we requested a *Pro Account* which has no limitations regarding the querying. Furthermore, *Qype* is an international database but not all entries are available in all languages. The problem is that a language has to be defined for the querying. Typically, the chosen language depends on the language of the client. So a German Android phone would not find any entry ,e.g., in New York. We also have to consider legal concerns[2].

3.1.2.2. DBpedia/Geonames

Geonames and *DBpedia* are only used together. *Geonames* is a database which provides names of geographical locations under the mostly free CC-License. They provide over 14 million entries linked with geographical positions in the whole world but they just provide names. This is the reason why we decided to enrich the information with a subsequent request to *DBpedia* which is based on information from *Wikipedia*. Both providers are independent and non-commercial.

Interface/API/Technology The *DBpedia* API works via SPARQL requests. The public SPARQL endpoint offers querying capabilities.

```
http://dbpedia.org/sparql
```

Geonames requests are done using the already mentioned REST interface which responds in XML. The service is also available with JSON, nevertheless it is not implemented using this format. The main implemented endpoint for the *Geonames* API is described in the following for a search focus on Geo coordinates restricted by a bounded box.

```
http://api.geonames.org/findNearby?
```

There are no keys necessary, but a username which is *mfacets*. One advantage of these APIs lies in their being free to use and free of charge. The downside of this is that the response of the *Geonames* and *DBpedia* bundle usually takes up to 118 seconds before the requested data can be delivered.

[2] `http://www.qype.com/developers/api_terms`

Parameters	Description	Required/optional
lat	east-west position as decimal	required
lng	north-south position as decimal	required
radius	metric search radius	required
maxRows	max number of rows per entry; default 10	optional
username	authentication	required

Table 3.2.: Overview of the used parameters at DBPedia /Geonames

Provided Data The data from *Geonames* provide named entries of geo locations such as city names, names of regions, geo-federal information and even continent names where the specific locations are placed in. One can also be request nearby searches in several variations. *DBpedia* provides information from Wikipedia thus also detailed information about regions and locations delivered by *Geonames*. We start with a request to *Geonames* that responds with rough information about the locations and enrich them in a second step by querying *DBpedia* about that location.

Restrictions For *DBpedia* as well as *Geonames*, we were not confronted with any technical restrictions that concerns us other than that there is a lack of technical performance at these providers. The effect is the already mentioned time of around two minutes that we have to wait on average. *Geonames* actually has a daily maximum amount of requests but we had never reached the 30,000 possible inquiries per day. [3]
The terms of use of *DBpedia* and *Geonames* impose less restrictions then the terms of use of the other data provider. There are just license restrictions like the CC-license and GNU Free Documentation License[4].

3.1.2.3. Google Places

Google Places offers information about `Places`, `Events` and location-related pictures. *Google Places* creates their data based on owner-verified listings and other contributors. They offer international entries with – due to its origin – focus on the United States of America.

Interface/API/Technology The *Google Places* API requires some optional and non optional parameters. The most important parameters of the nearby search request used by the *MobEx* server are listed below.
The end point for *Google Places* nearby search is:

```
https://maps.googleapis.com/maps/api/place/nearbysearch/json?
```

After retrieving records from the nearby search, we enrich the data with the information available via the details search. The parameters for the details search can be seen in Table 3.4.

[3] `http://www.geonames.org/export/`
[4] `http://wiki.dbpedia.org/Imprint`, `http://www.geonames.org/export/`

Parameters	Description	Required/optional
key	the used API Key (see below)	required
location	latitude and longitude separated by a comma	required
radius	radius in meters; maximum is 50,000	required
sensor	Boolean value whether the given location comes from a GPS sensor; here 'false' because from no GPS device	required
pagetoken	required to retrieve more results exceeding the first 20	optional
types	categories like bowling-alley or bar	optional
rankBy	order results by either prominence or distance	optional

Table 3.3.: Overview of the used/most important parameters for *Google Places* nearby search

Parameters	Description	Required/optional
key	the used API Key (see below)	required
reference	a reference that unique identifies a place; can be found in the results of nearby search	required
sensor	Boolean value whether the given location comes from a GPS sensor; here 'false' because from no GPS device	required
extensions	whether additional fields should be included; used to request a review summary	optional

Table 3.4.: Overview of the used/most important parameters for *Google Places* details search

The endpoint for *Google Places* details search is:

```
https://maps.googleapis.com/maps/api/place/details/json?
```

The API key *AIzaSyAY7Ew_6xcprCHx3ZvQ2dWr8XbVrZL5RN0* will be used each request.

Provided Data The API of *Google Places* represents a provider which offers high data quality and enriched information about places, businesses or points of interest and is backed by the same database that is used by Google Maps or Google+ locals. Each entry is ordered by category. A search can be done for "nearby", which returns a list of places based on the users location or "radar" which returns a less detailed list around a specified area, or text-based, which is the same as nearby just restricted by a search request. As already mentioned, only the nearby and details request is implemented in our server procedure.

Restrictions During the development period we recognized 2 huge restrictions within Google Places that made this provider less attractive. We did not abandon the usage of the

database as it provides very reliable data at a very high level quality. With around 5 seconds[5] this is also a quite fast provider.
There is a restriction for the maximum number of requests for a given (very short) time frame (usually some seconds) as well as a maximum number of requests within one day. The maximum of requests will be measured by using a credit point system, where radar and text search cost more points. In order to not exceed the (freely) available credits, we decided to use queries with the minimal cost per request[6]. To handle the first issue we implemented a request resending procedure used when *Google Places* rejects a request. Typically, the reason for the rejection is the retrieval speed. Additionally, we have to inquire pages (one page has 20 entries) that cost the mentioned credit point every time. Because of that we request (at most) the first 5 pages.
We are also faced with a very restrictive terms of use policy. So, we have to consider that we provide the information which data was queried from *Google Places*. The current Google policy can be seen at its homepage [7].

3.1.2.4. KlickTel

KlickTel is a German data provider. The specialization of *KlickTel* lies in phone book entries, business directories and itinerary planning. They sell several digital information services and explicitly search in address data. They are also active in mobile applications that provide these kinds of services.

Interface/API/Technology *KlickTel*'s query parameters can be seen in Table 3.5.
The endpoint is

`http://openapi.klicktel.de/searchapi/geo?`

The API Key *93a8afe1365201876013a59ced4729a3* is required for each request. Here we also have a quite fast API. Within the first 4 seconds the requested data usually arrived.

Provided Data We mainly use *KlickTel* to get information about places/organizations, such as companies and public areas. The data are mostly not free of redundancy and do not always have as high a quality as *Google Places*. An HTTP request is sent to a REST interface; the response is implemented in JSON. We use the so called geo search engine. There are also some other search methods available like a meta search to inquire a special kind of category in a determined environment, white pages which just finds private address entries, yellow page search which explicitly only searches for companies and businesses and inverse search. This listing is not complete but rather an outline of the options.
We decided for our search method because it is the least filtered method where a distance parameter can be used.

[5] average value measured by overall 135 samples
[6] `https://developers.google.com/maps/documentation/business/articles/usage_limits`
[7] `https://developers.google.com/places/policies#terms_of_use_and_privacy_policy_requirements`

Parameters	Description	Required/optional
key	means a keyword for what should be searched for	required
ygeo	latitude in the WGS84 geodetic System	required
xgeo	longitude	required
count	is the maximum entry counter	optional
what	keywords such as a company name or sector such as 'bakery'	required
distances	the given distance/radius in km where it will requested	required

Table 3.5.: Overview of the used/most important parameters for KlickTel

Restrictions There are a *Basic* and a *Pro* account available with different service levels. For instance, it will be not possible to request telephone numbers, unfiltered details about the entries or media enriched information. In order to gain access this kind of information, we had to request a *Pro* account which is not usually available for free. Nevertheless within our main search method provided by *KlickTel*, we cannot request without the 'what' parameter. This means that a keyword like 'supermarkt', 'bar', 'pizza, 'baecker', 'restaurant', 'nachtclub', 'moebel' etc. always has to be present. For a client request we thus search with distinct, hard coded keywords. KlickTel results sometimes also cause parsing problems. The terms of use can be found on their homepage[8]. Mainly it is about the storage of huge amounts of data or to consider that no end user may extract data automatically.

3.1.2.5. Twitter

Twitter is a website for micro blogging. It is in particular a communication platform for online diaries and social network of private persons, companies, or persons in public life. Due to its distribution in western countries, *Twitter* is often used as a broadcast medium for economic reasons by various groups.

Interface/API/Technology We communicate with *Twitter* via a Java library, namely Twitter4j[9]. Thus we do not have to take care of any endpoint Uniform Resource Locators (URLs) or query parameters other than the method parameters for the library. We retrieve Tweets by geolocation and parse them into RESOURCE objects. The authentication process needs four keys:

OAuthConsumer

- TWITTER_CONSUMER_Key = G4G0LZQ7Pnn9SV4xmr2bQ
- TWITTER_CONSUMER_SECRET = DHuzLafxJWEYzYkhVaQKUwj8NYqrU0bMEUrECdpRgpk

OAuthAccessToken

- TWITTER_TOKEN_KEY = 1177235755-5Pus1Uqk93JzNIT6N909hY2OLBl2GQkhtLiSZUC

[8] http://openapi.klicktel.de/terms/nutzungsbedingungen?lng=en
[9] http://twitter4j.org

- TWITTER_NONCE_KEY = Bspf9Y2iQZJcOoOZiyjMTV5xHKguBmpzaso5ZuZ5Ls

The average response time is 4.9 seconds which is reasonably good.

Provided Data *Twitter* as a social media platform provides micro blog articles, i.e. short text messages written by users at a given location and possibly about, e.g. a restaurant at that location.

Restrictions Although there are no technical difficulties, there are some legal restrictions to which we have to adhere. Mainly this concerns data privacy and copyrights as well as other general restrictions by using the content served from *Twitter*.

3.1.2.6. Last.fm

Last.fm is an internet radio and social media platform for music enthusiasts. They collect and analyze data on music listening habits and provide recommendations based on their analysis. *Last.fm* has a huge database and a extensive API which allows the retrieval of various information concerning music.

Interface/API/Technology The only used endpoint associated with *Last.fm* is:

```
http://ws.audioscrobbler.com/2.0/?method=geo.getevents&
```

Via this endpoint we implemented a method for requesting event information (e.g. concerts) based on geo location data. Table 3.6 shows the query parameters for *Last.fm*. The API key

Parameters	Description	Required/optional
lat	latitude	optional
long	longitude	optional
distance	radius (in m)	optional
limit	max amount of entries	optional
api_key	Constants.LASTFM_API_KEY	required
format	in our case JSON	required

Table 3.6.: Overview of the used parameters at Last.fm

21515cddfaa5830f7053e74aa2f21764 must be present in each request. The average response time is 4.44 seconds.

Provided Data The API offers a a lot of different possibilities. Starting from general information about artist, their album, charts, events or library of track information to also, for this project important, geo located information.
The data, we are requesting, is provided in JSON format. The response contains information about the title of this event as well as the artist, the location determined by city, country street, postal code and geo locations, an internet URL as well a starting and finishing date and time and a short description associated to this event. Often there are also pictures available.

Restrictions *Last.fm* makes legal restrictions when using their API, although there are no technical restrictions. There are terms of use[10] and also terms of service[11] which govern the use of *Last.fm* data and their API. One major issue is that internal caching is not yet implemented because the terms of use impose to cache the queried data. So it is important to implement a caching system in the near future.

3.1.2.7. Eventful

Eventful is a California-based web service that is used for searching, tracking and sharing information about current and upcoming events. Consumers as well as event organizers create new events by publishing calendar entries.

Interface/API/Technology The API communication of Eventful works via a REST interface with an XML file as response. The endpoint is described in the following and will be completed by the parameters in Table 3.7.

`http://api.eventful.com/rest/events/search?`

Parameters	Description	Required/optional
location	latitude and longitude separated by comma	optional
app_key	The authentication for the API	required
within	means the radius in units mentioned below	optional
units	here we used the metric system in 'km' default 'miles'	optional
include	what information should be included; here 'categories'	optional
page_size	is the number of entries per page here 100	optional
page_number	the page number to retrieve	optional

Table 3.7.: Overview of the used parameters at Eventful

The API key that is necessary as a parameter is *F5QkKnDswjcndL9b* . We measured an average response time of 7.5 seconds. This time is measured like all the previous and succeeding providers within this section as the duration between request and response without parsing or entity resolution.

Provided Data The data that are provided include events, venues, images and information about the performer. We obtain these data as an XML file. The response also contains – as with *Last.fm* – information about the title and description of events as well as the artist, the location determined by city, country, street, postal code and geo locations, a web URL as well

[10] `http://www.lastfm.de/legal/terms`
[11] `http://www.lastfm.de/api/tos`

as a start and end date and time. In addition, we also receive more detailed information that can be read up at in the documentation[12]. Often there are also pictures available.

Restrictions The Eventful API presents us with one major technical issue: duplicates. In order to remove the huge amount of duplicates we implemented a removing method at our server. Another peculiarity of this provider is the need to request the pages of the paginated results separately as it is necessary with *Google Places*, however there is no credit system. Again, there are terms for the (API) usage[13].

3.1.2.8. OpenPOI

OpenPOI is an open data project of OGC-Network[14]. The provided data can be queried without any restrictions and are free of charge. *OpenPOI* is a collaborative project and provides standardized information about point of interests worldwide.

Interface/API/Technology *OpenPOI* provides a REST interface for data querying. The interface allows to query information dependend from the current location. Furthermore, the data can be queried without any authentication. The downside of this provider is the delay. So, one request takes up to 35 seconds. This endpoint of this data provider is:

`http://openpoi.ogcnetwork.net/poiquery.php`

The table 3.8 describes the parameter which we are using. These parameters are mandatory. Moreover, additional parameter are described in the documentation of the *OpenPOI* Web Services API[15]. The result is represented a JSON document.

Parameters	Description	Required/optional
lat	east-west position as decimal	required
lon	north-south position as decimal	required
radius	default 50 meters	optional
format	the preferred format; here *application/json*	optional

Table 3.8.: Overview of the used parameters regarding *OpenPOI*

Provided Data *OpenPOI* provides point of interest. The data can be queried by using the API and are available as XML and JSON document. The provided data comprises e.g. hospitals, parks, governments, schools, malls, and so on.

[12] `http://api.eventful.com/docs/events/search`
[13] `http://api.eventful.con/terms`
[14] `http://www.ogcnetwork.net/`
[15] `http://openpoi.ogcnetwork.net/api.php`

Restrictions *OpenPOI* has no restrictions but there is a high delay regarding the querying. Furthermore, the provided data are often not formatted as described in the documentation which leads to parsing errors. *OpenPOI* is a collaborative project and there are no supervisory authorities. Independently, the data can explicitly be used without any restrictions by law so they are completely free to use, adapt, change, store or contribute to without any warranty of correctness.

3.1.3. Used Technologies

In order to achieve the functionality of server/client described, we use different technologies. Java was used as the programming language, Apache Tomcat is our server environment and communication between data providers and the server is implemented via XML, JSON or SPARQL while communication with the client takes place using Java object serializations (optionally compressed with gzip).

3.1.3.1. Java

Java as a modern object oriented programming language is commonly used nowadays. Java was developed in 1995 and was invented by Sun Microsystems and is now owned by Oracle since 2010 [2]. Java code is compiled to byte code and interpreted at runtime of the program. Java also brings many advantages concerning the simplicity of writing code like garbage collector and single inheritance. Java is influenced by Smalltalk, C++, and C#. [9]

We decided for Java because it is largely platform-independent. Used operating systems in our case are Linux on productive systems and Windows 7. Furthermore, the versions of *MobEx* discussed in this documentation are based on existing Java code. The previously existing code was written conforming to Java 6 standards. For our extensions to the server we switched the coding standards of the entire server to Java 7 with the exception of the shared package, as the Android client does not understand the new constructs of Java 7.

3.1.3.2. Apache Tomcat

Tomcat was also developed by Sun Microsystems as a web server and in 1999 became an open source product by the Apache Software Foundation. Tomcat is nowadays very widely used especially in university environments. It mainly consists of 3 different components:

- Catalina is a servlet container in Tomcat and implements servlet an JavaServer Pages (JSP) specifications.
- Coyote is the HTTP Connector and necessary for web server or application container.
- Jasper is the JSP file to Java parser in order to get handled by Catalina.

We decided to use Tomcat as a server environment furthermore because the development team already had experience with this software. The fact that it is open source and free of charge as well it being also written in Java are other reason for using Tomcat. The last release

was deployed in July 2013 in Version 7.0.42. The master branch of MobEx has been tested with versions of Apache Tomcat up to version 7.0.28-4.

3.1.3.3. JavaScript Object Notation (JSON)

JSON is a commonly used data transfer format nowadays. While XML offers different options for specifying data, JSON is more lightweight, because of the more comfortable readable format for humans without losing the possibility of automatic data processing. Overall, it can represent objects, arrays and values such as strings and numbers and nearly any combination build from that, also recursive ones. JSON is based on a subset of JavaScript.There are no license restrictions at all. So it is explicitly allowed to use, copy, modify, merge, publish, distribute or sublicense it.

3.1.3.4. Extensible Markup Language (XML)

XML is widely used as a markup language and for data transfer between electronic devices. The first specification was defined in 1998 by the W3C and was also continuously developed in the consecutive years. In the following years there came some extensions such as Document Type Definition (DTD) or XML-Schema, which are means to validating an XML File. XPath and XQuery can be used for data processing. One reason for the common usage of XML nowadays is that it is possible to enrich information with meta data such as the type of an entry or declaring something as a part of another.

We use XML as a transfer format when communicating with data providers.

3.1.3.5. SPARQL Protocol And RDF Query Language (SPARQL)

SPARQL is a graph based query language for RDF, where SPARQL stands for "SPARQL Protocol And RDF Query Language (SPARQL)". Since 2007, SPARQL is the successor of many other RDF query languages. It was also specified by the W3C in 2008, who also standardized many protocols, formats and modeling languages that are commonly used in the World Wide Web. The query example in the following selects the population of Mannheim.

```
1 SELECT ?number
2 WHERE {<http://dbpedia.org/resource/Mannheim>
3         <http://dbpedia.org/ontology/populationTotal> ?number}
```

Each string that begins with a question mark is a variable, so we select the object (we call number) from the subject-predicate-object triple that has the subject (instance) "dbpedia:Mannheim" with the predicate "dbpedia-owl:populationTotal".

3.2. Implementation

3.2.1. Modeling

In an earlier version of the server, there existed for each implemented data provider exactly one model class which stored the data. This means that the requester/client had to know of all available data provider as well as their model classes. Thus, the implementation of further data provider would have been very costly. We analyzed and merged all these models into one universal/core model so that the client does not have to care about the data provider.
The resulting core model that is used to store the data which were queried from the data provider is called resource model. The resource model comprises a lot of simple (primitive datatypes) and complex (non-primitive datatype) properties which were derived from the original models. There exist two more sub-models which describe the complex properties ADDRESSDATA and SCHEDULE. Furthermore other properties are constrained by stated enumerations. The resource model enables the possibility to describe a `Place`, `Event`, `Organization` or `Person`.

3.2.1.1. Resource Model

The RESOURCE model (Figure 3.4) is independent of the data provider and act as container for the queried data. The model comprises many properties that can be divided into the six categories *identifier*, *location*, *description*, *time*, *resources* and *states*. A distinction is made between simple and complex properties. Complex properties are described by further models which are described in the following sections. The *identifier* part includes the properties UUID, type and source, these properties are mandatory. The property type has to be `Place`, `Person`, `Event` or `Organization`. These types are fixed so they exist always and they will never be changed. Hence, there exist exactly four trees and each of them is a root node of a tree. These trees are facet trees and represent the available and valid categories (see Section 3.2.3.2). The *location* category comprises the geographic coordinates as well as the postal address. At least one of these two properties have to be available. If any of the mandatory properties is not avail-

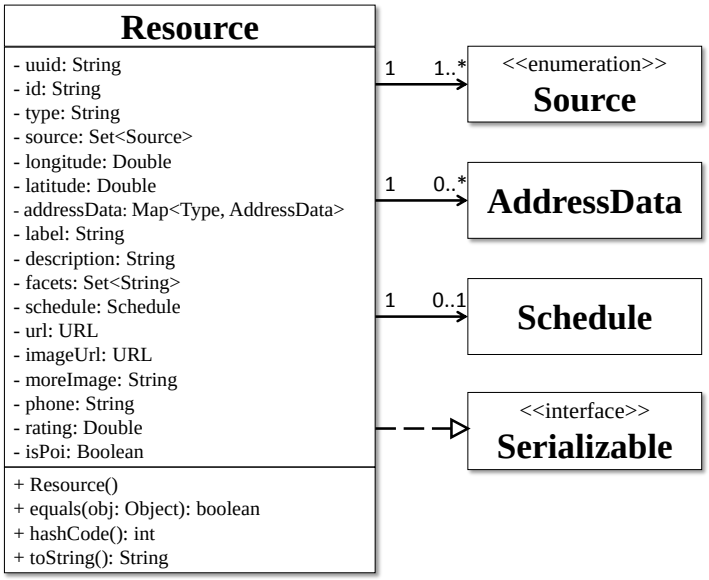


Figure 3.4.: RESOURCE Class and its relation to other Classes as Class Diagramm

able then the resource object will discarded except it is of type `Person`. RESOURCE objects of type `Person` are discarded because information about a person is even useful without a location. The *description*, *time*, *resource* and *state* categories include only optional properties. An overview of all properties with examples and further descriptions can be found in Table 3.9 below.
The RESOURCE class is shared with the client which means that the server sends as answer of a request a set of RESOURCE object so Java objects. Each of these RESOURCE object is tagged as New, Update or Delete (see Section 3.2.1.4). It is important to take into account that the client knows all necessary classes regarding the RESOURCE class. Furthermore, the Java version must be considered.

Property	Description	Example
UUID	local RESOURCE object ID, generated UUID	067e6162-3b6f-4ae2-a171-2470b63dff00
id	resource ID, original ID provided and used by the former owner (data provider)	-
type	type of the RESOURCE object, e.g. category, allowed values are `Place`, `Person`, `Organization` or `Event`	`Place`
source	name of the data provider	*Qype*
longitude	east-west position	8.46604
latitude	north-south position	49.48746
addressData	complex property	see Section 3.2.1.2 (ADDRESSDATA)
label	e.g. name, title	Mannheim
description	e.g. abstract, summary, message	is a city in southwestern Germany
facets	category, has to be in the facet tree	City
schedule	complex property	see Section 3.2.1.3 (SCHEDULE)
URL	e.g. homepage, wikiURL	mannheim.de
imageURL	e.g. thumbnail, image	http://../logo.png
moreImage	more image URLs, URLs have to split by a delimiter	-
phone	phone number, server tries to convert all phone numbers into the same format	+49621 2930
rating	feedback of other user, allowed values are between 0 and 5	3.5

Table 3.9.: Properties of the RESOURCE Model

3.2.1.2. AddressData Model

The ADDRESSDATA model (Figure 3.5) is a sub-model of the previous described RESOURCE model. This model is only a container for the queried postal address data. It should help to manage the data so that they are logically separated into different containers. It is possible to add several ADDRESSDATA object to one RESOURCE object. Furthermore, the different ADDRESSDATA objects can be tagged as birthplace, deathplace, or place. If necessary, this list can be expanded. Currently, only a RESOURCE object of type `Person` has more then one ADDRESSDATA object.

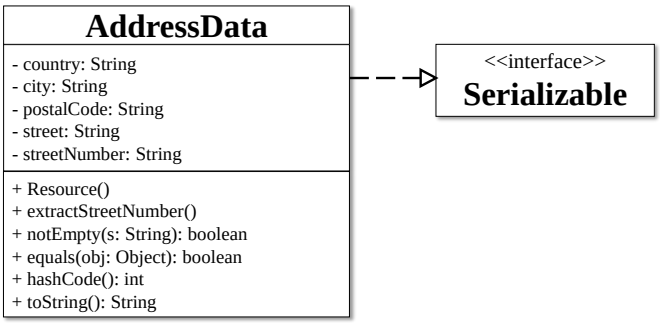


Figure 3.5.: ADDRESSDATA Class and its relation to other Classes as Class Diagramm

Frequently, data provider serve the street name only with the street number. Indeed, the model requires that these two values are assigned to two different properties. This is necessary because the *Entity Resolution* should compare the postal address data as accurately as possible. As result of this, we implemented also a method which tries to separate these two values. Otherwise the model does not include properties that are mandatory. However if the ADDRESSDATA object is empty then it is obligatory to set the geographic coordinates except the RESOURCE object is of type `Person`. RESOURCE objects of type `Person` are discarded because information about a person is even useful without a location. If only the geographic coordinates are available, then the postal address data can be searched and added with the implemented reverse geocoding method. An overview of all properties with examples and further descriptions can be found in Table 3.10 below.

This class is shared with the client like the RESOURCE class (see Section 3.2.1.1).

Property	Description	Example	Annotation
country	country of the event	Germany	-
city	city of the event	Mannheim	-
postalCode	postal code of the location of the event	68159	-
street	street of the event	Paradeplatz	data provider provide the street name often with the street number
streetNumber	street number of the event	42	street number should not be stored in the street property

Table 3.10.: Properties of the ADDRESSDATA Model

3.2.1.3. Schedule Model

In contrast to the previous described models, the SCHEDULE model (Figure 3.6) is not only a container. It allows to store two different time models, namely *time interval* and *opening hours*. The first one is independent of a weekday but represents a specific time interval which can be represented by two time-stamps. This allows to store the start and end time of, e.g. a concert. In addition to the start and end time property, there is also a property named *info* which allows to store a description or annotation regarding the time interval, e.g. *closed on 24.12*. This information will be displayed combined with the time interval. Typically, opening hours comprise multiple time intervals which are bound to at least one weekday. Such time intervals which are repeating and bounded to a weekday cannot be represented as a time-stamp. Thus, it is possible to define one or more time intervals for the same weekday. The start and end time values for the opening hours have to be delivered in a specific format to the corresponding method. So, we implemented a method which uses different pattern to solve this problem. The pattern is as follows HH:MM whereby HH represents the hours and MM the minutes. As an alternative, the values can also be individually set as integer. It is not possible to set seconds. If the format of the string is incorrect, then the value of the property remains null.

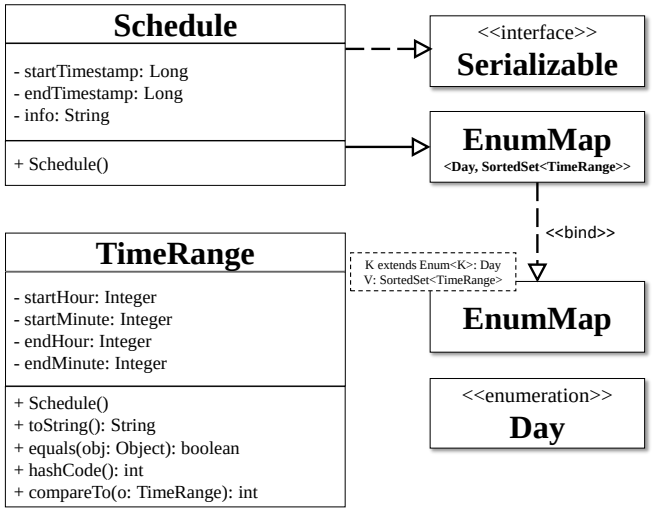


Figure 3.6.: SCHEDULE Class and its relation to other Classes as Class Diagramm

The client can determine which of the two models is used by the implemented state methods. The state method returns the type of SCHEDULE object. The *time interval* and *opening hours* models can also be combined, but there are currently no usage scenarios. All properties in this class are optional. An overview of all properties with examples and further descriptions can be found in Table 3.11 below.
This class is shared with the client like the RESOURCE class (see Section 3.2.1.1).

Property	Description	Example	Annotation
startTimestamp	specific moment in time	1374504097	22.07.2013, 14:41:37
endTimestamp	specific moment in time	1744010783	07.04.2025, 07:26:23
Set<Day>	weekdays	Mon	stated enumeration
Set<TimeRange>	opening hours for a specific day	08:00-12:00, 13:00-18:00	exists for each available day

Table 3.11.: Properties of the SCHEDULE Model

3.2.1.4. Forest of Resource Objects

Motivation The *Entity Resolution* merges RESOURCE objects. It is important to know which RESOURCE objects were merged as well as if both objects provide different data for the same property. Hence, it needs to be decided which value should be chosen. During the *Entity Resolution* all objects are compared. Thus, it is possible that one or both of these RESOURCE objects were already merged into another RESOURCE object. Furthermore, the *Entity Resolution* uses multithreading to increase the speed. So it is possible that the same RESOURCE object is compared to two other RESOURCE objects at the same time. In the worst case, both comparative processes want to merge. Hence, to deal with these problems we develop the *Forest of* RESOURCE *Objects*. For more detailed information see [12].

Implementation The *Forest of* RESOURCE *Objects* is a construct which is used by the *Entity Resolution*. The construct consists of multiple trees. So, a tree represents the merging process between multiple RESOURCE objects. In the following, we describe the creation of such a tree step by step. The describtion is related to Figure 3.7. The figure shows three different threads which were successively started. When, the first thread was started only the *blue* RESOURCE objects were available. Once the second thread was started new RESOURCE objects (*red* nodes) were available, additionally to the *blue* RESOURCE objects. Hence, the threads have a shared memory but work only with these RESOURCE objects which were available at the moment of the creation of the thread.
When a data provider thread (see Section 3.2.2.3) completes its work, the result (RESOURCE objects) is passed to the *Entity Resolution*. As a result, a new entity resolution thread is started. This thread compares the new RESOURCE objects against each other as well as with the RESOURCE objects which were earlier delivered by another data provider thread. This ensures that each pair of RESOURCE objects is compared exactly once. If there is a match between two objects e.g. node **A** and **B** then the data are merged into the object which was earlier delivered to the *Entity Resolution*, so node **B** (see Figure 3.7). Thereby it is possible that a RESOURCE object was still merged into another object, e.g. node **G** matches node **F** but node **F** was already merged into node **H**. For this reason the merge processes are represented as a tree whereby the parent is always the object in which the data were merged. In order to avoid collisions during the modification of the RESOURCE objects only the comparative processes are executed in parallel, the merge processes of the RESOURCE objects are still sequential. (see Section 3.2.2.4)
Thus, in the next step we consider the different situation which can appear during a merging process. We illustrate the situations in Figure 3.7. As already mentioned, the nodes are the RESOURCE objects and the edges represents a merge. The number of the edges define the order of the merge processes. When, both objects should be merged and were not yet merged e.g. node **A** and **B** then a new tree will created. Node **B** is the root node of the tree and node **A** is added as a child. So, the nodes are connected by an directed edge e_2. Another situation could be, if a merging partner e.g. node **F** is still merged into node **H** then the data of single node **G** has to merged into the root node of the tree which contains node **F**. The figure shows this problem in the third part. The dotted edge e_7 represents that the green and red node should be merged but the red node is not a root node. So, the green node has to be append as a child to

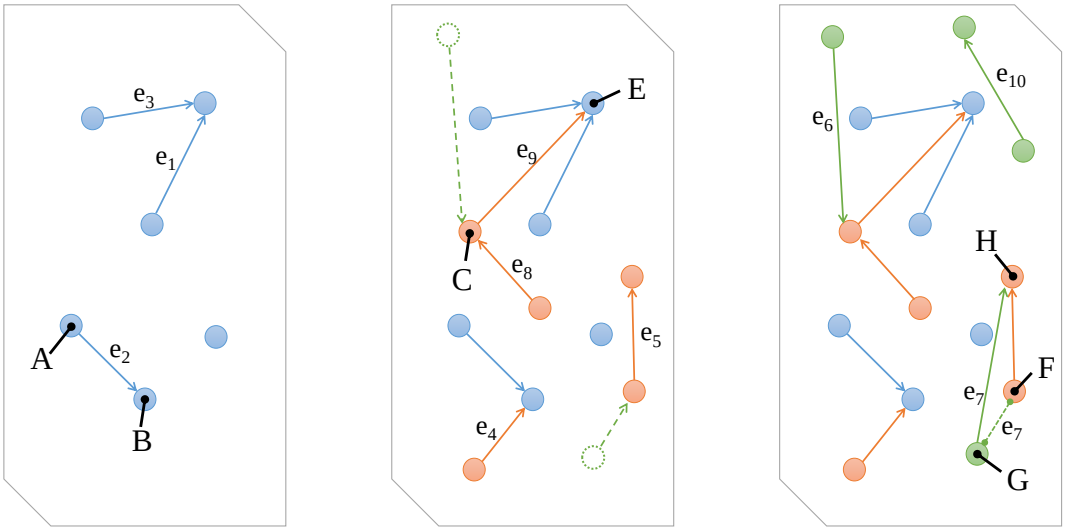


Figure 3.7.: Forest of RESOURCE Objects. The graphic illustrates the merging results of different entity resolution threads. The numbers describe the order of the merging and the arrows show the direction of the merge process. The source of the RESOURCE objects does not matter.

the root node of node **F**. So the data will merged into node **H**, represented by the solid edge e_7. It is also possible that both nodes are already part of a tree. In this case the root node of both trees will be merged and the root node of the younger tree is added as a child to the other root node, e.g. node **C** and **E**. In this context, it has to be considered that the edge e_8 still point to node **C** after node **C** was merged into node **E**. Thus, the edges illustrate the history of merge processes between the nodes. So, if an edge was established between two nodes, the edge will never moved or removed again. Therefore, the root node represents always the current state. In this way a forest of trees is created. Thus the RESOURCE objects are modeled as a tree during the *Entity Resolution*. After each merge process the results are send to the client if a root node will updated then the update will send to the client as well. Furthermore if two trees were merged then the old root node will be send as delete command.

3.2.2. Processing of Data

The main task of the *MobEx* server is the processing of data. It comprises the querying of all data from the data provider (see Section 3.1.2). Each data provider has its own schemata for representing the data. We implemented for all data providers an individual translator which transfers their schema in our own schema. Hence, the obtained data will be stored in the previous described model (see Section 3.2.1). Thus, the following steps do not have to care how the data provider provide their data. First, the FACETMANAGER validate the facet property of each of the created RESOURCE objects. The values of this property have to be part of the facet tree so that our Android client will handle these objects correctly. The facet tree itself is also provided by the server through a REST interface (see Section 3.2.3.2). Then, the

RESOURCE objects will be forwarded to the *Entity Resolution*. The *Entity Resolution* examine the RESOURCE objects in order to find and eliminate or merge duplicates. Finally, the RESOURCE objects will be delivered to the requester. In order to ensure that the requester gets the result as fast as possible, the described steps are parallelized (see Section 3.2.2.3). So, there exists as already described (see Section 3.2.1.4) multiple entity resolution threads. When one of this threads finished the resolution then the result will delivered immediately to the requester. Thus, the result will be sent incrementally.

3.2.2.1. Querying of Data Sources

The aim of the querying is to obtain information about `Places`, `People`, `Organizations` and `Events` regarding the current location. The requester has to pass four basic information (see Section 3.2.3.1) which are necessary to perform the querying.

- current location as geographic coordinates (longitude and latitude)
- search radius in kilometers
- search language
- set of providers which should be queried

The search language parameter is currently not supported by all implemented provider also for the search radius parameter exists lower and upper bounds. The server performs at least one request to each selected data provider. The requests to the data provider are started in parallel to ensure that they do not block each other. In case that a data provider does not respond, the server retries to establish a connection up to ten times. Subsequently, the server treats this as an empty response. An error will be logged but this information will not delivered to the client. The result will also be rejected if the delivered result document format is broken (e.g. JSON). Currently all providers have REST interfaces which allow to query the data by a HTTP requests. The only mandatory feature is that the API of the provider supports requests depending on a location. An answer of a data provider will be parsed into several RESOURCE objects. During the parsing of the information into a RESOURCE object, it will also be checked if the geographic coordinates as well as the postal address information is available. If only one of the information is available the server performs another request to the provider Mapquest[16] which provides geocoding as well as reverse geocoding. The REST interface of Mapquest allows so send multiple requests in one HTTP query. Furthermore the provider supports the result as fast as Google but without any restrictions. If both information is missing the RESOURCE object will discarded. This is part of the preprocessing of the data for the *Entity Resolution*.

[16] `http://www.mapquest.de/`

3.2.2.2. FacetManager

The FACETMANAGER administrates the facets. Therefore, it validates the facet property of each RESOURCE object because each value of this property has to be part of a facet tree. If a tree does not contain a specific value then the FACETMANAGER tries to map the value to one which exists in a facet tree. This process will be described in the following sections. The mapping is defined by a set of rules which are stored in a local SQLite database. In addition, the FACETMANAGER loads in certain time intervals all provided categories from each implemented data provider to check if new categories are available. If this is the case, the manager tries to create a new rule. The idea of this concept are discussed in detail in the following.

Basic Idea Typically, each data provider has its own facet structure. The task of the FACETMANAGER is to map facets of a data provider to our own facet structure. In order to map the assigned facet of a RESOURCE object by a data provider to a fixed set of facets, we use a facet tree. The mapping is implemented by predefined rules and they are stored in an *SQLite* database. The rules and the mapping are described in detail in the next section. We decided for an *SQLite* database because it is a light-weight, does not have to be installed separately and it is supported by Windows and Linux. Furthermore, we decide to use a facet tree which is static at run-time to avoid to confuse the user. When a RESOURCE object has a facet which is not covert by any rule then the FACETMANAGER should try to find a similar facet in the facet tree by string similarity. The database schema allows to to map one facet to multiple facet as well as multiple facets to one facet. For instance a “medical doctor for animals” can be mapped into the category “pets” as well as “medical services”. If one is looking for the doctor, he can be found in multiple categories where one would expect to find a doctor. Furthermore, when there is no such doctor to assign to the specific facets, such as “medical services” under “pets”, it will not be part of the currently used individual facet structure.

Mapping and Storage The mapping is implemented by means of three entities that are dependent on each other, namely *dict*, *facet* and *map* (see Figure 3.8). The *dict* entity comprises the categories we obtain from the data providers. Typically, the attribute **description** is an extension or description of a short cut regarding the attribute **facet**. In this context, the **source** attribute contains the name of the data provider which provides this data. The *facet* entity owns the attributes **origin** and **human**. All facets which are part of the facet tree are stored in this entity. The **origin** attribute contains the notation of the facet as it appears in the facet tree, e.g. a facet name without spaces. The **human** attribute represents the name of the facets as they will be shown at client-side, e.g. spaces are added in the right places. The entries of the *facet* entity are matched to entries of the *dict* facet whereby n:m relations are realized by the *map* entity. This relation is also described in Figure 3.8. New or unknown facets provided by a data provider are stored automatically in the *facet* entity. When the string similarity method does not find a match for the new or unkown facets then a rule has to be created manually. There are currently no mechanism implemented which avoid inconsistents in the database.

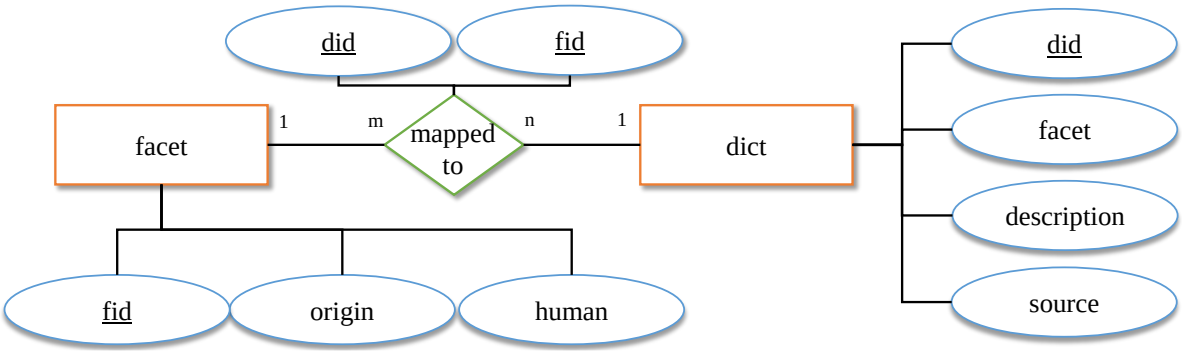


Figure 3.8.: Entity Relationship Model. Structure of the SQLite database.

Detecting new Facets The recognition of new facets is important because if a RESOURCE object contains an unknown category then the client cannot determine the correct position in the facet tree. It can occur anytime that an implemented data provider extend their pool of categories. For this reason, the provided categories of all implemented data provider are loaded at regular time intervals. The reloaded data will be compared with the data which are already stored in the SQLite database. If there is a value which is not stored in the database then it has to be a new facet.
It is also possible to wait until a data provider returns an entry with a new facet. However, in the worst case the data provider changed its hole facet structure. So we would have to deal with a great amount of new facets which results in a significantly longer waiting time for the requester. Therefore, the preventive measure allows to use a more complex algorithm to determine a matching partner in the facet tree.
The server attempts to find a match for a new facet with string symmetries. So, it compares the new facet with all existing facets in the facet tree and calculate a confidence value between 0 an 1. We are using the Levenshtein distance [8] which determines the minimum number of delete, replace and insert operations to convert the new facet in the selected. If the highest resulting confidence value is higher then the defined threshold of 0.76 then the new facet has a match. In all other cases a notification is stored in the log-file for the administrator. The threshold is based on personal experience with an earlier project [6].
An idea which could not be pursued because of lack of time was to combine the new facet with the provided description to determine the meaning of the facet through reasoning.

3.2.2.3. Multithreading

It is an important goal to sent the requested data as fast as possible to the client. Therefore, all processes are working in parallel threads. There exists currently two main stages which are the querying of the data from the data provider as well as the *Entity Resolution*. Both stages are divided in multiple sub-threads. Besides these stages there exists also three central units, namely *Main-Thread*, *Entity-Manager-Thread* and *Entity-Resolver-Thread*. They are responsible for multiple sub-threads. In the following we consider the Figure 3.9 to describe how these threads interact with each other.
At the beginning, the client sends a request (1) to the server. Subsequently, the *Main-Thread*

of the server starts at least one thread for each implemented data provider. The number of threads for each data provider depends on how the data have to be queried from the data provider. In the case of Eventful, the query result is distributed over multiple pages. This stage is illustrated in the Figure 3.9 as the directed edges which point from the *Main-Thread* to multiple provider-threads (2).
These provider threads are not synchronized so that they terminate at different points in time. If a provider thread finished, it passes the result to the *Entity-Manager-Thread*. The task of the *Entity-Manager-Thread* is only to listen to the provider threads, to take their results, and pass it on the fly to the *Entity-Resolver-Thread*. Furthermore it observes the entity resolver worker

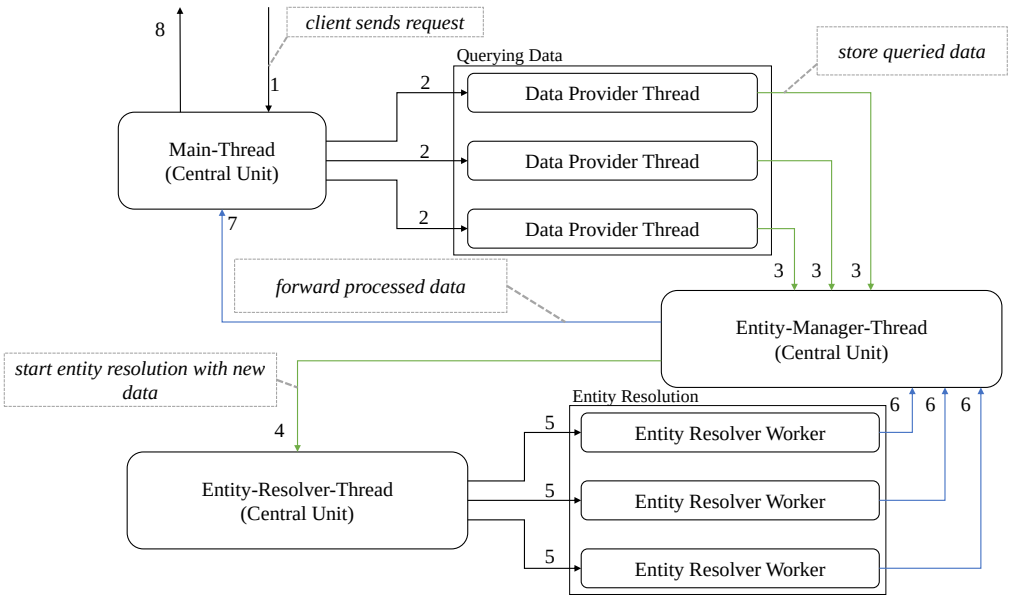


Figure 3.9.: Interaction of the threads at server side during processing of a request. The numbers describe the order of the processes.

threads which are described in the following. The data are not directly delivered to the *Entity-Resolver-Thread* because this could block its main task. The *Entity-Manager-Thread* lives as long as the *Main-Thread*. The figure 3.9 describes this stage with the directed edges that point from the different provider threads to the *Entity-Manager-Thread* (3). The edge which points from the *Entity-Manager-Thread* to the *Entity-Resolver-Thread* illustrates the transfer of the data from the second too the third central unit (4). The other edges which are connected with the *Entity-Manager-Thread* are described in the following.
Every time the *Entity-Resolver-Thread* gets data from the *Entity-Manager-Thread* it starts a new thread which is called entity resolver worker. The number of these threads depends on the number of provider threads. The task of each thread is the *Entity Resolution*. For further information as well as the interaction of these threads among each other see Sections 3.2.1.4 and 3.2.2.4. As already mentioned, the entity resolver worker are observed by the *Entity-Manager-Thread*. If one of these entity resolver worker finished, then the *Entity-Manager-Thread* takes and stores the result. This is shown in Figure 3.9 by the edges which point from the *Entity-Resolver-Thread* to multiple entity resolver worker as well by the edge from these entity resolver worker to the *Entity-Manager-Thread* (6).

After the *Main-Thread* has started all provider threads it listens all the time if the *Entity-Manager-Thread* provides results. If there are results then the *Main-Thread* takes these and sends them to the client. The *Main-Thread* continues until all data are processed. After no more results are available, so all provider as well as entity resolver worker threads are terminated, the *Main-Thread* instructs the *Entity-Manager-Thread* to shutdown. Then the connection to the client will be terminated. This last stage is illustrated in Figure 3.9 as the edge between *Entity-Manager-Thread* and *Main-Thread* (7).
Thus, the data are send incrementally to the client (8). For a more detailed description at server side regarding (4), (5) and (6) see Sections 3.2.1.4 and 3.2.2.4. For the handling of these data on the client side (8) see Section 2.2.3.

3.2.2.4. Entity Resolution

In the following, we will deal with the inner workings of the *Entity Resolution* which takes place in the EntityResolverWorker. When the *MobEx* server queries data providers, the reply to the requesting Android client should be quick, as users are not willing to wait a long time – they will most likely expect results in a matter of seconds. Still, results should not contain duplicates. We are therefore faced with the tradeoff of running time and performance: While the process of matching may not substantially increase the time until the client receives at least some results, false merges should be avoided at all costs (as they essentially render the data useless) while a few remaining duplicates may be acceptable.

In *MobEx*, *Entity Resolution* takes place on Resource objects. A Resource represents an entity of any of the four types (`Events`, `Organization`, `Persons`, `Places`). When querying data providers, all retrieved records can be mapped into such a Resource, allowing for easy comparability. We apply certain preconditions before comparing Resources as explained below. Afterwards, comparisons of resources and thus the matching by a means of scoring takes place. The relevant properties of a Resource and how they are handled during the entity resolution process – especially the scoring – are explained in below.

Basic Idea / Preparation One of the most important aspects for our entity resolution process is that it has to be fast, i.e., reliability is less important than speed. Our resolution process is multi-staged. We decide which resources to compare by precondition heuristics (see Section 3.2.2.4 below), i.e. we do not carry out further comparisons on resources not fulfilling the preconditions. The remaining resolution process then takes place on the properties of the Resources (see Section 3.2.2.4 below). Resolution can be set to a certain level where higher levels subsume all the steps of lower levels. Levels are ordered such more promising attributes, i.e. more distinguishing attributes that are algorithmically cheap to compare, are used at lower levels. As such, the amount of time it takes to compare the properties increases for higher levels and while still beneficial, it is less so. Economically speaking: higher levels should have a decreasing marginal utility as the processing time increases more than the performance of the resolution. Resolution is also incremental, meaning that when a bulk of resources has been resolved by a worker thread, it is forwarded to the client. Results of later resolution steps may cause updates to already sent results. This can be seen in Figure 3.10, where we receive

an incomplete entry for a café at some point and after entity resolution update the existing resource on the client.

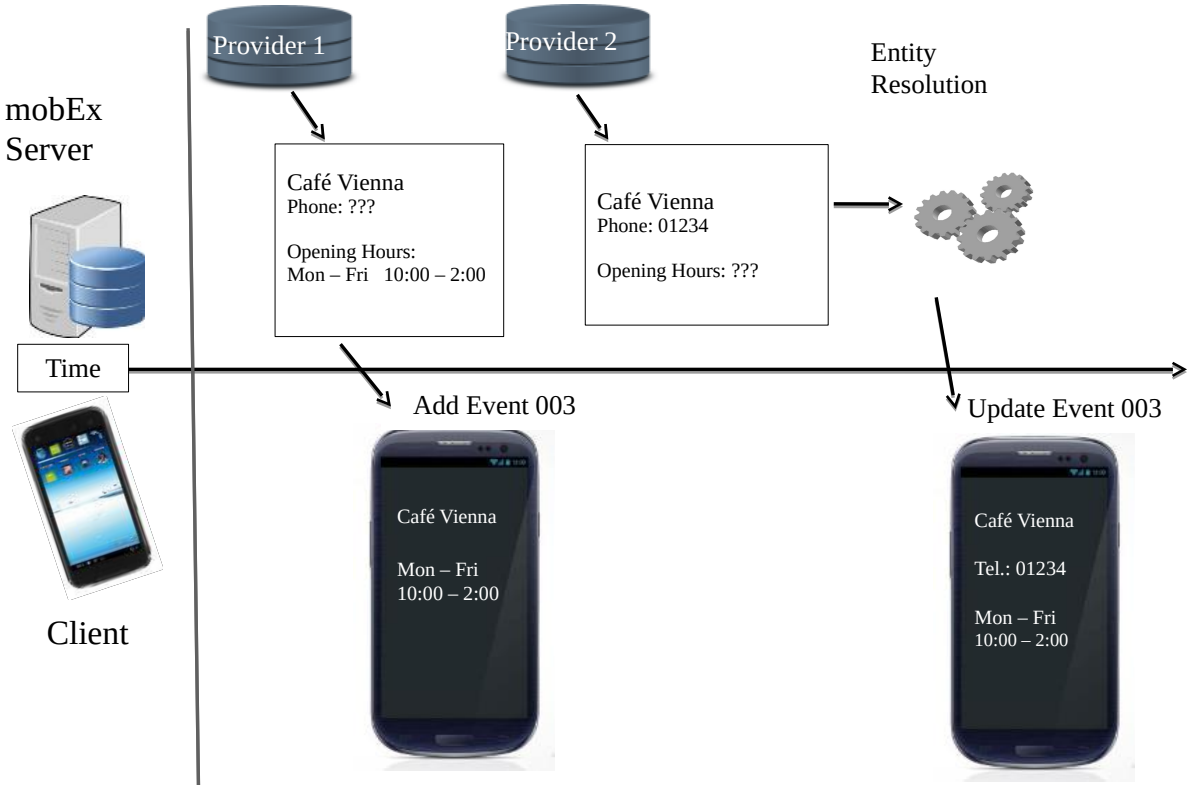


Figure 3.10.: An example of our resolution process including an update after resolution detects a duplicate.

Preconditions When querying data providers, we may very well receive a total of more than 1000 instances in major cities. Naive *Entity Resolution*, i.e., comparing all pairs of resources, is therefore out of the question, since that would result in $\binom{n}{2}$ and thus $O(n^2)$ comparisons. Instead, we cut down the number of comparisons by applying the following preconditions:

- Provider (*source*): Records retrieved from Twitter – or rather the resources these were mapped to – are ignored during the entity resolution process. Carrying out *Entity Resolution* on Tweets will not improve results, as Tweets are already free from duplicates and will not have corresponding resources in other provider's results.
- Type: Only compare instances of the same *type*, i.e. compare `events` with other `events`, but not with `places` or organizations (analogously for the other types)
- *Physical location:* The greater the physical distance between two resources, the less likely it is that they describe the same entity. This idea is also backed up by [11]. We account for that by calculating the distance between resources using the Haversine formula [14]. We will only consider pairs of resources for resolution if

a) they have a distance of at most 500m from one another. We call this value the *distance threshold*.

or

b) their postal addresses are similar (to account for wrong coordinates from a data provider). We define two postal addresses as similar, if the average of their Jaro-Winkler and Levenshtein similarity is greater than 0.75. This only done for resolution levels of 40 or higher (see explanation further below).

The value of 500m that we use for the distance threshold was determined empirically. When looking at a visualization of results from data providers on a map, we found that the last (obvious) duplicates were less than 500m apart. The Haversine formula is more appropriate than other distance measures such as the Euclidian distance since it is easy and fast to calculate and works well for small distances where other formulas show rounding errors [14, 1].
Let p_i be a point consisting of latitude and longitude. Let $lat(p_i)$ return the latitude of p_i and let $lon(p_i)$ be the function to get the longitude of p_i. Furthermore, $rad(x)$ converts x from angle to radians and $R_\oplus = 6378.1370$km (which is the earth radius). The Haversine distance in meters d between two points p_1 and p_2 is then defined as follows:

$$
\begin{aligned}
dLat &= rad(lat(p2) - lat(p1)) \\
dLon &= rad(lon(p2) - lon(p1)) \\
a &= \sin\left(\frac{dLat}{2}\right)^2 + \cos(rad(lat(p1))) \cdot \cos(rad(lat(p2))) \cdot \sin\left(\frac{dLon}{2}\right)^2 \\
c &= 2 \cdot \arcsin\left(\sqrt{a}\right) \\
\mathbf{d} &= \mathbf{1000} \cdot \mathbf{R}_\oplus \cdot \mathbf{c}
\end{aligned}
$$

This checking of preconditions takes place in the EntityResolverWorker directly preceding the resolution. It is directly integrated into the resolution process, because checking the preconditions, especially the distance precondition, cannot be sensibly carried out at any other point.

Scoring and Comparisons The scoring and comparisons take place to calculate a score by which it can be easily decided whether two Resources represent the same entity or not. The scoring takes place in the EntityResolverWorker. In the scoring process, we only compare resources for which the preconditions apply. When comparing resources, certain properties are more important than others. We account for that by assigning weights to the different properties as shown in Table 3.12, which are used in the scoring process. Furthermore, the resolution process can be set to work at certain levels where higher levels include more properties in the process, thus possibly improving the results at the cost of speed. Higher levels subsume lower levels. The scoring works as follows: Let r_1, r_2 be two resources, $C = \{c_1, \ldots, c_n\}$ the set of resource's properties (but not values) that are relevant (i.e. included in the currently set resolution level) for *Entity Resolution* (see Table 3.12) and that are present on both Resources

(i.e. they are not null or empty), n=|C|, $w_1, \ldots, w_n$ the weights assigned to each component and $compare(r_1.x, r_2.x)$ a comparing function, defined as follows:

$$compare(r_1.x, r_2.x) = \begin{cases} 1 & \text{for } r_1.x = r_2.x \\ 0 & \text{otherwise} \end{cases}$$

for all properties except for the label and parts of postal addresses for which $compare(r_1.x, r_2.x)$ is defined as the average of the Jaro-Winkler and Levenshtein similarity. Thus, $compare(x, y)$ is bounded by 0 (signifying dissimilarity) and 1 (signifying identity). With the score calculated as

$$score(r_1, r_2) = \sum_{i=1}^{n} w_i \cdot compare(r_1.c_1, r_2.c_1)$$

we consider $r_1 = r_2$ if $score \geq t$, where we call t the comparison threshold (we use $t = 0.7$). There are two exceptions to this process to speed it up:

- Duplicates that a provider delivers (i.e. the source and the provider's id for two resources are identical) are always considered to be identical.
- Resources with the exact same label and description – given that both properties are present, i.e. non-empty, on both resources – are always considered identical. This assumption is useful for those cases, where e.g. an event organizer inserted the same information into multiple provider's databases. This does *not* depend on the resources originating from the same provider.

If you were to add new properties or comparisons to the resolution process, you would have to assign a sensible weight an a resolution level to the properties and add code to handle the property accordingly. That is, one would add a code block to the EntityResolverWorker that is active for the given level and adjust the score as it is done in the other blocks with respect to the weight.

Merging Merging takes place in the EntityResolverWorker based on the determined score. The scoring and merging parts are tightly integrated into a single process. When merging resources, some attributes require special treatment, while others may not need to be touched at all. Table 3.13 shows how resources are merged. Attributes that are not shown below remain unchanged. In the merging process, the older/oldest resource takes precedence in so far, as its properties will mostly be assumed to be correct unless they were empty or stated otherwise in the "Merging" column in Table 3.13. We decided to do so, because an older resource will most probably have been transferred to the client. Aside from that, there is no indication that slower providers have better data unless the merging strategy (see Table 3.13) determines so. A resource is considered old(er) if another resource has already been merged into it, the oldest resource is the root of its merge tree (see Section 3.2.1.4 and Figure 3.7). We call the oldest resource in a merging process *rOld* or merge target, while the other resource is *rNew* or the merge source. When merging, **only rOld** will be **changed** (as we merge additional new information into rOld and send it to the client again). The merge forest is explained in Section 3.2.1.4.

Attribute	Data type	W.	Lvl.	Note
label	String	3	1	also used for label and description identity check.
description	String	0	1	only used for label and description identity check.
schedule	Schedule	2	5	only on type “event”
url	URL	4	5	
imageUrl	URL	1	10	
phone	String	3	20	
postalAddresses	PostalAddress	2	40	property contains possibly multiple addresses (birth place, death place); only compare places of the same type, i.e. birth place with birth place but not with death place.
↪ city	String	0.1		+0.1 bonus if countries are similar
↪ country	String	0.1		
↪ postalCode	String	0.1		+0.2 bonus if cities are similar
↪ street	String	0.2		+0.3 bonus if cities are similar
↪ streetNumber	String	0		+0.5 bonus if streets are similar; an identical street number alone does not change the score (weight=0) while a similar street *and* then an identical street number gives a bonus of 0.5

Table 3.12.: The relevant properties of a Resource for the entity resolution process. The column W. shows the weights used in the resolution process.

3.2.3. Data Access (API for the Android Client)

The server provides two different REST interfaces which enables the possibility to communicate with the server. Thus, a loose integration of a client is possible. One of them is the RESTResourceProvider which allows to query `Places`, `People`, `Organization` and `Events` regarding the current location. This is the main interface and the only one which allows to send data to the server. The other one is called RESTFacetStructure and provides only a tree of the valid categories of the queried data.

3.2.3.1. REST Resource Provider

The RESTResourceProvider allows to query information depending on the current location. The requester has to send a HTTP GET request with specific parameters to start the querying on the server (see Section 3.2.2.1). The parameters are *location*, *search radius*, *language*, and a set of *data provider* which should be considered. The mandatory parameters are the *location* as geographic coordinates as well as at least one *data provider*. If one of these values is missing then the request will be skipped. For all other properties exists default values (for example

Attribute	Merging strategy
uuid	preserve uuid of rOld
id	preserve id of rOld
source	merge the sets of sources
latitude longitude	overwrite if rOld's coordinates are null
label	overwrite with rNew's label if it is of the same length or longer. We heuristically assume that a longer label gives more information.
description	overwrite with rNew's description if it is same length or longer. We heuristically assume that a longer description gives more information.
facets	merge the two sets of facets
schedule	carry over start and end time if they were null in rNew; merge opening hours for all days
url	overwrite if rOld's url is null
imageUrl	carry over from rNew, if rOld's imageUrl is null
moreImage	carry over from rNew, if rOld's moreImage is null
phone	carry over from rNew, if rOld's phone is null
postalAddresses ↪ city ↪ country ↪ postalCode ↪ street ↪ streetNumber	carry over those information from rNew that were not available in rOld. Note that this may be a bad idea, as it may result in two correct information sets being merged into one incorrect one. However, these cases should be rare and the majority of merges should be sensible.

Table 3.13.: The merging strategy for the attributes of Resources.

requests see Section A.4.2).

In order to ensure that the traffic is reduces to a minimum the RESTRESOURCEPROVIDER also supports gzip compression. Thus, the client has to set the 'content-encoding' property in the header of the HTTP request to 'gzip'. Thus, the server enables gzip compression during the transmission of the result. Further, the compression can reduces the traffic up to 80%.
The server sends the result incremental to the client. They are stored and serialized in a RESPONSE object (see Section 3.1.1.1). Thus, the server writes the data to the GZIPOUTPUTSTREAM object. The HTTP connection will be closed by the server if all data were transferred.
The endpoint of this interface is:

```
http://server-url/mobex-server/RestResourceProvider
```

Thus, the client has to initiate a HTTP GET operation with the required parameters. An overview of all available parameters with examples and further descriptions can be found in Table 3.14 below. Other examples can be found in Section A.4.2.

Parameter	Range of Values	Description	Example
long	floating-point number	east-west position of a point	8.895003
lat	floating-point number	north-south position of a point	49.446231
distance	floating-point number	in kilometers	3.12
lang	some string	language of the device	en_GB
search-dbpedia	Boolean	DBPedia/Genonames	true
search-eventful	Boolean	Eventful	false
search-qype	Boolean	Qype	false
search-twitter	Boolean	Twitter	false
search-gplaces	Boolean	Google Places	true
search-openpoi	Boolean	OpenPOI	true
search-lastfm	Boolean	LastFM	false
search-klicktel	Boolean	KlickTel/Telegate	true

Table 3.14.: Parameters of the RESTRESOURCEPROVIDER Interface

3.2.3.2. REST Facet Structure

The RESTFACETSTRUCTURE interface provides four static trees which are provided as a JSON document. The root nodes of these trees are `Place`, `Person`, `Organization` and `Event`. The nodes of each tree represent the available facets as well as their hierarchy. Each RESOURCE object which is delivered by the REST Resource Provider interface has a property facet. The facet property contains only values which are part of the facet structure (see Section 3.2.2.2). Furthermore, each RESOURCE object has even a property named type. This property contains as value one of the root nodes (`Place`, `Person`, `Organization` and `Event`). This enables that a client can represent the RESOURCE object grouped and hierarchical.
The trees contain together more than 1000 different facets and were build based on different data which were provided by the implemented data provider. The main part of the structure is based on data from DBPedia and Qype. At server side, the trees are partially stored as JSON document as well as in another document in triple syntax[1]. These two documents are disjoint and indeed there is no reason to store the trees in different formats. However, the reason for this is the way of the implementation of this interface by a previous team. Hence, the trees are currently not stored in one document. However, it is possible to merge these two documents. However, there was only a low priority for this task.
The client has to initiate a HTTP GET operation with the following interface to load the trees as JSON document. Parameters are not available.

```
http://server-url/mobex-server/RestFacetStructure
```

A segment of the trees is also available in the Section A.4.4.

[1] Also used in the context of SPARQL and triple stores like Virtuoso [7].

3.2.4. External Libraries

The server needs multiple external libraries for the communication with the data providers, parsing of the queried data as well as the logging of info, warning or error messages. Each library typically provides more functionality than the server uses. Thus, the 'Commons HttpClient' library is only included for a correct encoding of URIs regarding the data provider 'Mapquest' which handles geocoding and reverse geocoding requests. The 'Commons Lang' library is required for the escaping of characters of HTML entities and the other two 'Common' libraries are only dependencies of the first two. In addition to the HTML entity problem, some data providers also provide labels and descriptions which contain HTML tags. Therefore, the 'Jsoup' library is included because it provides a method to convert HTML text into a normal text.
The libraries 'Simmetrics', 'Owl2fs' and 'SQLite' are required in the context of the REST FACET STRUCTURE or FACETMANAGER. The first one is required to calculated the string similarity of two facets (see Section 3.2.2.2), the second one too build the facet trees based on an RDF document (see section 3.2.3.2) and the last one two store and read the mapping of the facets (see section 3.2.2.2).
An overview of each library, the used version, the developer, license information as well as the homepage which provides the library can be found in Table 3.15 below.

Library	Version	Developer	License
Commons Codec	1.7[1]	Apache Software Fd.	Apache License, 2.0
Commons HttpClient	3.1[2]	Apache Software Fd.	Apache License, 2.0
Commons Lang	3.1[3]	Apache Software Fd.	Apache License, 2.0
Commons Logging	1.1.1[4]	Apache Software Fd.	Apache License, 2.0
JSON	2011-02-02[5]	Douglas Crockford	-
Jsoup	1.7.2[6]	Jonathan Hedley	MIT License
Log4J	1.12.17[7]	Apache Software Fd.	Apache License, 2.0
Owl2fs	20110315[8]	twouse	Eclipse P. License, 1.0
Simmetrics	1.6.2[9]	UK Sheffield Uni.	GPLv2
SQLite JDBC	3.7.2[10]	Taro L. Saito	Apache License, 2.0
Twitter4j	3.0.3[11]	Yusuke/Community	Apache License, 2.0

Table 3.15.: Overview of the external libraries

[1] http://commons.apache.org/proper/commons-codec
[2] http://hc.apache.org/httpclient-3.x
[3] http://commons.apache.org/proper/commons-lang
[4] http://commons.apache.org/proper/commons-logging
[5] http://json.org
[6] http://jsoup.org
[7] http://logging.apache.org/log4j/1.2
[8] https://code.google.com/p/twouse
[9] http://sourceforge.net/projects/simmetrics
[10] https://bitbucket.org/xerial/sqlite-jdbc
[11] http://twitter4j.org

3.3. Future Work

3.3.1. Caching

Whenever the server receives a request from a client, it freshly requests the data from the data provider regardless of earlier requests, which might have resulted in some or all of the required data being requested. One open point in the server is thus the implementation of a sensible form of caching. For that, it is important to take into account certain technical and legal aspects:

- Data providers may have certain legal limitations to caching such as storage time limitations.
- A client may choose which data providers to query. Thus, the caching has to include the data source.
- *Entity Resolution* is expensive. It would thus make sense to include the result of *Entity Resolution* in the caching. This is not trivial, as a client can choose to include only certain providers, while a resolution result in the cache could contain data from other providers as well. It will thus be important to model the caching appropriately.

3.3.2. Current & Additional Data Providers

Currently, there is a limited set of data providers that have been implemented. Also, some data providers are prone to change their API in the future in which case the implementation of that provider's adapter in the server should be changed. One example for such possible change is the Klicktel adapter. There are speculations that the Klicktel API may be changed so that requests given a certain location will be easier. Changes to any provider's API will only reflect in that provider's adapter.

If a new data provider is to be implemented, there are several steps that have to be carried out:

- Adding new providers may require new information being added to `RESOURCE` objects that do not logically fit into the existing Resource model. In that case, the data structures in the shared package should be adapted. This package (and sub-packages) are shared with the client side.
- A request parameter has to be added on the client side. The server has to check this parameter in the `RESTRESOURCEPROVIDER`. For example, if you were to implement a provider called X, a boolean parameter `search-x` should be added to the `RESTRESOURCEPROVIDER`. The currently implemented providers may serve as an example.
- Implement an adapter class for the new provider. Adapters have to implement the interface `Callable<List<Resource>>`. An adapter's task is to query a data provider and parse the received data into Resource objects. The existing adapters may serve as an example of how to implement new providers.

- A section that calls the implemented adapter has to be added to the RESTRESOURCEPROVIDER. It has to be conditioned on the client's request so that a data provider is only called if the client requests to do so. Furthermore, the RESTRESOURCEPROVIDER does not work iteratively but in a threaded fashion. Thus, it is important that the request to a data provider is submitted to the ENTITYMANAGER. This relies on the implementation of the `Callable` interface as pointed out above.

3.3.3. Entity Resolution / Merging

The entity resolution process may not yet be optimal. Evaluations of the resolution process suggest that the weights assigned to the different fields of a resource in the scoring part of the matching process may have to be adjusted. Also, some information that are available in the resources are not yet used. One possible idea would be to not only use the distance between two resources in the precondition, but also use it as some kind of similarity metric. Instances that are closer to each other could then receive a larger bonus than those further apart.

Currently, the entity resolution process is built to find resources with information about the same real-world object and merge them. In the future, the resolution could be extended to also find relationships between different objects, e.g., a concert (as an event), organized by some agency (organization) taking place at a concert hall (place) and a singer (person) being the main actor of that event. This makes the resolution process by far more complex. If one were to extend the resolution like this, it would also make sense to include Tweets in the process. However, this would require some sort of Natural Language Processing to recognize entities mentioned in the Tweet, which may be computationally expensive, i.e. the run time of the resolution could increase significantly, which may not be acceptable for on-the-fly resolution. A simpler approach could be to just look at the hashtags in the Tweet.

Additionally, the process of merging may have its flaws. When merging postal addresses for example, we currently merge so that information from both resources may get mixed up which may result in two correct information sets being merged into one incorrect one. However, the cases where this actually happens should be rare. It should be determined, whether these cases are actually seldom enough to cause only small errors in the data the client receives. If this process of merging resources is too error-prone, it should be replaced by a better one such as taking over the address that has more information attached.

4. General

4.1. Project Management

4.1.1. Work packages

At the beginning of the project all team members started with the organization of the project itself. One of the key issues here was to clarify the goal of this team project and identify the work packages as well as the sub-goals for this project. The outcome of this can be seen in the further listing.

- Organization of the project
- Team organization and team building activities
- Use-case and User-story definition
- Reorganization and Refactoring of the program architecture
- Participation in a contest "BW Goes Mobile"[1]
- First release in the Android market
- Bug fixing
- Implementation of Logging
- Initiation and maintenance of data collection
- Analysis of the collected data
- Documentation

Many of the items listed above were gradually added afterwards, because the project team did not expect some of these issues, for instance the need to restructure the program architecture, because of unsatisfactory system performance, ongoing system failures or difficulties to extend the current system. The participation of the "BW Goes Mobile" contest also accounted for additional workload.
The work packages were used to determine the reached milestones later on.

During the implementation period the ticketing system in Redmine was intensively used. A more precise description can be found in the following section.

4.1.1.1. Ticket System

We used Redmine as project management tool and for managing all of our user stories. Within the ticketing system of Redmine, we categorized and prioritized as well as assigned all coding tasks over the entire project. We further announced meetings and tracked discovered bugs.

[1] `http://bw-goes-mobile.mfg.de/de`

At the end of this project, we assigned more than 60 user stories ("features") via this system. Nevertheless also assigned many sub-tasks without opening a new "ticket" (also called "issue").

4.1.1.2. Milestones

We defined several small goals throughout the entire project and summarized them in work packages. The different goals usually depended on each other. The table 4.1 below shows the precondition relationship of the single milestones as well as the time of realization.

no.	Milestone	depended from	finished date
1	Organization of the project	nothing	end of October 2012
2	Team organization and team building activities	no. 1	ongoing process
3	Use-case and User-story definition	no. 1	end of December 2012
4	Reorganization and Refactoring of the program architecture	no. 3	end of March 2013
5	Participation in a contest "BW Goes Mobile"	nothing	end of January 2013
6	First release in the android market	no. 4	mid of April 2013
7	Bug fixing	no. 6	end of July 2013
8	Implementation of Logging	no. 4	end of July 2013
9	Initiation and maintenance of data collection	no. 6 & 7 & 8	mid of August 2013
10	Analysis of the collected data	no. 9	mid of September 2013
11	Documentation	all milestones	end of September 2013

Table 4.1.: Overview the milestones

The first item "Project Organization" divides into installation and configuration of our server and the conduction of several analysis. Besides using Redmine as a collaboration tool and ticketing system for tracking our progress, we have also decided to use a Tomcat server environment. Furthermore, we outlined the prospective functionality of our application and did several analysis for instance a competitor analysis.

In the second step, we strengthened our team spirit during nonofficial, not work related events. For a successful team, this get-together events are essential to learn about the background and the experience of your team members.

The third step was a short one in our ongoing process. Nevertheless it was absolutely necessary to visualize and analyze the underlying requirements. We conducted brainstorming sessions about the later capabilities of the app, created use-cases and derived user stories from it. On this basis we were also modeling our data structure for the entire client-server-architecture.

After we analyzed the already existing code and architecture, we decided to use nearly three months to reorganize and rewrite parts of the current program. This section also included the implementation of a dynamic facet tree and the integration of log4j on the server side. The Java version has been updated from version 6 to 7 and a Jenkins build server has been installed. On the client side, the Android Google Maps API has been updated which resulted in a complete rewriting of the affected parts. The software architecture has been cleaned up and modularized and an event-based approach for inter-component communication has been implemented.

The participation in the contest "BW Goes Mobile" was taken out of an impulse and was actually not a purpose of this project. Nevertheless this decision to participate caused work during the later ongoing progress.

The app publishing in the Android-store was delayed because of the necessary restructuring of the app. In this work package, we prepared our release as well as a description, screenshots, terms of use and other information material.

The bug fixing sessions started after the first answers of the first questionnaire (see evaluation paper) have been received from our focus group. The bug fixing process took approximately 2 months. The implementation of the necessary logging (see evaluation paper) that was used for the subsequent data collection process was also done during this step. The whole evaluation process is described in the evaluation paper.

4.1.2. Work Schedule

At the beginning of this team project we agreed upon regular meetings, usually twice a week. During the meetings which lasted about 1,5 hours, we discussed the next steps of our ongoing project. The minute taker as well as the moderator of a meeting rotated in round robin fashion over time. We used these sessions for checking the past process, allocated new workload, specified our outcome and discussed difficulties or issues. We used Google-Hangout [2] for our online meeting during our semester breaks. Physical meetings were held in a seminar room at the university.

4.2. Continuous Integration

Martin Fowler defines continuous integration as "a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible." [5] Thus, a project that follows the guidelines of continuous integration has the benefit to hold an integrated and up-to-date version of the current software ready to test and possibly ready to deploy. Integration

[2] https://plus.google.com/hangouts

problems can be avoided or are identified very early in contrast to traditional software development practices. We followed the principles of continuous integration in order to benefit from those advantages.

4.2.1. Code Repository

The client and server projects are stored in Git repositories[3]. For the client and server project, there is exactly one central repository which enables to store the current state of development and provide it to the other members. The shared package between client and server has to be synchronized manually. Each member has a local working copy of the repositories that is also a backup of the central repository. The repositories are administrated by Git because it offers numerous advantages compared to SVN including decentralization and a better branching and merging support.

Branching

A repository comprises of at least one branch. The client and server repositories are using a specific branch model. Hence, there exist three permanent as well as multiple temporary branches. The permanent branches are called `master`, `development` and `experimental` and comprise different conditions of the project. The project comprise an Eclipse workspace. However, a temporary branch is always derived form the experimental branch and will be created for each new feature which should be implemented, if the implementation is complete then the temporary branch will merged into the experimental branch. When an temporary branch is no longer needed, then it will be closed. The experimental branch is only used by the developer of the project and should not be used by other projects. Once, the experimental branch does not contain any critical issues it will merged into the development branch. The development branch can be used by other projects in order to verify the stability. Finally, the development branch will be merged into the master branch which is used by the end-user, if for a certain time no problems occur. For further details about the branching model and instructions on how to excecute certain steps of the model (e.g. preparing a new release), please see `http://nvie.com/posts/a-successful-git-branching-model/`.

4.2.2. Building

4.2.2.1. Automation

Build automation is a key element of continuous integration [5]. It covers automated compiling and packaging of the source code. To facilitate this process we use Jenkins, a continuous integration tool that runs on our development server and can be accessed and configured using a web interface[4]. Compiling and packaging is triggered whenever a new commit is pushed on the server's source code repository. This way, a developer can immediately see if his changes are free of compiling errors. Moreover Jenkins is configured to send an Email out to the

[3] git@mobex.ctlnode.de:REPOSITORY-NAME
[4] `http://server-url:8080`

developers if any errors occur which enforces to have an error-free version of the software at all times. The automatic build process is currently only configured for the client project. Thus, the server build process has to be triggered manually, because not all versions should be accessable by other projects, to avoid conflicts.

4.2.2.2. Distribution

After the automated building, the software artifact is either deployed on an application server or installed on mobile devices, depending on client or server.
For the distribution of the client APK, which is the result of the compiling and packaging process, we configured the build process to move the APK file to a directory on the server that is accessible via a website[5]. That way, the recent APK can be easily downloaded and installed using a smartphone. The corresponding commands are in the *post-build* section of the *build.xml* script. To ensure that an existing APK file is not overwritten whenever a new file is moved, the filename always contains a consecutive version number.
The server will be automatically deployed after the building process on the same machine by Tomcat. The implemented REST API of the server enables the possiblity to communicate with it after the deploying process (see Section 3.2.3.1). Typically, an already deployed version of the server will be updated, whereby the URL of the endpoint does not change. This means that the client uses automatically the newest version. Moreover, the server project consists of three permanent branches (see Section 4.2.1). Thererfore, there exist three different endpoints[6] which can be used.

4.2.3. Releasing of Client and Server

4.2.3.1. Server

A new version of the server will published if critical bugs were fixed or a new feature was implemented. The conditions that have to be met for a release were already described in detail in section 4.2.1. Futhermore, it is not necessary to change the build script or any other config file. Hence, only the build process has to be triggered manually (see section 4.2.2.2).

4.2.3.2. Client

To prepare a new release that can be published on Google Play, we follow the guidelines of the specified Branching model. After a new branch for the release has been created, several changes have to be made in the code.

1. Changing the server URL in the class `Server` from the URL of the development server to the one of the production server.

2. Disabling the log by setting the log level to *None* in the class `Log`.

[5] `http://server-url/mobexapk/`
[6] `http://server-url:8081/mobex-BRANCH/`

3. Incrementing the version number and version code in the files `AndroidManifest.xml` and `AndroidManifest.xml.tpl`.

4. Changing the path in the build script (`build.xml`) where the final APK file is moved to. This property is found under the *post-build* target.

5. Incrementing the version number in the Jenkins build job configuration to make sure the final APK file is correctly named.

Subsequently, the release branch should be tagged and merged into the master branch. Jenkins will build an APK that is ready to be released on Google Play using the Webinterface. Ideally the release branch is also merged back into the development branch to add the tag and possible bugfixes that were made prior to releasing also to the development version. After merging, server URL, log and build script need to be changed.
Please note that the build process automatically inserts the correct Google Maps API key for the release APK. The API key is found in the `AndroidManifest.xml`. To insert the correct API key, we created an exact copy of the manifest file (`AndroidManifest.xml.tpl`) except for the API key where we inserted a placeholder. The build script reads the API key from the `ant.properties` file and inserts it into the placeholder. The signing process of the APK is also done automatically during the build process. The required parameters are stored in the `ant.properties` file and the keystore for the release APK is found in the root directory of the client project.

4.3. Events and Contests

In the course of the development of the MobEx (mobile exploration) app, it was presented on various occasions and by various means:

- The castle party ("Schlossfest") of the University of Mannheim on September 8th 2013
- At the award presentation ceremony of the *Baden-Württemberg goes Mobile* contest (BW contest) on October 17th 2013
- Distribution of flyers at the University of Mannheim in May 2013
- Continuous representation on a Facebook page

On the occasion of the castle party of the University of Mannheim, which is an anually recurring event, we created a video presentation of the app, which can be found on the video platform YouTube in both German (`https://www.youtube.com/watch?v=mlF4hh8qTuY`) and English (`https://www.youtube.com/watch?v=xdfUMAS9CNY`). This video was directly shown to the visitors of the castle party by the project developers in an interactive presentation.

Thanks to the Telegate corporation, we received a special honor in connection to this project: the *Klicktel Award 2013* – which was awarded for a project in mobile applications – went to

Figure 4.1.: Team MobEx. From left to right: Bernd Pfister, Christian Bikar, Bernd Opitz, Michael Jess, Timo Sztyler, Florian Knip, Ansgar Scherp.

the MobEx project. Among others, we thus have the Telegate corporation to thank for being present and findable on Google Play. During the final presentation of the closely connected BW contest, we had the chance to present our app in front of an audience of representatives of businesses as well as other researchers. This also gave us the chance to socialize.

Previous to these presentations, we put around 1000 flyers on display in some lecture halls at the university, which raised the awareness of our first market-ready prototype.

Furthermore, we created and maintained a Facebook page, which promoted update information of the app and upcoming events and also presents the app as far as possible. The Facebook can be reached at `https://www.facebook.com/mobileExploration`.

Bibliography

[1] U. S. Census Bureau. *Geographic Information Systems FAQ*, chapter What is the best way to calculate the great circle distance. Movable Type Ltd., `http://www.movable-type.co.uk/scripts/gis-faq-5.1.html`, 1997.

[2] Jon Byous. *Java Technology: The Early Years*, chapter -. Sun Microsystems, `http://web.archive.org/web/20100105045840/http://java.sun.com/features/1998/05/birthday.html`, 2003.

[3] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

[4] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[5] Martin Fowler. Continuous integration, 2006.

[6] Jakob Huber, Timo Sztyler, Jan Noessner, and Christian Meilicke. CODI: Combinatorial optimization for data integration–results for OAEI 2011. *Ontology Matching*, page 134, 2011.

[7] Markus Krötzsch, Sebastian Rudolph, and York Sure. *Semantic Web - Grundlagen*. Springer London, Limited, London, 2008.

[8] Vladimir Iosifovich Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[9] Dirk Louis and Peter Mueller. *Java 7*. Pearson Deutschland, GmbH, München, 2012.

[10] Scott Main. *Say Goodbye to the Menu Button*, chapter Two Demos That Changed the World. Android Developers Blog, `http://android-developers.blogspot.de/2012/01/say-goodbye-to-menu-button.html`, 2012.

[11] Axel-Cyrille Ngonga Ngomo and Sören Auer. Limes: a time-efficient approach for large-scale link discovery on the web of data. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, pages 2312–2317. AAAI Press, 2011.

[12] Bernd Opitz, Timo Sztyler, Michael Jess, Christian Bikar, Bernd Pfister, Florian Knip, and Ansgar Scherp. Entity resolution - an incremental approach. In -, 2013.

[13] Mark Schneider, Ansgar Scherp, and Jochen Hunz. A comparative user study of faceted search in large data hierarchies on mobile devices. -, 2013.

[14] B. P. Shumaker and R. W. Sinnott. Astronomical computing: 1. computing under the open sky. 2. virtues of the haversine. *Sky and telescope*, 68:158–159, 1984.

[15] Helmut Vonhoegen. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1 edition, 1994.

[16] Helmut Vonhoegen. *Einstieg in XML.* Galileo Press, Bonn, 6 edition, 2011.

[17] Wikipedia. *HSL and HSV*, chapter Basic principle. Wikipedia, `http://en.wikipedia.org/wiki/HSL_and_HSV`, 2013.

A. Appendix

A.1. Installation

This section describes the installation of the development environment for client and server. The instructions are also described in the attached powerpoint presentation 'Einrichten der Android Entwicklungsumgebung.pptx'.

A.1.1. Client

The mobEx-client has been developed using the Android SDK and Android Development Tools provided by Google. The Android SDK Manager[1] can be used to download and install all necessary software and tools. At least the 'SDK Tools', 'SDK Platform-Tools', 'Android 2.3.3 SDK', 'Android 2.3.3 Google APIs' and 'Google USB Driver' should be installed. As IDE, Eclipse is recommended as Google provides helpful plugins to support the development[2]. To import the android project (i.e mobex-client) into eclipse go into 'New -> Project -> Android -> Android Project From Existing Code', select the 'mobex-client' folder and check 'copy projects into workspace'. After that, check that the properties of the project are correct (right-click on the project -> properties). In the Android section the Google API Version 2.3.3 should be selected and in the Java Compiler section select 'Compiler level 1.6'.

A.1.2. Server

For server development we recommend using Eclipse and a Tomcat application server (preferably version 7). After cloning the project from the repository into the workspace folder it can be imported by selecting 'New -> Project -> Dynamic Web Project'. As project name use 'mobex-server' and either select an existing tomcat 7 runtime or create a new one. After a click on 'finish', the project appears in the workspace and can be launched via right-click 'Run As -> Run on Server'.

---

[1] `http://developer.android.com/sdk/index.html`
[2] `http://developer.android.com/sdk/installing-adt.html`

A.2. Project Data

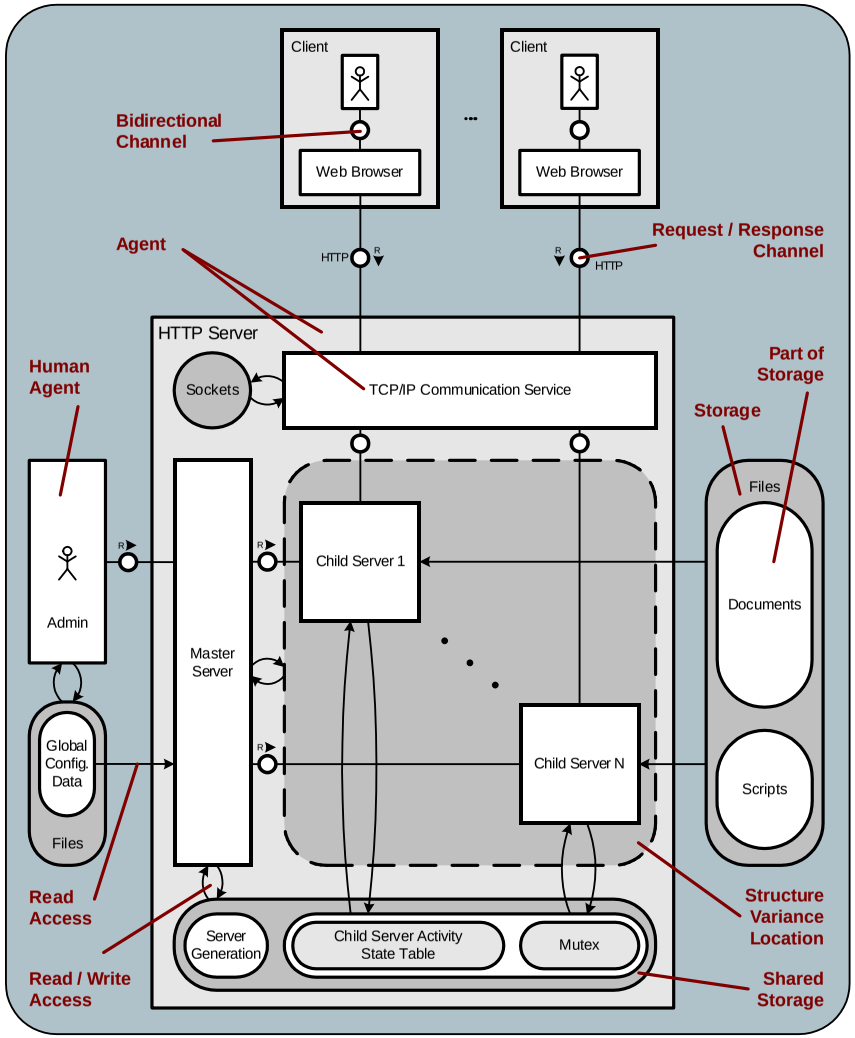
All project data are stored on an USB flash drive which is attached to this documentation. The directory structure is as follows:

- Documentation
  - Analyses & UseCases
    - Data Model
    - Graphical User Interface
    - Related Work
    - Traffic Compression
  - BW Challenge
  - Evaluation
  - Marketing
  - Meetings
    - Introduction
    - Members
    - Protocols
  - MFacets
  - Server
    - Data Provider
    - Git
    - Jenkins
- Event
  - Schlossfest
  - Telegate & Klicktel
- Media
  - Pictures & Screenshots
  - Videos
- Paper
- Server & Project Data
  - Git Repositories
    - mobex-client
    - mobex-server
    - mobex-techdoku
  - MFactes (Predecessor of MobEx)
  - MobEx APK
    - nightly
    - release
  - Server Virtual Machine
    - mobex.ctlnode.de

A.3. Fundamental Modeling Concepts

Compositional Structures

Block diagrams - Reference Sheet



FMC Block diagrams show the compositional structures as a composition of collaborating system components.

There are active system components called agents and passive system components called locations. Each agent processes information and thus serves a well-defined purpose. Therefore an agent stores information in storages and communicates via channels or shared storages with other agents. Channels and storages are (virtual) locations where information can be observed.

basic elements		
A	active system component : agent, human agent	Serves a well-defined purpose and therefore has access to adjacent passive system components and only those may be connected to it. A human agent is an active system component exactly like an agent but the only difference that it depicts a human. (Note 1: nouns should be used for identifier "A" Note 2: do not need to be depicted as rectangle or square but has to be angular)
S	passive system component (location) : storage, channel	A storage is used by agents to store data. (Note: do not need to be depicted as ellipse or circle but has to be rounded) A channel is used for communication purposes between at least two active system components. (Note: channels are usually depicted as smaller circles but may also vary like the graphical representation of storage places)
	access type	Directed and undirected edges represent the kind of access an active system component has to a passive system component. The types of access are read access, write access and a combination of both. (Note: usually undirected edges depicting read/write access are used on channels whereas two directed edges also depicting read/write access are used on storages)
common structures		
A S	read access	Agent A has read access to storage S.
A S	write access	Agent A has write access to storage S. In case of writing all information stored in S is overwritten.
A S	read / write access (modifying access)	Agent A has modifying access to storage S. That means that some particular information of S can be changed.
A1 A2	unidirectional communication channel	Information can only be passed from agent A1 to agent A2.
A1 A2	bidirectional communication channel	Information can be exchanged in both directions (from agent A1 to agent A2 and vice versa).
A1 A2 A1 A2	request / response communication channel (detailed and abbreviation)	Agent A1 can request information from agent A2 which in turn responds (e.g. function calls or http request/responses). Because it is very common, the lower figure shows an abbreviation of the request/response channel.
A1 S A2	shared storage	Agent A1 and agent A2 can communicate via the shared storage S much like bidirectional communication channels.
advanced		
A1 A2	structure variance	Structure variance deals with the creation and disappearance of system components. An agent (A1) changes the system structure (creation/deletion of A2) at a location depicted as dotted storage. System structure change is depicted as modifying access. After creation agent A1 can communicate with agent A2 or vice versa.

A.4. Early Drafts

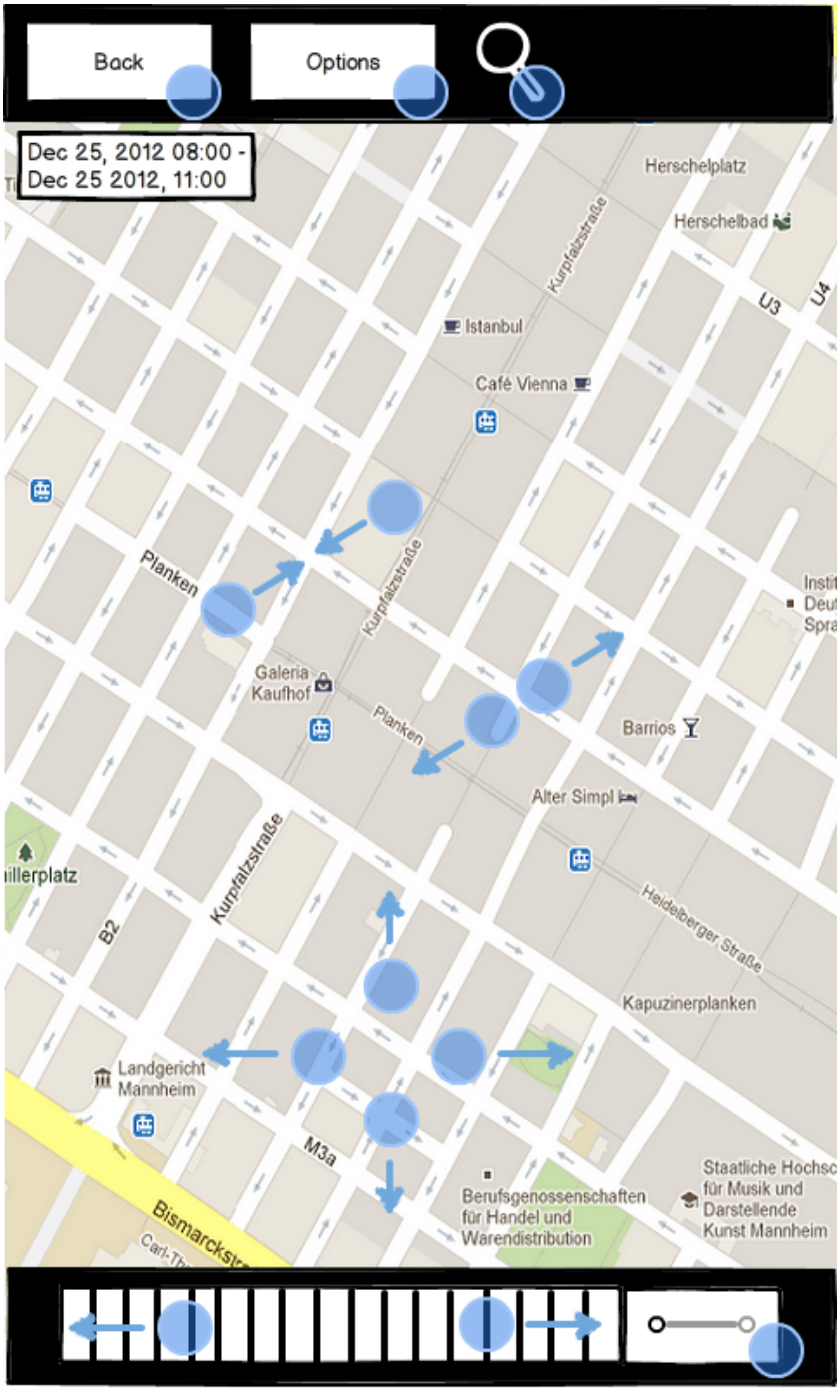


Figure A.1.: Early map GUI draft with a button for toggling between time period start and end (bottom right)

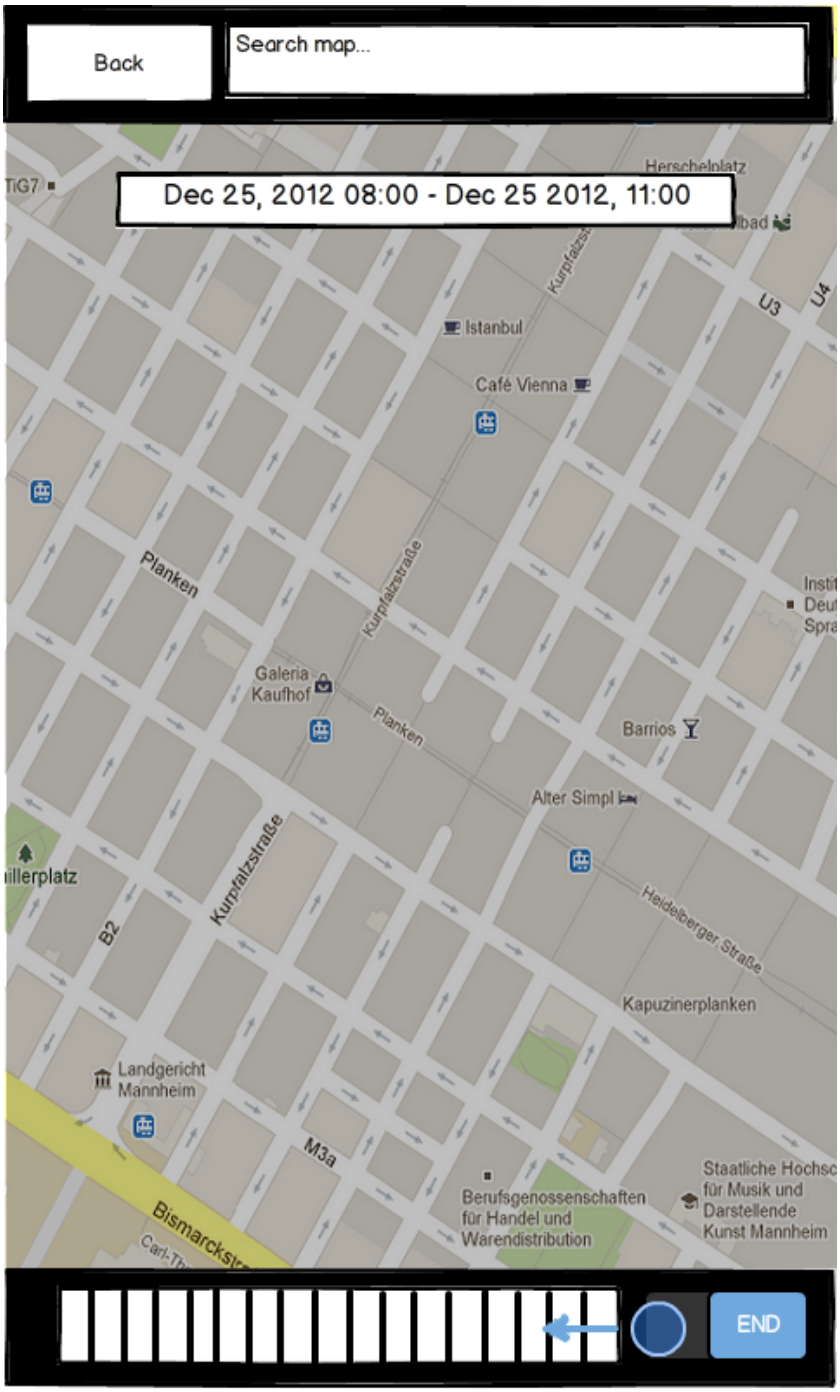


Figure A.2.: Early map GUI draft with a switch for toggling between time period start and end (bottom right). The top action bar in this draft is in search mode.

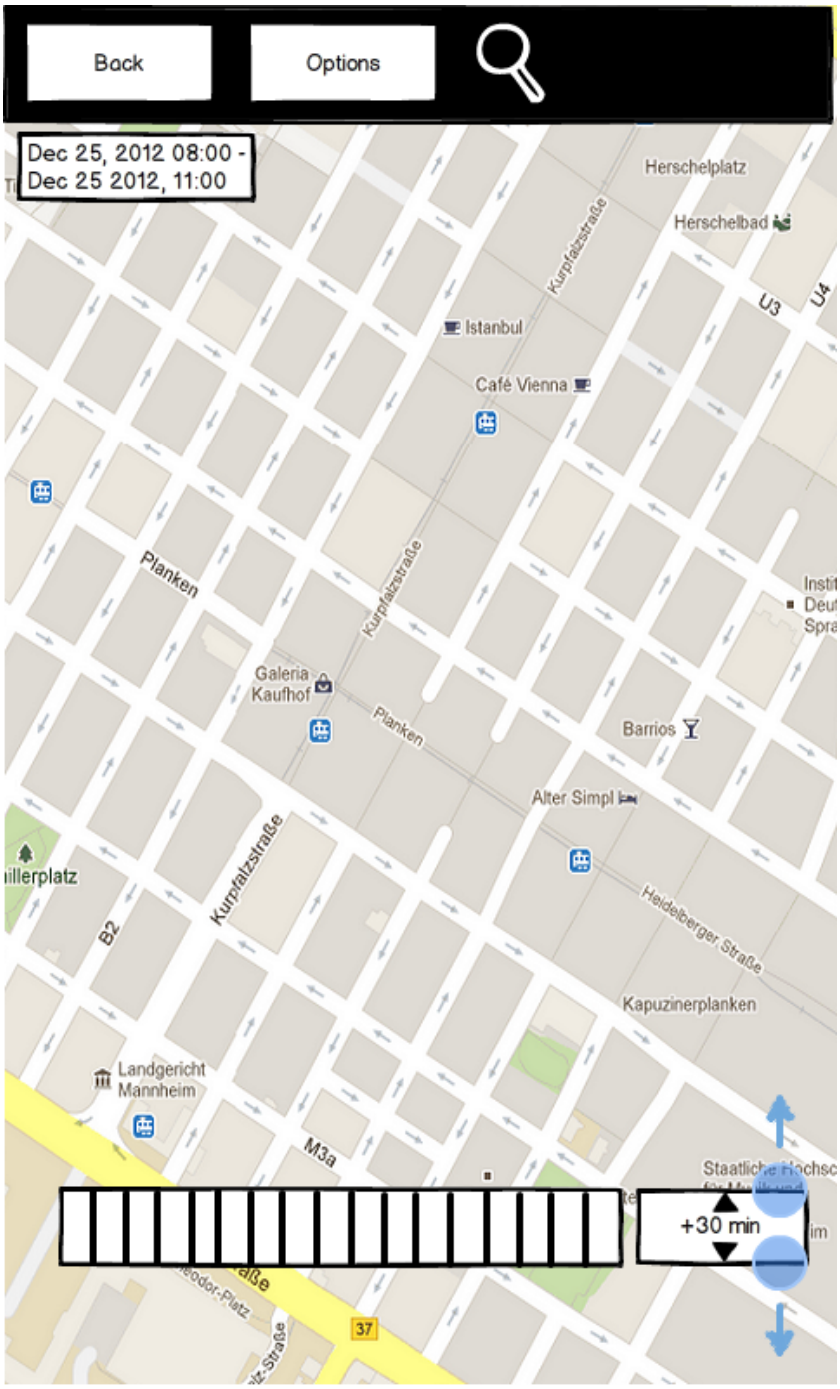


Figure A.3.: Early map GUI draft with a single time slider and a control for adjusting a point in time relative to it (bottom right)

A.4.1. Demo Client

```
1  package de.unima.mobex.server.test;
2
3  import java.io.EOFException;
4  import java.io.ObjectInputStream;
5  import java.net.HttpURLConnection;
6  import java.net.URL;
7  import java.util.HashSet;
8  import java.util.Set;
9  import java.util.zip.GZIPInputStream;
10
11 import de.unima.mobex.shared.Response;
12 import de.unima.mobex.shared.model.Resource;
13
14 /**
15  * Copyright 2012/2013 (C) Team MobEx
16  *
17  * Demo client for testing the mobex server
18  *
19  * @author Bernd Opitz, Bernd Pfister, Timo Sztyler
20  * @version 2.0
21  *
22  */
23 public class Client
24 {
25    public static void main(String[] args) throws Exception
26    {
27       URL url = new URL("http://localhost:8080/mobex-server/
             RestResourceProvider?long=8.46469&lat=49.4832&distance=3.1&
             search-dbpedia=true&search-eventful=true&search-qype=true&
             search-twitter=true&search-gplaces=true&search-openpoi=true
             &search-lastfm=true&search-klicktel=true&lang=de_DE");
28
29       HttpURLConnection conn = (HttpURLConnection) url.
             openConnection();
30       conn.addRequestProperty("Accept-Encoding", "gzip");
31       conn.setRequestMethod("GET");
32       conn.setUseCaches(false);
33
34       Set<Resource> insertion = new HashSet<Resource>();
35       Set<Resource> deletion = new HashSet<Resource>();
36
37       ObjectInputStream inputStream = null;
38       try {
```

```
39          System.out.println("Content Encoding: " + conn.
               getContentEncoding());
40
41          if("gzip".equals(conn.getContentEncoding())) {
42             System.out.println("Server answered with gzip compressed
                   stream.");
43             inputStream = new ObjectInputStream(new GZIPInputStream(
                  conn.getInputStream()));
44          } else {
45             inputStream = new ObjectInputStream(conn.getInputStream
                  ());
46          }
47
48          Object obj = null;
49
50          while((obj = inputStream.readObject()) != null) {
51             if(obj instanceof Response) {
52                Response resp = (Response) obj;
53
54                Set<Resource> tmp = new HashSet<Resource>();
55                tmp.addAll(resp.getInsertions());
56
57                // this is all necessary
58                deletion.addAll(resp.getDeletions());
59                tmp.addAll(deletion);
60                tmp.removeAll(deletion);
61                insertion.removeAll(tmp);
62                insertion.addAll(tmp);
63             }
64          }
65       } catch(EOFException e) {
66          System.out.println("End of Inputstream");
67       }
68
69       for(Resource res : insertion) {
70          System.out.println(res.getLabel());
71          System.out.println(res.getSchedule());
72       }
73       System.out.println("Size Of Result: " + insertion.size());
74    }
75 }
```

A.4.2. Example Requests

The interface of the RESTRESOURCEPROVIDER was already described in Section 3.2.3.1. In the following, there are some HTTP request examples for different cities:

Mannheim

```
http://localhost:8080/mobex-server/
    RestResourceProvider?
        long=8.46469&
        lat=49.4832&
        distance=3.1&
        search-dbpedia=true&
        search-eventful=true&
        search-qype=true&
        search-twitter=false&
        search-gplaces=true&
        search-openpoi=true&
        search-lastfm=true&
        search-klicktel=true&
        lang=en_US
```

New York

```
http://localhost:8080/mobex-server/
    RestResourceProvider?
        long=-74.005972&
        lat=40.714393&
        distance=2.3&
        search-dbpedia=true&
        search-eventful=true&
        search-qype=true&
        search-twitter=false&
        search-gplaces=true&
        search-openpoi=true&
        search-lastfm=true&
        search-klicktel=false&
        lang=de_DE
```

A.4.3. Example Documents

A.4.4. Facet Trees

```
{
        "name": "root",
        "children": [{
                "name": "Event",
                "children": [{
                        "name": "Space Mission"
                },
                {
                        "name": "Sports Event",
                        "children": [{
                                "name": "Grand Prix"
                        },
                        {
                                "name": "Mixed Martial Arts Event"
                        },
                        ...]
                },
                ...]
        },
        {
                "name": "Organisation",
                "children": [{
                        "name": "Company",
                        "children": [...]
                },
                ...
        },
        {
                "name": "Person",
                "children": [...]
        },
        {
                "name": "Place",
                "children": [...]
        }]]
}
```

Listing A.1: An excerpt of the facet tree provided by the server. The full tree fills more than 30 pages.

A.4.5. Usecases

mobEx
Erster Start

Level: User Goal

User: Application User

Stakeholders and interest:

User: Der Nutzer hat Interesse daran Lokationen, Institutionen, Plätze, POI, Diskotheken und Events in der lokalen Umgebung zu finden.

Universität: Will das Nutzungsverhalten und den Einfluss der Mobiltelefone auf das soziale Leben der User untersuchen.

Precondition: Das Android Betriebssystem ist erfolgreich gestartet und wird ausgeführt. Der mobEx Client ist bereits installiert und seit erfolgreicher Installation noch nicht gestartet worden.

Postcondition: Suche mit dem mobEx Client wird gestartet.

Main success scenario:

1. Nutzer startet den mobEx Client auf seinem Smartphone
2. Eine kurze Screenshot Anleitung in Form eines Erststartassistenten wird eingeblendet.
3. Der Nutzer erhält Informationen zur Datenerhebung und Sammlung personenbezogener Daten und Nutzungsverhalten. Dabei erscheint der Hinweis auf wissenschaftliche Untersuchungen der Uni Mannheim.
4. Er bekommt die Möglichkeit diese Zuzustimmen oder abzulehnen. Default Auswahl ist die Zustimmung.
5.

Extensions:

*a. zu jedem Zeitpunkt: „Optionen“ sollen anwählbar sein

Die Schaltfläche zum Schließen der App kann angewählt werden

2a. Der Assistent ist jederzeit abbrechbar

Special requirements:

Stabiler Systemstart und keine Abbrüche während der Laufzeit sind in der Mehrzahl aller Fälle zu gewährleisten.

Frequence of occurrence: einmalig

mobEx
Suchen über Facettenmodus

Level: User Goal

User: Application User

Stakeholders and interest:

User: Der Nutzer hat Interesse daran Lokationen, Institutionen, Plätze, POI, Diskotheken und Events in der lokalen Umgebung zu finden.

Universität: Will das Nutzungsverhalten und den Einfluss der Mobiltelefone auf das soziale Leben der User untersuchen.

Google: Muss folgende Voraussetzungen erfüllen um im Markt zugelassen zu werden.

Precondition: Das Android Betriebssystem ist erfolgreich gestartet und wird ausgeführt. Der mobEx client ist bereits installiert und mindestens einmal gestartet worden.

Postcondition: Der mobEx Client übergibt die Zielkoordinaten dem Navigationsprogramm

Main success scenario:

1. Nutzer startet den mobEx client auf seinem Smartphone
2. Der Nutzer wird auf den GRID-Bildschirm geleitet und beginnt seine Suche über Eingrenzung der Kategorien
3. Verfeinert seine Suche über das Facettenmuster bei jedem Klick. Er sieht somit alle Orte der Umgebung, eingegrenzt durch seine bisherige Auswahl
4. Der Nutzer wählt einen der im oberen Feld vorgeschlagenen Lokationen an und erhält ein Informationsfeld zu dieser Lokation in einem definierten Template festgelegtem Layout.
5. Der Nutzer hat die Möglichkeit sich durch einen weiteren Tastendruck dorthin navigieren zu lassen soweit Ortsinformationen vorhanden sind. Dabei startet das Programm die Navigationssoftware des Smartphones

Extensions:

*a. zu jedem Zeitpunkt: „Optionen“ sollen anwählbar sein

Es soll jederzeit auf Kartenmodus gewechselt werden können

Die aktuelle Auswahl soll auf Kartenmodus angezeigt werden

Das Konfigurieren eines anderen Gebietes oder Stadt soll in dem Kartenmodus anwählbar sein.

Der Radius in welchem eine Lokalität, Institution oder Person gesucht werden soll, soll unter dem Menüpunkt Optionen anwählbar sein oder durch den Kartenausschnitt definiert werden.

2a. In der ersten Ebene werden die vier Hauptkategorien angezeigt.

3a. bei falscher Auswahl der letzten Facette geht man auf den vorherigen Stand zurück indem man auf die Mitte des Facettenblocks klickt.

3b. Die Suche kann weniger strikt eingegrenzt werden wenn mehrere Facetten gleichzeitig angewählt werden sollen indem die übergeordnete Facette länger als 1 Sekunde angewählt wird. Diese Auswahl ist in jedem Fall als additive Auswahl zu verstehen also (a v b) nicht als (a ∧ b).

3c. Bei Anwahl des Facettenpunkt „weitere Facetten“ wird eine alphabetische Sortierung aller Facetten vorgenommen werden. Dabei sollen Kategorien, die lediglich einen Eintrag enthalten nicht in der Anzeige zu finden sein sondern ausschließlich über explizite Suche des Schlüsselwortes dem Nutzer gezeigt werden.

Special requirements:

Stabiler Systemstart und keine Abbrüche während der Laufzeit sind in der Mehrzahl aller Fälle zu gewährleisten.

Die Auswahlanzeige muss mit jeder weiteren Eingrenzung aktualisiert werden.

Frequence of occurrence: Suchen sollten kontinuierlich wiederholt werden können

mobEx
Suchen über Kartenmodus

Level: User Goal

User: Application User

Stakeholders and interest:

User: Der Nutzer hat Interesse daran Lokationen, Institutionen, Plätze, POI, Diskotheken und Events in der lokalen Umgebung zu finden.

Universität: Will das Nutzungsverhalten und den Einfluss der Mobiltelefone auf das soziale Leben der User untersuchen.

Google: Muss folgende Voraussetzungen erfüllen um im Markt zugelassen zu werden.

Precondition: Das Android Betriebssystem ist erfolgreich gestartet und wird ausgeführt. Der mobEx client ist bereits installiert und mindestens einmal auf dem entsprechenden Gerät gestartet worden.

Postcondition: Der mobEx client übergibt die Zielkoordinaten dem Navigationsprogramm

Main success scenario:

1. Nutzer startet den mobEx client auf seinem Smartphone
2. Der Nutzer wird auf den GRID-Bildschirm geleitet welcher im linken oberen Eck einen Link zum Kartenmodus enthält.
3. Der Kartenmodus wird geladen mit seiner Position im Zentrum. Der Zoom ist so voreingestellt, das der Nahbereich erfasst wird.
4. Das aktuelle Datum / Uhrzeit ist voreingestellt
5. Es werden die Events auf dem sichtbaren Teil der Karte eingezeichnet welche anwählbar sind
6. Der Nutzer erhält weitere Informationen dieses Ortes, soweit er auf das Symbol auf der Karte klickt. Sollten mehrere Events / Elemente auf dem gleichen Ort vermerkt sein wird eine Auflistung aller Events angezeigt sobald er diesen Ort anwählt.
7. Er erhält ein Informationsfeld mit weiteren Informationen zu einem bestimmten Event in Form eines vor definierten Templates.
8. Der Nutzer hat die Möglichkeit sich durch einen weiteren Tastendruck dorthin navigieren zu lassen. Dabei startet das Programm die Navigationssoftware des Smartphones.

Extensions:

*a. zu jedem Zeitpunkt: „Optionen“ sollen anwählbar sein

Es soll jederzeit auf Facettenmodus gewechselt werden können

Das Konfigurieren eines anderen Gebietes oder Stadt soll direkt anwählbar sein.

Die Karte wird bei jedem Zoomwechsel welchen den Ansichtsbereich vergrößert oder verschiebt nachgeladen.

Der Radius in welchem eine Lokalität, Institution oder Person gesucht werden soll wird durch den Bildausschnitt bestimmt.

2a. Durch raus und rein zoom Gesten wird die Detailschärfe und der Bildausschnitt im Kartenmodus verändert. Die Elemente mit Events im Sichtfeld zzgl. 50 % der nicht mehr sichtbaren Randbereiche werden asynchron aktualisiert.

2b. Durch ziehen der Karte kann dessen Ausschnitt verändert werden und für benachbarte Regionen gesucht werden.

3a. bei falscher Auswahl kann der aktuelle Zeitpunkt durch Doppelklick auf die Datumsanzeige wieder erreicht werden.

5a. Das Datum und die Uhrzeit auf die sich die Ansicht bezieht kann durch Tasten des im oberen Bereich eingeblendeten Datums bzw. durch den neben dem time wheel befindlichen Button geändert werden. Nach einer Änderung wird die ausgewählte Zeit kurzfristig in der Bildschirmmitte eingeblendet.

Special requirements:

Stabiler Systemstart und keine Abbrüche während der Laufzeit sind in der Mehrzahl aller Fälle zu gewährleisten.

Frequence of occurrence: Suchen sollten kontinuierlich wiederholt werden können