BSc project IN3405

Socialocalize

Version 0.2

December 10th 2008

Universities

Delft University of Technology
Faculty of Electric Engineering, Mathematics and Computer Science (EEMC)

Osaka University
Cyber Media Center (CMC)

Team members

Gosse Bouma	1220217
Peter de Klerk	1228188
Kristian Slabbekoorn	1228196

Instructing party

Y. Teranishi	- Associate professor at Osaka University

Commission

Ir. B.R. Sodoyer	- BSc project coordinator at Delft University of Technology
J. de Vries	- International coordinator at Delft University of Technology

Preface

At Delft University of Technology, the final course of the computer science bachelor's program is called the Bachelor (BSc) Project. In this course students should bring all that they've learned in the past three years into practice. The bachelor project is concluded with this final report.

Students Gosse Bouma, Peter de Klerk and Kristian Slabbekoorn have worked on the BSc Project at the Osaka University under supervision of Mr. Teranishi. We would like to thank him for his guidance.

Mr. Sodoyer and Mr. De Vries of Delft University of Technology also helped us before and during our project, we would like to thank them for their cooperation.

Contents

Summary

In Japan, most cell phones have the capability of receiving GPS signals. Besides the cell phone there are many other devices that can keep track of GPS data, including the GPS device made by Sony. If all this GPS data is collected and organized, one could get a good overview of people's whereabouts.

That is why the goal of this project was to develop a tool that keeps track of people's movements, share a visualization of these movements with other people and allow for other means of social interaction, such as chatting, messaging, finding other people and so on.

To realize this goal we created the Sococa system. The name stems from the full name we had devised at first, **Soc**ial**oca**lize. Additionally, the Japanese phrase 'sokoka' can be taken to mean something akin to 'over there?'.

The system consists of a client part and a server part. The client program can parse and upload users' locations to a server in real time. The server has an interface which also allows other programs to upload their location data. Because the user doesn't always have an internet connection available, it is also possible to upload files with location data at a later time. This can be done on the website or with the client program.

The server hosts a website on which a map with all positions of the users are shown. When a user uploads a new location to the server, this is directly updated on the map. The map also has the capability to show animated routes the user took in the past.

The website has some social networking features. It is possible to create an account and add some personal information to it. It is possible to look up one's friends using the search option and add them as friend on the website. After they are added, users can chat in the global chat or send private messages to each other.

To ensure the users' privacy, the user can choose to limit the availability of his location data to his friends or only to himself.

The most important things the current system is lacking are full OpenId support and support for the Opensocial API. OpenId is a service which allows users to have only one account for the whole internet. This technology is supported by the website, but not by the client program. The OpenSocial API links social networking sites to each other, i.e. friends can be easily imported from one site to another. Because of the complexity of this technology, we have chosen not to implement this.

Introduction

In modern society more and more devices have the capability of tracking GPS data. GPS data can be used for different kinds of purposes, but it is generally used to determine the location of some individual or object. Especially in Japan, where we are developing this project, the cell phone is often used as a navigation device. People can now easily learn their own location, but they might also want to know where their friends are located. Currently, there are tools for this purpose, but those are in early development, hence their functionality is still limited. In this era of Web 2.0 technologies and social networking, a good utility providing functionality for tracking friends' locations is necessary.

The goal of this project is to develop such a tool which keeps track of people's movement. Using this tool, users should be able to upload their position received from some location detection device. With this location data the tool should display all users on a map. Using this map, users can easily check where their friends are located. Another feature is showing the user's movements in the past. The location data will be uploaded via the internet and stored in a database. All location data will be viewable on a portal website.

In chapter 1 the assignment is described, in chapter 2 an analysis is made of the current situation. The design is worked out in chapter 3 while chapter 4 describes the implementation. Conclusions and experiences are given in chapters 5 and 6. Chapter 7 is about recommendations for extending and improving our system. This thesis is concluded with a bibliography and two appendices. The first appendix contains the RAD document and the second the prior research report.

1 Assignment

In section 1.1 the assignment is introduced. In section 1.2 a summary of the plan of action given and finally, in section 1.3, the technologies used during the implementation of the system are introduced.

1.1 Assignment description

The goal of this BSc project is to develop a tool that can keep track of people's movement, share a visualization of these movements with other people and allow for other means of social interaction, such as chatting, messaging, finding other people and so on.

Users of the system should be able to input their (real-time) location data in various ways. For example, the standard used in Sony GPS devices, NMEA 0183, and XML. This location data should be saved in a database and users of the system should be able to view their movements via the website. On the website, the user can also keep track of the movements of their friends.

In addition, the location data should also be available for use by sample applications like an e-mail client add-on (Zimbra zimlet), a web log, a photo service and a sight recommendation service. For this purpose, the system should provide an API from which other applications can call functions.

The main focus will be on displaying the data on the website in a correct way. The only sample application with priority is the Zimbra add-on. This add-on should display the latest location of the sender of an email to the user.

1.2 Plan of action

The project is done in 4.5 months; from September 1st to January 12th. The project planning is as follows:

- Week 1-3: The requirements will be gathered using interviewing. At the end of this period a requirements document is delivered to the parties involved.
- Week 7: The system design documents will be delivered and presented to the parties involved
- Week 15: A prototype implementation will be delivered and presented to the parties involved
- Week 19: The final report is delivered and a final presentation at Osaka University is given to the instructing party
- Week 20: An final presentation of the project is given at TU Delft

To enhance the working process:

- All data is saved on a Google Code SVN server
- All data is back upped locally at least once a week
- A weekly progress meeting with the instructing party is held
- Mr. Sodoyer is updated every three weeks about our progress

1.3 Technologies used

The following programming languages and technologies will be used throughout the project:

- Java
- Java Persistence
- MySQL
- Apache Tomcat
- Soap
- Apache Axis
- JavaScript
- JSP
- PHP
- AJAX
- Reverse-AJAX
- XML
- CSS

2 Analysis

This chapter describes the current situation (section 2.1) and the capabilities our system should have. A general overview of the system is given in section 2.2. Section 2.3 will explain what types of input there are, while section 2.4 describes the different types of uploading. Afterwards, the website and its features are discussed (2.5) and this chapter is concluded with the description of sample applications (2.6).

2.1 Current situation

Before we can determine what the system *should* be able to do, it is necessary to analyze and describe the current situation. Most notably the questions "What needs to be automated?" and "How is the situation handled now?" need to be answered. By doing this the client's goals for the assignment will be better identified. For us as developers this information is invaluable as it will enable us to more precisely gather requirements for the system and eventually develop a system most catered to our client's needs and wishes.

It quickly became clear that our system would provide entirely new functionality – that is functionality that was not available to our client before. In the current situation, there is no system which provides the functionality our client is looking for. Hence, the actions our client wants to be able to perform using our system, namely automatically inputting location data and sharing this with others, are not being performed in the current situation. The capability of receiving and storing GPS data exists, however there is not sufficient functionality to share this data with others. Existing free and open applications such as whrrl.com and pownce.com (shut down during the writing of this document), while providing the option to share your location with others, don't provide an automated and user-friendly interface to do this. They rely on the user to input the location data himself. The ability to input this data automatically and share it with others is the functionality we look to develop.

In the current situation the location data exists in different forms. A GPS device developed by Sony receives and stores location data in a format called NMEA 0183. Furthermore, our client himself has implemented an application for Apple iPhone called iSpotter which stores location data derived from wireless internet hotspots in XML format. Our system will need to be able to read these formats and convert them into meaningful location data to share with others.

2.2 Overview

Figure 2.1 shows a simple overview of the Sococa system. On the left side of the figure are all the inputs. The inputs can be divided into two parts:

- GPS device. This device collects GPS data from satellites. This data includes location and time. The user can connect this device to his pc, to upload this GPS information. The uploading is done by the special Sococa client application. With this application it's possible to upload this GPS information in real time. In other words: When new GPS data is received from a satellite, this data can be directly sent to the server. The location data is sent in the XML format.

- Other applications. The sococa system can also be accessed by other applications than the Sococa client application. This can be applications made for cell phones/pda's or other devices. The applications can connect to the server and send their location information in the XML format. Authentication is done by basic authentication.

The inputs are reviewed in more detail in section 2.3.

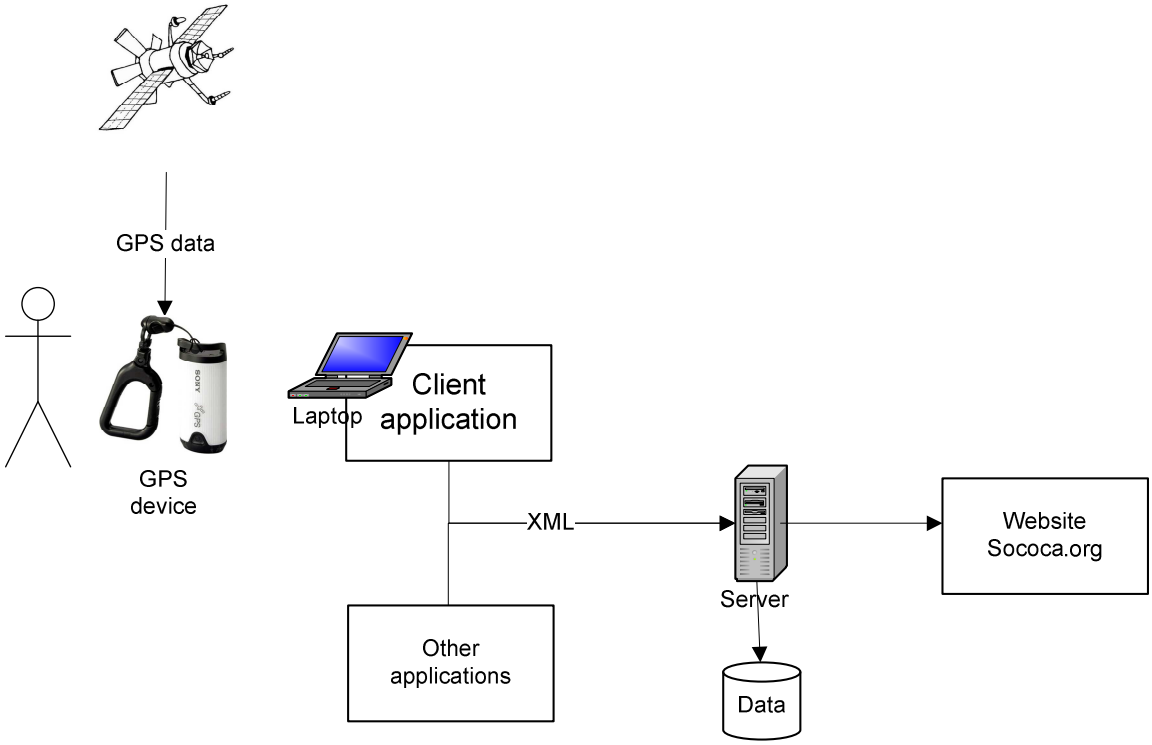


Figure 2.1

As shown in figure 2.1 the XML data arrives at the server. The server then has two jobs:

- Save the XML data to the database.
- Update the website. The Sococa website has a map with all the users in a given area. When new location data is uploaded to the server this map is updated accordingly. More information about the website in section 2.5.

2.3 Inputs

Because there are several ways of collecting GPS data, the Sococa system should be able to accept as many different types of inputs as possible.

GPS logging device

The Sony GPS logging device collects location data using GPS. The location data is saved in log files. Every day a new log file is created. Within the log file location data is saved in the NMEA 0183 format (Hirschelmann, 2006). See figure 2.2.

```
@Sonygps/ver1.0/wgs-84
$GPGGA,042132,3449.4385,N,13531.4702,E,1,03,03.9,-00000.0,M,033.8,M,,*6C
$GPGSA,A,2,06,16,23,,,,,,,,,,04.0,03.9,01.0*0C
$GPGSV,2,1,07,06,51,205,48,16,63,323,50,21,22,099,29,23,30,305,49*70
$GPGSV,2,2,07,24,11,057,00,03,,,32,31,,,36,,,,*4F
$GPRMC,042132,A,3449.4385,N,13531.4702,E,000.0,298.9,040908,,,A*7D
$GPVTG,298.9,T,,M,000.0,N,000.0,K,A*07
$GPGGA,042152,3449.4429,N,13531.4563,E,1,03,03.9,-00000.0,M,033.8,M,,*6E
```

Figure 2.2

Within this format there are several message types. These messages provide all kinds of information, such as speed, position, time, accuracy, etc. Only two messages mention a longitude and a latitude, so these can be used to determine a user's position. These message types are GGA and RMC.

XML
XML is used by information systems to share structured data over the internet. This makes it suitable for using it as an input format. So by accepting XML as input, our system can read input from all kinds of programs, as long as those programs use the same DTD (document type definition). An example of XML is shown in figure 2.3 below.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <locations>
        <location>
            <longitude> 34.494385</longitude>
            <latitude> 135.314702</latitude>
        </location>
        <location>
            <longitude> 34.494429</longitude>
            <latitude> 135.314563</latitude>
        </location>
    </locations>
```

Figure 1.3

iSpotter
One of the programs from which Sococa should accept input is iSpotter. This program was developed by Mr. Teranishi and collects location data on the iPhone/iTouch using a WiFi-based location detection technology. This location data is saved in XML, so our system should accept this input (see previous section: XML).

2.4 Uploading of location data

There are two ways of uploading new location data to our system. This can be done in real-time, or when no internet connection is available at collection time, by uploading files with location data later.

Real time uploading
An important feature of the Sococa system is its ability to upload and save data in real time. This allows other people to follow the users' movement in real time. In case of the GPS logging device, a new file for saving location data is created each day. The system should automatically select the latest file from which to read from. Whenever a location entry is added to the file, the system should detect this. When a new location entry is detected, it should be parsed to a Java object and sent to the server, which will save it in the database.

Uploading of a single file
It is possible users do not have an internet connection available when they are logging their location data. In this case it is not possible to upload data in real time to the server, so this has to be done at a later time. The device the user is logging with will have saved a file with all locations of that day. Users should be able to upload this file when they have an internet connection available.

2.5 Website

All location data stored in the database should be made visible on a website. User can check routes they took in past, see the latest positions of their friends and other people and can manage their friendships among Sococa-users.

Route tracking
Users should be able to view their movements in a given time interval. The route they took in the interval should be animated on a map. Also, the movements of the user's friends are shown on the map. This visualization should look like the interface of the Real Time Rome project, developed by MIT (MIT, 2006). RTR shows their user's movements as a line. In our system this should also be a line, but one which is animated. Furthermore, a user avatar is added to the line, so it is clear which line belongs to which user.

Latest position of people
Users of Sococa can use the real time upload function. When they use this functionality their location data is automatically uploaded and saved in the database. On the website the map can show (a selection of) users with their latest position. The possible selections are: self, friends and everyone. When a user changes his position, this is directly updated on the map. So when the server receives new location data for a user, it should send an update to the map. The map is then updated with the user's latest position.

Friend interaction
Users are able to register other users as their friends. When two users are friends it is easier to keep track of one another, as one can set filters on the map to only view friends.

When a user wants to add another person as a friend, he can search for that person using the search bar. All Sococa users matching the search query are shown and one can select the right user by viewing the profiles. When the users finds the person he was looking for, he can add him as a friend.

Because of privacy reasons users can set their visibility. They can choose to make themselves only visible to friends or to nobody. Of course, it is also possible to make yourself visible to everybody.

Finally, users are able to send text messages to each other.

2.6 Sample applications
The Sococa system should be extensible for other applications. These so called sample applications are mash ups with other well-known applications.

Zimbra is an email client. Just like for other programs such as web browsers it is possible to develop add-ons for extra functionality. In Zimbra's case, these add-ons are called zimlets. Sococa will provide a zimlet as well. This zimlet will show a little map on an email message. The latest position of the sender is shown on this map.

Other sample applications are for example:

- Blogging service: Uses location data to automatically generate blog entries.
- Photo service: Users can add photos of places they visited to their locations.
- Sight recommendation: Using the users' location data, the system gives users recommendations of places to visit. For example, because friends also visited this location.

3 Design

This chapter is about the design of the Sococa system. Firstly a global overview of the system is given(3.1). After that, the client application (3.2) is explained, followed by the design of the server (3.3) and the Sococa map design (3.4). The chapter is concluded by the API design (3.5).

3.1 System overview

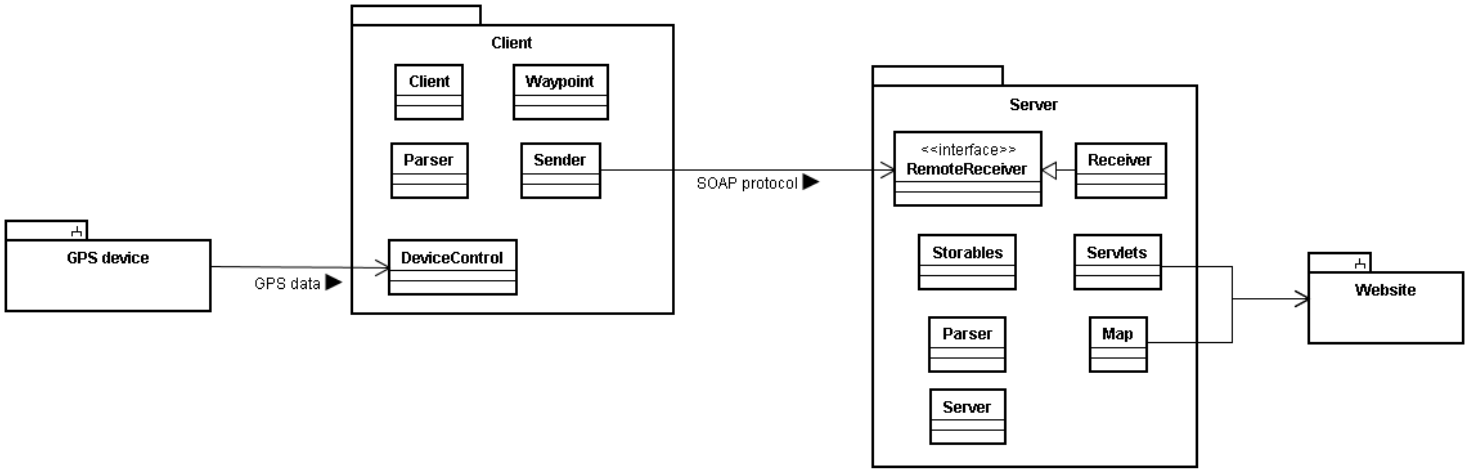


Figure 3.1

A high level design of the Sococa system is shown in figure 3.1. We have separated our system into two parts: The client and the server. The client is a stand-alone program which runs on the pc of a user. When the GPS device is connected to the computer, the user's location data can be uploaded to the server. To achieve this, the client has to do some efforts. The main class of the client is also called client. This class controls all other classes in the package client. In short, GPS data is received by the DeviceControl class, the Parser then parses this data to waypoints. The waypoints can then be sent to the Server. The Sococa system uses SOAP to send the waypoints to the server. At the server, the Receiver class handles the receiving of waypoints. Receiver is an implementation of the interface RemoteReceiver. At the server the waypoints are transformed to storables. The Servlets and the map display the storables graphically on the website. More details about the various parts of the Sococa system are explained in the next sections.

3.2 Client

The client is a stand-alone application on the user's computer or logging device. It is responsible for reading the log files from the GPS logging device. When new locations are read, they are directly sent to the server.

3.2.1 Observer pattern

The client is built using the Observer pattern (Lethbridge, 2005). When a device is connected to the system, the Client will start a DeviceControl. This class is basically a (observable) thread which polls the GPS logging device every few seconds. When new lines are read, an update is sent to the observer Client (the main class). Client then makes sure the lines are parsed to waypoints (see next section) by calling the LocationParser. It also sends information to the GUI to inform the user a new waypoint has been read.

3.2.2 Waypoint

Waypoint is the storage class of the client. It contains all location data which is needed by the website to view user's movements. See figure 3.2.

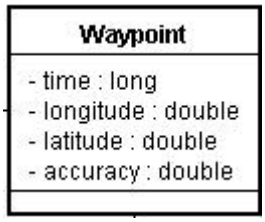


Figure 3.2

The time is measured in milliseconds since January 1st 1970. The longitude and latitude are GPS coordinates to define a position on the globe. The accuracy indicates the quality of the GPS signal, the larger the number, the larger the error. The error in meters can be calculated by multiplying the accuracy by 6. The accuracy is usually called the Dilution of precision (Dana, 2000).

3.2.3 Sending data using SOAP

When new waypoints are created, they should be sent to the server. Using apache Axis it is possible to get an interface of the server. On our server the interface is called RemoteReceiver (figure 3.3). Using this interface it is possible to access the server as if it were a local object (RPC). The interface consists of two methods: uploadWaypoint and authenticate. The first sends an array of Waypoints to the server. The latter sends a username and password to the server for logging in through the client.

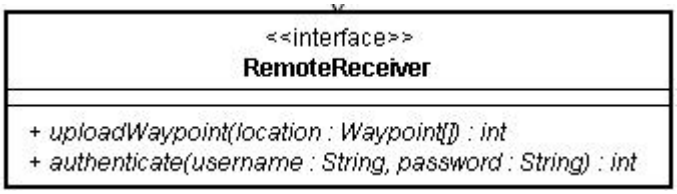


Figure 3.3

Although it looks like methods are called and objects are passed directly, this is of course not the case. In reality Apache Axis converts the java objects to SOAP messages. A SOAP message is a simple XML description for the Java objects.

Below, in figure 3.4, is an example of what a Waypoint looks like in XML.

```
<schema elementFormDefault="qualified"targetNamespace="http://client">
     <complexType name="Waypoint">
        <sequence>
            <element name="accuracy" type="xsd:double"/>
            <element name="latitude" type="xsd:double"/>
            <element name="longitude" type="xsd:double"/>
            <element name="time" type="xsd:long"/>
        </sequence>
     </complexType>
</schema>
```

Figure 3.4

SOAP messages are sent over the HTTP protocol. To identify the user, BASIC authentication is used. In essence, this means that a username and password are added to the header of the HTTP message.

When the username and password are present in the database, the waypoints are added to the database, otherwise an error message is returned. When HTTPS is implemented, BASIC authentication can be used to ensure safety.

3.3 Server

The server is responsible for all data management, because it controls the database. Aside from that, it handles the incoming waypoints from the client and updates the information on the website.

3.3.1 Java Persistence API

The database and server are implemented using the Java Persistence API (from now on called JPA). It is used to manage relational data in Java. This has a big influence on the server design. Tables from the database are implemented as Java classes. This means columns are attributes of a class. Using annotations one can refer to the correct database column, if the name is different from the attribute name used in the Java class. In the example in figure 3.5 below the get-method for the ID of a User is shown. In the database a User table is saved with a user_id to identify it. The corresponding Java class should also implement a user_id. Because of our naming convention in Java it's called userId. In order to keep referring the right column an annotation is used. The annotation @column tells JPA this attribute is also present in the database, the name parameter tells it is named user_id.

```
@Column(name="user_id")
public int getUserId() {
      return userId;
}
```

Figure 3.5

The big advantage of using JPA is that complex SQL queries are avoided. Suppose a new LocationElement needs to be added to the database. In Java and SQL this would look like something like figure 3.6.

```
LocationElement loc = new LocationElement();

//make database connection
//execute a query like:
Insert into LocationElement (longitude, latitude, accuracy, user_id) values
(loc.getLongitude(), loc.getLatitude(), loc.getAccuracy(),loc.getUserId());
//close database connection
```

Figure 3.6

In JPA this is a lot easier, because the class LocationElement is already linked to the table LocationElement. See figure 3.7.

```
LocationElement loc = new LocationElement();
//make database connection
loc.persist();
//close database connection
```

Figure 3.7

In both cases some kind of database connection is required. In JPA this is called a transaction. Before modifying something in the database, one is supposed to open a transaction and modify some objects. When all objects are ready to be saved the transaction is committed.

An update command shows the same difference. Suppose we want to update the longitude of a locationelement. In traditional SQL this would look like this:

```
update locationelement set longitude = 23.458 where user_id = 2;
```

However in JPA it's just a matter of calling the set-method in a transaction context.

```
loc.setLongitude(23.458);
```

3.3.2 JSP and Java Servlet technology

JSP (Java server pages) is used to create dynamic web content (Sun MicroSystems, 2008). Among other things, it allows one to write Java code inside HTML pages. There are two ways in which java can be embedded in HTML:

- By using the special HTML tag library
- By inserting <%> tag, which allows one to write Java code within

We have used the <%> tag, because that proved to be the easiest way to work with. Using this method we could simply code in Java, which we were already familiar with. This saved us the time we would have spent learning the new tags.

A Java Servlet is an object extending a web server (Sun MicroSystems, 2008). It can be used to build web based applications. It is basically the same as PHP or CGI, except that it is written in Java. Servlets have access to all Java API's including JDBC, which means it is possible to make database access.

Combining the strength of JSP and Servlets allowed us to handle sessions, users and locations as Java objects.

3.3.3 Singleton pattern

Server is the main class of our application. In order to ensure there is always only one instance of this class present, we have used the singleton pattern (figure 3.8).

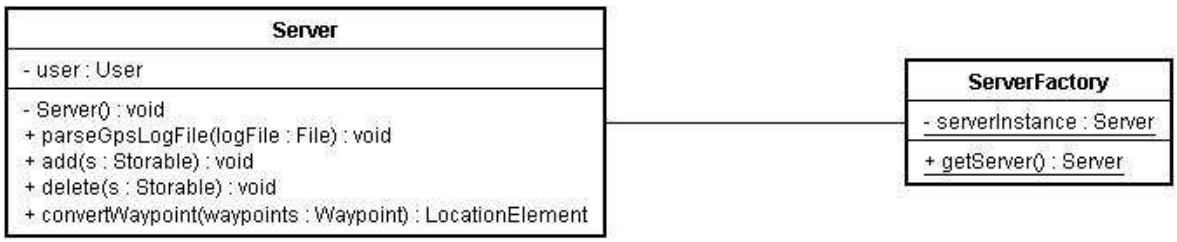


Figure 3.8

The ServerFactory controls the instance of Server. If a class requires an instance of Server, it should call the static getServer() method of ServerFactory. This will return the server instance or, in case it is not initialized yet, a new instance of Server.

3.3.4 The Storable class

As mentioned in section 3.2.1, the Java Persistence API was used to implement the database tables as java classes. To avoid duplicate code and to keep our system extensible, the abstract class Storable was created. This class consists of two implemented methods: add and delete, and one abstract method: getId. All classes which represent database tables extend this class. For example, the add method is the same for each class, consisting of a simple persist call to an object. Implementing this same method in every class would be redundant.

The Storable class also enhances the extensibility of our system. When a new table is added to the database, one does not have to worry about database access, as that is taken care of in the Storable class.

User
The most important class that extends Storable is User. This is because User has a collection of all other Storable classes.

UserDetails
We have decided to make a separate table with user details, to reduce the size of the table user in the database. Reducing the size of this table should result in a (minor) speed increase. In this table, information like address, favorite tracking color and so on are stored. In short, information which is required for some purposes, but not used very often. Because userdetails should be part of user, there is a one-to-one relationship between them.

LocationElement
LocationElement is basically a Waypoint (3.1.2) with a user_id added. It represents one spot where the user has been. All these spots can be connected by lines and form routes the user has taken.

BlogEntry
Each blog entry of the user is saved in the BlogEntry class. This is a very basic class that just saves text written by the user. It should also be possible to generate blog entries using the user's LocationElements.

Friend
The friend class represents a friendship between two users. This class basically holds two users and the status of their friendship. The status can vary between pending and confirmed. This means that after user A made a friend request to user B, user B should confirm that they are friends.

PrivateMessage
This class saves a message sent from one user to another. The User has a collection of this class, also known as his inbox.

3.3.5 Parsing

Just like the client, the server is capable of parsing files. The server can, besides the log files, also parse XML files. Because these parsers have so much in common, an abstract super class called parser is introduced. The server now does not have to know which type of parser it is using, a simple 'parse' call to parser will do. The decision whether to use a GPS or an XML parser is taken by looking at the file extension of the uploaded file. See figure 3.9.

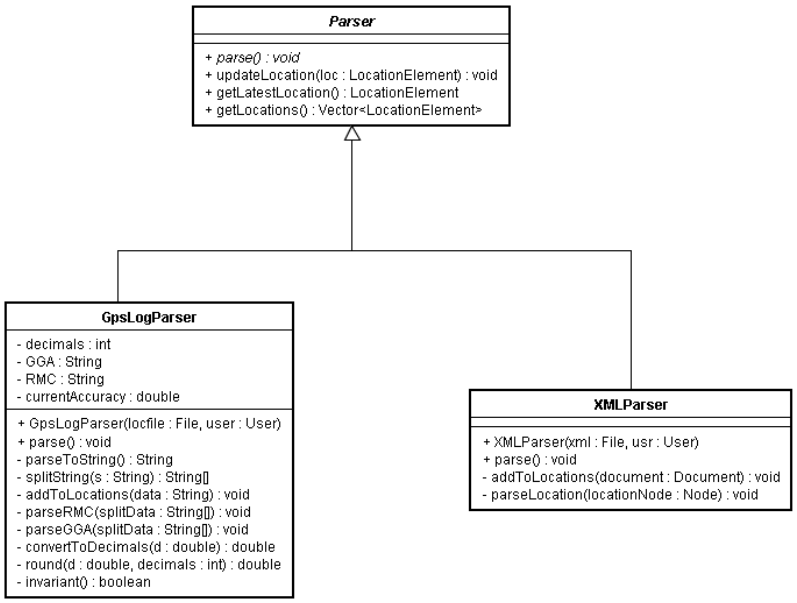


Figure 3.9

3.4 Sococa map design

The browser-side location processing of the Sococa system relies heavily on the Direct Web Remoting (DWR) library for Java. This library simplifies making AJAX (Asynchronous JavaScript and XML) and/or reverse-AJAX calls by allowing users to call server-side Java functions from the browser (in JavaScript), and dynamically generating JavaScript on the browser (reverse-AJAX). The main functionality of the map consists of the procedures described below.

3.4.1 Initialize map

Whenever the user loads the location page, the map is loaded, and all public users are displayed on the map, as shown in figure 3.10:

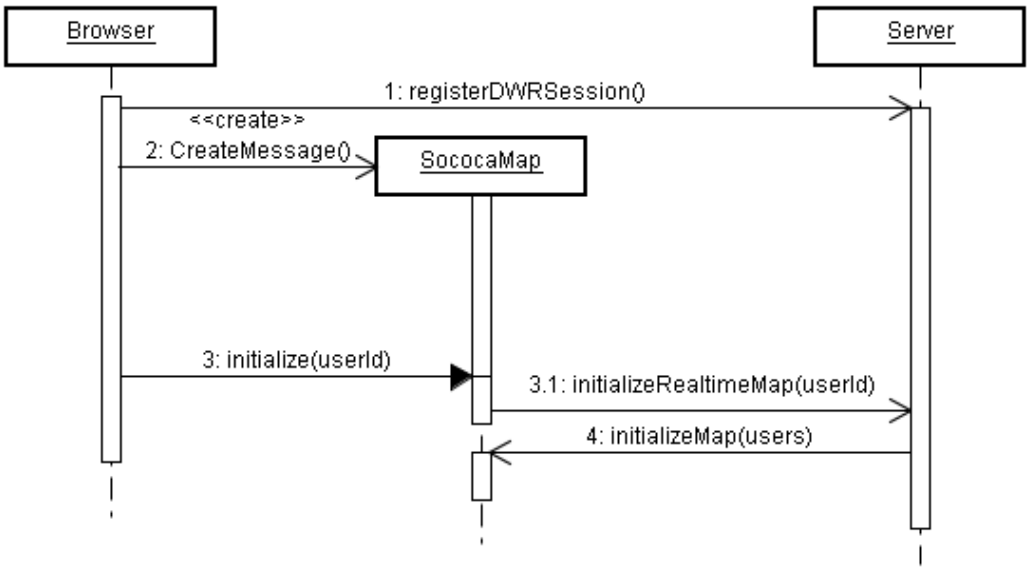


Figure 3.10

First, whenever the page loads, the JavaScript-object SococaMap is created. This object, in turn, creates a Google Maps GMap object, initializes some variables, and makes a DWR call to the server to register the current user to DWR's WebContext object, which keeps track of all currently active browser sessions. This map is then initialized by calling the initialize method, passing it a userId as argument (the userId is obtained from the server through JSP). The SococaMap then makes an asynchronous DWR call to the server, which queries the database for all users visible to the calling user (public users and friends). The server makes a callback to the browser with an array of User-objects, which DWR automatically converts to JSON (JavaScript Object Notation) objects. This way, it is possible to extract the necessary location data on the browser and add the users' latest known locations to the map.

3.4.2 Receive new waypoint

Whenever a new waypoint is uploaded to the system, either through the client, by uploading a log file to the portal, or by some other device uploading waypoints to the SOAP web service, it will be determined whether or not this waypoint is the latest location of the user by checking the timestamp. If it is the latest, this latest waypoint will be sent to the browser and displayed on the map, as shown in figure 3.11.

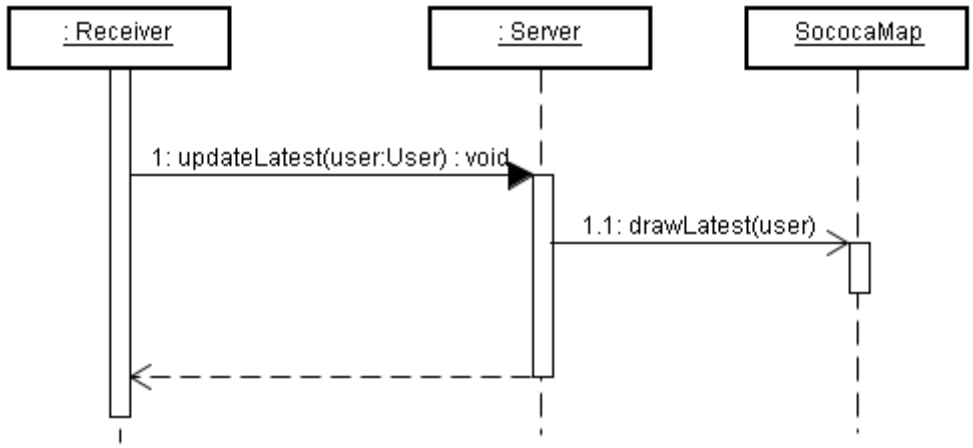


Figure 3.11

The user object sent to all browsers currently on the map page contains the latest position of this user. The map will then be updated for all sessions by moving the user's avatar to the new location.

3.4.3 Routes

A user can also see the route he has taken in a given period animated on the map. This is done through a calendar JavaScript library, which will display a calendar on a button press and insert the subsequently selected value into an input field. When the start tracking button is pressed, these values will be read and all waypoints within this interval will be fetched from the server, as in figure 3.12.

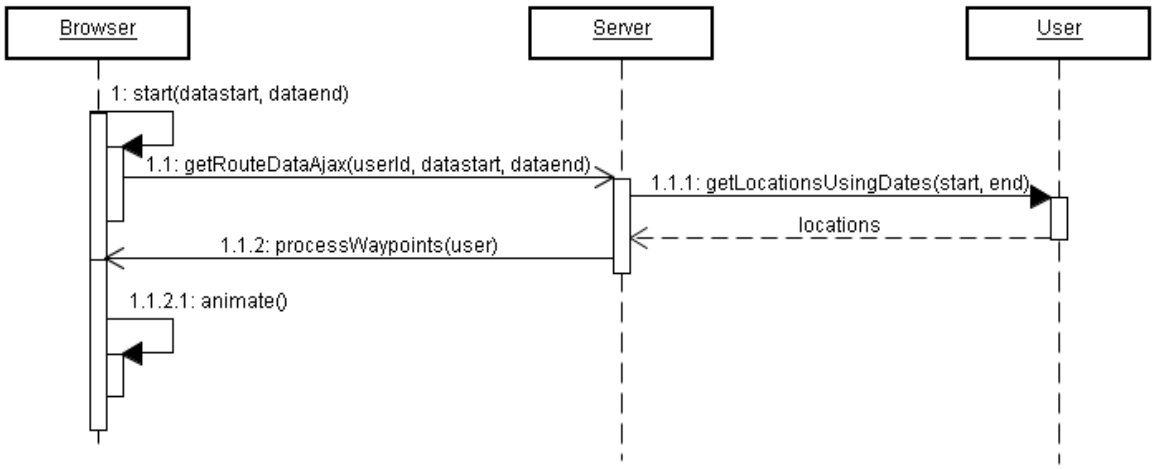


Figure 3.12

First, the start function is called with the inputted start and end dates. This function will make a regular AJAX call to the Server, passing it the dates and the user's ID. The server will fetch all waypoints within the specified period from the database, set them to a user object, and return this user, upon which the JavaScript callback function processWaypoints will be called. In this function, the waypoints are extracted from the user object and the friend objects within, converted to Google Maps waypoint objects, and all needed variables inserted into Entity objects. The animate function is then called, which will animate the routes of the user and his friends on the map. This is done by calculating the points in a certain step interval by linearly interpolating between two waypoints of the user, and sequentially moving the entity's marker along these points, adding overlays to the map to leave a trail. It is also possible to increase or decrease the animation speed through buttons displayed on the map that manipulate the step variable, which determines the amount of distance to travel between two drawn points. It is possible to pause tracking and resume it again, or to stop tracking and return to the regular real-time location display.

Finally, it is possible to delete waypoints from the database. This is done by selecting a period using the calendar, and clicking the 'Delete locations' button. This will make an AJAX call to the server, which will delete all LocationElements within the selected period from the database. Through a callback function, the user will be informed whether or not the operation has succeeded, as in figure 3.13:

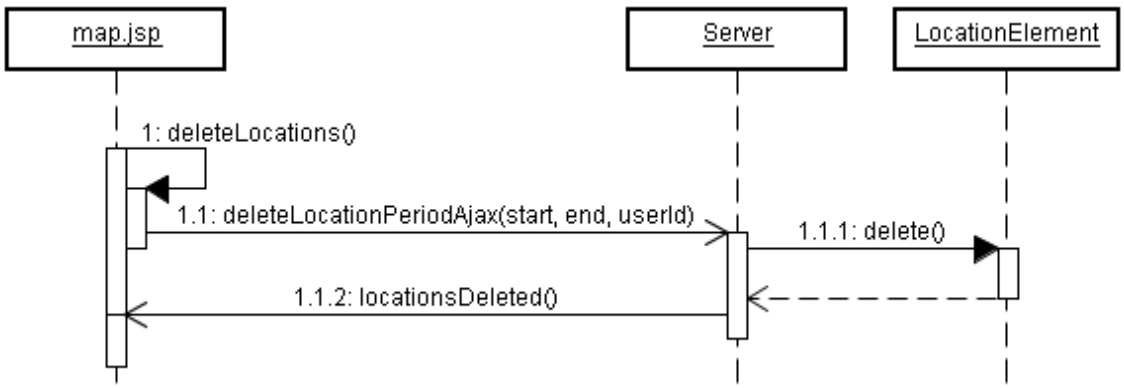


Figure 3.13

3.5 API Design

The SococaMap Javascript object is built as a JavaScript library available for public use. First, a user has to be registered to the system, and generate a personal API key on his profile page (not yet implemented). It is then possible to call the MapService servlet, providing the API key as GET object in the URL, as follows:

http://www.sococa.org/socialocalize/MapService?file=api&key=ABCDEFGHIJKLMNOPQRSTUVWXYZ012345

The API key is a string of 32 randomly generated characters, each linked to a specific user. If the key is correct, the JavaScript API is returned to the user, who can include it into his own web page. After including the API, it is possible to create a new SococaMap object, assign it to a element on the page, and call the available functions. Examples of functions are:

- Initializing the map, showing all public users
- Adding control objects to the map (Route tracking control buttons, animation speed controls)
- Low-level functionality:
    - Retrieving all users in a specified area
    - Retrieving all users within a specified period
    - Retrieving all users in a specified area, within a specified period

For safety reasons, it is not possible to delete waypoints through the API. If a user wants to erase his past routes, he should log in to the portal site and delete them from there.

Due to time limitations, API authentication is currently not yet fully implemented. Also, no real security measures have been implemented so far.

4 Implementation

4.1 Website implementation

The website is an important part of our system. It provides the Social Network Service functionality to users. Here, users can manage their friendships, chat, send messages and so forth. In this chapter we will describe the implementation process of the website.

4.1.1 JSP/Servlet communication

The website was implemented in JSP (JavaServer Pages). JSP is basically HTML but with support for Java. Java code can be directly executed using the tags: `<% java code %>`. Since the underlying system is build in Java, it would make sense to use a web language which can handle Java, so the choice for using JSP was easily made.

Our JSP web pages have not been included in the design, since this part of the code does not add any objects or classes to the system. The JSP pages only use classes already exist in the design. However, the Java servlets were included. Because the JSP pages are not in the design, it might seem difficult to understand how the website handles its communication with the server. But actually it is quite simple. JSP pages can communicate with servlets through a pre-defined variable "request" which is of the type javax.servlet.http.HttpServletRequest. A JSP page can call this variable to request data from a servlet, which in turn accesses the server or database. How this is done is explained in 4.1.2.

The HttpServletRequest class uses HTTP POST communication. POST is a way of communicating data through different web pages by HTTP. Another way to do this is GET. GET http communication method is used by some servlets. The difference between the two is that GET should be only used to retrieve information from the server, not change its state. POST, on the other hand, can change the server state. For more information on GET and POST, see the HTTP/1.1 specification (World Wide Web Consortium, 2008).

To show an example, if we wanted to retrieve a user's details from the database for display in a JSP page, the code to do this would look something like below.

Firstly, the id for the user we want the details from is in the URL, so it can be retrieved from GET. Knowing this, we can now call the servlet inside the JSP page:

```
      RequestDispatcher rd =
application.getRequestDispatcher("/UserServlet");
      rd.include(request, response);
```

Then the servlet could retrieve the id of the user we're looking for from the URL using request.getParameter() to retrieve it from GET. After that, the appropriate user can be looked up in the database, like so:

```
      int searchingUser = Integer.parseInt(request.getParameter("id"));
      User searchedUser = em.find(User.class, searchingUser);
      request.setAttribute("searchedUser", searchedUser);
```

Now, the JSP page can in turn retrieve the user from request:

```
      User u = (User) request.getAttribute("searchedUser");
```

Eventually the JSP page has full access to the User object, and can call its functions to retrieve details and such.

Most website-server communication is handled like this, however sometimes it makes more sense to call a servlet from a form and use POST communication. We will give an example of such a method of communication below.

This example describes the data flow of a user that updates his details through the website using the editprofile.jsp page. First, the user fills in his details in a form. When he clicks submit, the data is gathered and sent to ProfileEditServlet, like so:

```
<form action="ProfileEditServlet" method="post">
<input type="text" name="userdetails" value="<%=userdetails%>">
<input type="submit" value="Submit">
</form>
```

(NOTE: The input field for userdetails is displayed in simplified form above, in reality userdetails consists of many different attributes which all need to be input separately.)
Then, the servlet attempts to retrieve the data from POST:

```
UserDetails newud = request.getParameter("userdetails");
```

Thirdly, the data is updated in the database:

```
User cu = (User)request.getSession().getAttribute("currentuser");
UserDetails ud = cu.getUserDetails();
ud = newud;
em.getTransaction().begin();
em.persist(cu);
em.getTransaction().commit();
```

And the user is redirected:

```
response.sendRedirect("profile.jsp");
```

So that he can see the updated profile.

4.1.2 Servlet/database communication

In the examples above, the servlet in both cases calls an `em.` method. This is the way a servlet accesses the database through persistence. The type of `em` is javax.persistence.EntityManager. Servlets call this class to access the database, for retrieving data as well as adding or updating data. The data stored in the database are Java classes, stored as Persistence entities (also see 3.2).

Retrieving is done with `em.find(Class<T> entityClass, Object primaryKey)` which returns an object of type T. Since the method searches by primary key, it can only find 1 (or 0, in which case null is returned) results.

Data in the database is updated or added through `em.persist(cu)`. For this method to work, the EntityManager first needs to initiate a transaction (`em.getTransaction().begin()`). A transaction in Persistence is defined as “a set of operations that fail or succeed as a unit” (Wikibooks, 2008). Finally the transaction is committed and the data in the database is updated when the transaction succeeded or rolled back in case of failure (`em.getTransaction().commit()`).

4.1.3 OpenID authentication

For users to authenticate to the website it was required to use OpenID authentication. This is a freely available open source technology which aims to provide central authentication for the every website that supports OpenID. The user logs in on what is called an OpenID provider, which in turn can authenticate the user on different sites called relying parties. Hence, logging in with a username and password is only required once.

For using OpenID in Java, there are a few libraries available. We chose to use the library "joid" (Java OpenID). This library provides support for both providing and relying parties. Our website will only be a relying party, not a provider. So in order to authenticate to our website, a user first needs to log in to an existing OpenID provider.

The process of logging in with OpenID using joid in a JSP page is quite simple. Basically the entire authentication is done in one line of code:

```
String s = OpenIdFilter.joid().getAuthUrl(id, returnTo, trustRoot);
```

The class OpenIdFilter is located in *`org.verisign.joid.consumer.OpenIdFilter`*. Here *`id`* is the URL of the OpenID provider the user wants to authenticate from. The parameters `returnTo` and `trustRoot` are the URL you want the user to return to and the URL you want the provider to authenticate the user at, respectively (these can be the same). This method returns a String which represents a URL that the user must visit in order to authenticate. So it makes sense to redirect the user here:

```
response.sendRedirect(s);
```

The user is now authenticated to the `trustRoot` page. Of course this doesn't show how the authentication process actually works under the bonnet; we merely call a predefined function from a library. For more information on how the authentication works we refer to the OpenID 2.0 specification (OpenId Foundation, 2008).

4.1.4 CSS & website design

The website design was a less important part of the website implementation. We are software engineers, not website designers, and thus we decided to keep the design simple but effective. For this purpose, we used only CSS and no DHTML or Flash. For certain functionality we used JavaScript as well. CSS is the most widely used technology for website design on the web, and hence finding documentation was easy.

The CSS code was split between 2 files: scc_style_outline.css and scc_style_forms.css. The scc in these files is short for Sococa. The first file contains code for , <table> and such tags. The second file contains code for used in forms, such as <input> tags. The amount of design code for these tags was getting large. To keep the CSS file organized we decided to split it up in two files.

4.1.5 Email confirmation

Before users can use the full functionality of our site we require that they enter a valid email address. To confirm this email address we use a confirmation system. When a user first accesses our website he is asked to enter an email address. Then, a servlet is called which generates a unique key and stores it in the database for the appropriate user. An email is sent to the specified email address by an email server on the same machine which the website runs on. The email contains a URL with the secret key, which when the user clicks it confirms him and allows him to access the rest of the website.

The secret key is generated by the following lines of code:

```
Random r = new Random();
String token = Long.toString(Math.abs(r.nextLong()), 36);
```

The security of this key is perhaps questionable, but since high security was not a priority for the assignment this implementation will suffice for the time being.

4.2 Testing

A lot of the Java classes are tested using JUnit (see figure 4.1). JUnit provides a testing environment for automated testing. It is easy to see if the system is still consistent after you have changed something in the code. This can be easily done by running the testAll test, which runs all tests of the system.

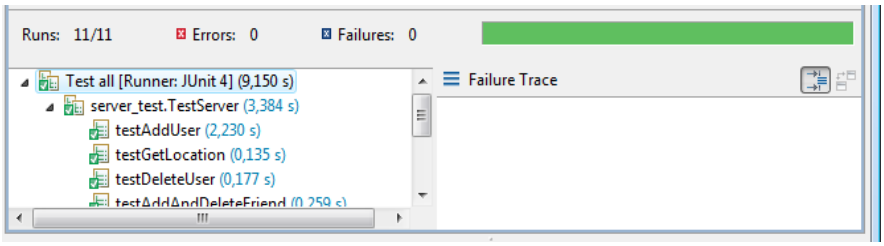


Figure 4.1

4.3 Design by contract

We have tried to use the design by contract (Meyer, 1992) methodology as much as possible. All Java classes have asserts and invariants to ensure the code is correct. Using this methodology in JSP/Javascript and other non-Java files was much harder or impossible, so there are few asserts and invariants in these files.

A good example of design by contract can be found in our Waypoint class. To ensure waypoints always have correct values for their longitude and latitude attribute, this is checked in the invariant (see figure 4.2). The invariant is asserted in every other method in Waypoint, except in the simple get-methods.

```
/**
 * longitude should always be between -180 and 180
 * latitude should always be between -90 and 90
 */
private boolean invariant(){
      return
      this.time >= 0 && this.time <= System.currentTimeMillis() &&
      Double.compare(longitude, -180)     >= 0 &&
      Double.compare(longitude, 180)      <= 0 &&
      Double.compare(latitude, -90)       >= 0 &&
      Double.compare(latitude, 90)        <= 0;
}
```

Figure 4.2

5 Conclusions

The goal of this BSc project was:

to develop a tool that can keep track of people's movement, share a visualization of these movements with other people and allow for other means of social interaction, such as chatting, messaging, finding other people and so on.

Our system is capable of the following to achieve this goal:

- Parsing and uploading of the user's location data in real-time
- Updating the user's position the map on the map in real-time
- Animate routes the user took in the past
- User can add other users as their friend on the website
- Users can chat with each other (global) or send private messages
- Users can set their privacy settings

There are some functions that were not or partially implemented:

- Full OpenId support. It's possible to log in to the website using your OpenId. The client however does not support this. A separate client password can be set on the website.
- OpenSocial API support. This proved to be more difficult than expected. Because we didn't have enough time to research how to implement it, we've chosen not to implement this feature.

Although we didn't have enough time to implement all initially required functionality, overall, we believe that we have provided a capable product that covers the essential functionality for a useful system. There were a number of reasons why we couldn't implement the minor features anymore. For one, the initial scope of the assignment may have been too large. Another reason is that we initially were inexperienced with using the various techniques that were necessary to implement the system.

During requirements gathering, we were slightly too optimistic about the amount of features we thought we could implement, so one or two 'must haves' did not make it into the delivered product. However, considering the size of the list of features, we believe we have made an accurate estimate overall of what we were capable of providing in the time given.

In the design phase, a great deal of time was spent researching the techniques necessary to effectively implement the system given the performance and scalability requirements. This was mostly due to us not having a clear view of the kind of system we were supposed to make beforehand, making us unable to do truly relevant preliminary research, as we were inexperienced with using said techniques. In the end, we were satisfied with our design, and ended up following it to a great extent.

Since we were working on the project by ourselves, and no other students were involved in its development, we were left with a lot of liberties and responsibility on how to design and implement the system. Our instructor, Mr. Teranishi, was of great help to us whenever something was unclear, and gave us a lot of freedom in how to shape the system into its final form. We ran into a number of problems during implementation, which will be explained in detail in the next section. Some of these led to minor design changes, as some things didn't work as effectively as we initially thought. However, in the end, we believe we have delivered a system that meets the necessary requirements, and is solid in its architectural design and program code.

6 Experiences

This chapter describes our experiences during our project. Some things went well, some things not. OpenId is discussed first (6.1), followed by the Java Persistence API (6.2) and Apache Axis (6.3).

6.1 OpenId

We have tried to implement OpenId for both logging in to the website as to logging in via the client. Logging in via a website was easy to implement, there are several libraries available on the internet which help you to support OpenId. The library we used was based on Java, so it was easy to integrate in our system. Logging in via the client proved to be much harder to implement. Because the client doesn't have a browser, it's much more difficult to redirect to the right provider. Besides that, there were no libraries available on the internet, that support OpenId on client programs. Because OpenId was basically impossible to implement for the client, we have chosen to use a separate client password.

6.2 Java Persistence API

The Java Persistence API is a way to easily use relational data in Java (as discussed in 3.2.1). The big advantage of JPA is that queries can be avoided. Instead of those easy methods can be called. Although this sounds alluring, configuring JPA was difficult. Also, the design had to be adjusted to facilitate JPA, which sometimes lead to some uncertainties. It was a nice experience to use JPA in a project, but we probably won't use it again. All the time we saved by not having to use queries, was spent by properly configuring JPA. Furthermore, the complexity of the classes was increased. So all in all JPA wasn't as easy and timesaving as we had hoped for.

6.3 Apache Axis

The uploading of the location data to the server proved to be easier than expected. This was because of Apache Axis. This technology allows developers to access remote object as if they are local ones. So calling a method of a remote object becomes just as easy as calling one from a local object.

Because our IDE Eclipse had excellent support for this technology implementing it was easy. Eclipse has some functions which create a sample network service which is easy to adjust to your own needs. Also the web services explorer allows you to view the XML which is generated by the client and it provides some handy testing features.

7 Recommendations

There is still room for extensions and improvements. Because of a lack of time and the complexity of the problems we were not able to implement/solve them. The most important improvements are listed below:

OpenSocial extension

Many well known social networking sites such as Orkut and MySpace have implemented the OpenSocial interface. This allows programmers to develop programs with easy access to information on the social networking site. Using this API it should be possible for users to import their friends from other social networking sites into the Sococa system. OpenSocial is currently not implemented in the Sococa system.

Blogging

One of the sample applications for the Sococa system is blogging. Because of the limited amount of time we have chosen not to implement this application. However, the class is largely implemented. So it should be easy to add this sample application to the system. The most work would probably be the implementing on the website.

Authentication API

The Sococa system offers an API to its users. With this API users can easily embed the Sococa map to their own website. Currently there is no authentication required to view this map. Because of this, only people who have their privacy setting set to “public” are shown on the map. When authentication is implemented, it should also be possible to see friends of the user.

Add time dimension to route tracking

Route tracking shows the route the user has taken in a given period. Although LocationElements have a time and date, this is not taken into account while showing the route. Because of this, it is not possible to see how much time somebody spent on a certain location. For instance, if a user goes grocery shopping in the morning and visits a theatre in the evening, it will not be possible to see the difference in time between the morning and the evening. Even more so, the route taken in the morning will be connected to the one taken in the evening. To avoid these kind of problems, some sense of time should be added to the route tracking on the website.

PermGen space error

There is a problem with garbage collecting in Tomcat in combination with JSP and Servlets. Deploying a project in Tomcat requires an amount of memory resources, however when the deploying is finished, not all resources are freed. This means that after every deployment a certain amount of memory is occupied, but never given back. This will cause the server to run out of memory slowly, resulting in a PermGen space error in the end. There are a lot of articles on the internet describing this problem, but a real solution hasn’t been found yet. Maybe this will be fixed in the next version of tomcat.

Friend requests

In the current Sococa system, users are able to create friendships between each another. When a user adds another user to its friends list, the other user is not notified. The other user should receive a message in which he can accept or reject their friendship.

OpenId log in via the client

At present time there is no library available to help facilitate logging in with an OpenId account without using a browser. When such a library becomes available, it should be implemented in the client.

8 Bibliography

1. Dana, P.H., *Geometric Dilution of Precision (GDOP) and Visibility*. 1st May 2000. 6 October 2008. http://www.colorado.edu/geography/gcraft/notes/gps/gps.html#Gdop

2. Elling, R., B. Andeweg, J. de Jong and C. Swankhuizen, *Rapportagetechniek*. Second edition. Groningen: Wolters-Noordhoff, 1999.

3. Hirschelmann, K.H., *NMEA 0183*. 7th August 2002. 8th September 2008. http://www.kh-gps.de/nmea-faq.htm

4. Lethbridge T.C. and R Laganière, *Object-Oriented Software Engineering*. Second edition. New York: McGraw-Hill, 2005.

5. Meyer B, *Applying "Design by Contract"*. 1992. IEEE. http://blackboard.tudelft.nl/courses/1/8034-070803/content/_787128_1/contract-ieee-comp-1992.pdf

6. MIT SENSEable City Lab, *Real Time Rome*. 3rd September 2008. http://senseable.mit.edu/realtimerome/

7. OpenId Foundation. *OpenID 2.0 specification.*3 September 2008. http://openid.net/specs/openid-authentication-2_0.html

8. Pezzè, M. and M. Young, *Software testing and analysis*. Hoboken: Wiley, 2008.

9. Sun Microsystems, *JavaServer Pages Technology*. 22th September 2008. http://java.sun.com/products/jsp/

10. Sun Microsystems, *Java Servlet technology overview*. 22th September 2008. http://java.sun.com/products/servlet/overview.html

11. Wikibooks. *Java Persistence/Transactions*. 30 September 2008. http://en.wikibooks.org/wiki/Java_Persistence/Transactions

12. World Wide Web Consortium, *HTTP/1.1 specification*. 23 September 2008. http://www.w3.org/Protocols/rfc2616/rfc2616.html

Appendix A
RAD document

RAD document

Gosse Bouma, Peter de Klerk, Kristian Slabbekoorn

Version 1.0

Status: final

1 Contents

_Toc218413849

1 Requirements

1.1 Introduction

We have divided the requirements into three parts:

- General:
  The general part contains all common information about the system. General requirements begin with the letter G.
- Backend:
  This part is about the specific requirements associated with the backend system, like input and output. Backend requirements begin with the letters BE.
- User Interface:
  Specific requirements about the user interface, denoted as UI

1.2 General requirements

Quality

G1. Response times between real-time upload and display on the map must be reasonable (lower than 500ms)

G2. Throughput should be sufficient, i.e. it should be able to handle many people at a time (at least 1000 people)

G3. The system should provide an API for easy implementation by other tools, and should be kept well documented and clearly structured:

3.1. The followed coding style should be the official Java coding standard

3.2. The reference manual will be provided as JavaDoc

3.3. Methods and classes should be implemented using the Design by Contract methodology

3.4. All methods have to be well tested using an automated testing environment (JUnit)

Platform

G4. The system will be written in the Java programming language

Process

G5. A prototype consisting of only reading GPS data and displaying the locations on the map must be developed first, and extended afterwards

G6. The system should be delivered before the 19th of December

1.3 Backend requirements

Functional requirements

BE 1. Types of input
- 1.1. The system accepts GPS data (NMEA 0183 protocol)
- 1.2. The system accepts XML data

BE 2. Methods of input
- 2.1. Input using a log file (either in XML or GPS data format)
- 2.2. Real-time

BE 3. Storage
- 3.1. Incoming data is saved in a centralized database
- 3.2. Incoming data is saved in a P2P distributed database

BE 4. Caching is used in database to speed up processing

BE 5. Interfaces are designed for:
- 5.1. Database access
- 5.2. Input through GPS Data
- 5.3. Input through XML files

Quality requirements

BE 6. Response time from database is fast (< 100 ms)

Platform requirements

BE 7. Programming languages
- 7.1. The database will be a MySQL database
- 7.2. The database will be connected to using JDBC

1.4 User Interface requirements

Functional

UI 1. Map views
- 1.1. User can display his current position on the map
- 1.2. User can change his visibility (visible by everyone, only friends or only himself)
- 1.3. User can choose to be anonymous
- 1.4. User can display a route on the map by hour, day, week, month, year
- 1.5. User can display other people's position on the map
- 1.6. The real-time location of the user and other people are animated on the map

UI 2. The web-interface supports OpenID authentication

UI 3. The web-interface implements the OpenSocial API

UI 4. Friends list
- 4.1. User can set his own status
- 4.2. User can add a friends list from an SNS
- 4.3. User can add a friend to his friends list
- 4.4. User can delete a friend from his list
- 4.5. User can chat with a friend

UI 5. Blog
- 5.1. User can automatically generate a blog entry
- 5.2. User can modify the generated blog entry
- 5.3. User can upload the generated blog entry to another SNS
- 5.4. User can choose to auto-upload generated blog entries to his SNS (in real-time mode)

UI 6. Photo sharing
- 6.1. User can add photos to his routes
- 6.2. User can delete photos from his routes
- 6.3. User can view photos at friends routes

UI 7. Sight recommendation
- 7.1. User can get a summary of sights nearest to him
- 7.2. User can get recommendations of sights his friends/other people with similar tastes have been

UI 8. Searching options
- 8.1. User can search for a person

Quality

UI 9. Extensibility – it must be easy for developers to add more windows with mash ups/other functionality to the portal

Platform

UI 10. The web application is properly viewable in Mozilla Firefox

UI 11. The web application is properly viewable in IE, Opera, Safari, etc.

UI 12. The map effects and animations are implemented using Processing

1.5 Main Functionality - MoSCoW/Dependency Diagram

Some requirements are dependent on others. Implementing a requirement without having implemented its dependency is impossible. We have made two dependency diagrams: table 1 and table 2. Table 1 consists of the main functionality, whereas table 2 consists of the sample applications.

Dependencies:

- D - The horizontal component is dependent on the vertical component.

The last column represents a MoSCoW classification. All requirements are assigned a specific priority (see legend below).

MoSCoW legend:

- **M** - MUST have this.
- **S** - SHOULD have this if at all possible.
- **C** - COULD have this if it does not affect anything else.
- **W** - WOULD like to have in the future.

Table 1

	BE 1.1	BE 1.2	BE 2.1	BE 2.2	BE 3.1	BE 3.2	BE 4	BE 5.1	BE 5.2	BE 5.3	UI 1.1	UI 1.2	UI 1.3	UI 1.4	UI 1.5	UI 1.6	UI 2	UI 3	MoSCoW
BE 1.1	X																		**M**
BE 1.2		X																	**S**
BE 2.1	D	D	X																**M**
BE 2.2	D	D		X															**M**
BE 3.1					X														**M**
BE 3.2						X													**C**
BE 4					D		X												**S**
BE 5.1								X											**M**
BE 5.2	D								X										**M**
BE 5.3		D								X									**S**
UI 1.1	D			D							X								**M**
UI 1.2											D	X							**M**
UI 1.3													X						**M**
UI 1.4											D		D	X					**M**
UI 1.5				D											X				**M**
UI 1.6				D												X			**M**
UI 2																	X		**S**
UI 3																		X	**S**

1.6 Sample Applications - MoSCoW/Dependency Diagram

Table 2

	UI 4.1	UI 4.2	UI 4.3	UI 4.4	UI 4.5	UI 5.1	UI 5.2	UI 5.3	UI 5.4	UI 6.1	UI 6.2	UI 6.3	UI 7.1	UI 7.2	UI 8.1	MoSCoW
UI 4.1	X															S
UI 4.2		X														S
UI 4.3			X													C
UI 4.4			D	X												S
UI 4.5				D	X											C
UI 5.1						X										C
UI 5.2						D	X									C
UI 5.3						D		X								C
UI 5.4						D			X							C
UI 6.1										X						W
UI 6.2										D	X					W
UI 6.3												X				W
UI 7.1													X			W
UI 7.2														X		W
UI 8.1															X	W

2 Use cases

Use case 1: Real time uploading of location data
Requirements: BE 2.2; BE 3.1
Steps

Actor actions	*System responses*
1. The user starts the client	2. Go to site for log-in
3. The user gives his log-in details	4a. Log user in, notify client program
	4b. Start updating user's current position
	4c. Show user's current positions on map

Use case 2: Show own current position in map
Requirements: UI 1.1
Actors: User
Preconditions: User is logged in. User's current position is known in the system.
Summary: The user's current position is represented in a map.
Steps:

Actor actions	*System responses*
1. Choose "show current position" command	2. Display a map with user's current position.

Use case 3: Change visibility
Requirements: UI 1.2
Actors: User
Preconditions: User is logged in.
Summary: The user changes who can see him on their map.
Steps:

Actor actions	*System responses*
1. Choose "edit preferences" command	2. Display user's preferences
3. Change "visibility" option (everyone, friends, self)	4a. Preferences are updated
	4b. Position data is uploaded appropriately.

Use case 4: Change visibility to anonymous
Requirements: UI 1.3
Actors: User
Preconditions: User is logged in.
Summary: The user chooses to be anonymous, his name will not be displayed in other people's maps.
Steps:

Actor actions	*System responses*
1. Choose "edit preferences" command	2. Display user's preferences
3. Check "anonymous" option	4a. Preferences are updated
	4b. Other people's maps are updated appropriately.

Use case 5: Animate a route in map
Requirements: UI 1.4
Actors: User
Preconditions: User is logged in. One or more locations of the user are known to the system.
Summary: The user's route is represented in a map.
Steps:

Actor actions	*System responses*
1. Choose "show route" command and supplies a 'from' date and a 'to' date	2. Display a map with a representation of the user's route.

Use case 6: Upload a log file (GPS or XML)
Requirement(s): UI 2.1; BE 2.1; BE 3.1
Actors: User
Goals: Upload a number of locations and save them in the database.
Preconditions: User is logged in. The user has connected his GPS device to his PC.
Steps

Actor actions	*System responses*
1. User gives "add log" command	2. System opens "browse file" dialog
3. User selects the desired log file and presses upload.	4a. Systems parses file and saves it's data in database
	4b. Show an uploading successful message
	4c. Show the map displaying the uploaded route

Postconditions
The system has saved the route's locations in the database for later reference. The map displays the uploaded route.

Use case 7: Delete a location
Requirement(s): UI 2.2
Preconditions: User is logged in and has uploaded at least one location.
Steps

Actor actions	*System responses*
1. User selects a location from the map	2. System highlights the selected location
3. User gives the "delete locaiton" command	4. Prompt for confirmation
4. User selects 'yes'	5a. The location is deleted from the database
	5b. The location is no longer shown on map

Use case 8: Set visibility of your location
Requirement(s): UI 2.3
Preconditions: User is logged in. User has added at least one location.
Steps

Actor actions	*System responses*
1. User gives the "only visible for friends" command	2. System makes the user's location only visible for friends

Use case 9: View a friends location
Requirement(s): UI 2.4
Preconditions: User is logged in. User has at least one friend. This friend has added at least one location.
Steps

Actor actions	*System responses*
1. User selects a friend	2. System show friend's options
3. User gives "show location" command	4. System shows friend's location

Use case 10a: Logging in to the site when logged in with openID provider
Requirement: UI3 (S)
Actors: User
Goals: To log in to the site and be able to see your location, change settings, etc.
Preconditions: The user must know his username and password, or be logged in with an OpenID provider
Summary: When a user wishes to log on, he should enter his credentials first, assuming he hasn't logged in at some other site providing openID authentication. After having logged in, he could see his and his friends' locations, change his profile, etc.
Related use case: Logging out of the site
Steps:

Actor actions	*System responses*
1 Browse to website	2a. Check with openID provider if user is logged in yet 2b. User is logged in with OpenID provider, log in the user

Postconditions: User is logged in

Use case 10b: Logging in to the site when not logged in with OpenID provider
Requirement: UI3
Related use case:
Extension of: 10a. Logging in to site (extension point: 2a Check OpenId provider)
Steps:

Actor actions	*System responses*
	2a. Check with openID provider if user is logged in yet 2b. User is not logged in yet
3. Choose login command	4 Prompt user for login/password
5. Input openID login and password	6a. Check login/password with openID provider 6b. Log in user if authorized, otherwise reject

Use case 11: Logging out of the site
Requirement: UI3 (S)
Actors: User
Related use case: Logging in to the site
Steps:

Actor actions	*System responses*
1 Use logout command	2. Log out the user at the openID provider

Use case 12: Setting your status in friend list (online, away, etc)
Requirement: UI5.1 (S)
Actors: User
Preconditions: The user must be logged in
Steps:

Actor actions	*System responses*
1 Open friendlist options	2. Show friendlist options
3 Set desired status	4 Display new status to user and all users in friendlist

Postconditions: New status is set

Use case 13: Add friends list from other SNS
Requirement: UI5.2 (C)
Actors: User
Goals: The user has transferred his friends list from another OpenSocial-based SNS.
Preconditions: The user must be logged in, and must have a friend base at a supported SNS.
Steps:

Actor actions	*System responses*
1 Open friendlist options	2 Show options for friendlist
3 Select 'add friend list'	4 Show prompt
5 Select network to import from	6 Show authentication prompt
7 Input username and password	8 Import friendlist and show them in user's profile

Postconditions: The friendlist is imported from the external SNS.

Use case 14: Add friend to friendlist
Requirement: UI5.3 (S)
Actors: User
Goals: The user has added a friend to his list.
Preconditions: The user must have friends.
Steps:

Actor actions	*System responses*
1 Open friendlist options	2 Show options for friendlist
3 Select 'add friend'	4 Show prompt to input nickname, name etc. of friend
5 Input a friend's details	6a Add friend
	6b Show friend in user list

Postconditions: The friend has been added to the list.

Use case 15: Delete friend from friendlist
Requirement: UI5.4 (S)
Actors: User
Goals: The user has deleted a friend from his list.
Preconditions: The user must have friends.
Steps:

Actor actions	*System responses*
1 Open friendlist friend options	2 Show options for friend
3 Select 'delete friend'	4 Prompt for confirmation to delete the friend
5 Select 'yes' to delete friend, 'no' to cancel	6 If yes, delete friend, otherwise remove prompt and do nothing

Postconditions: The friend has been deleted from the list.

Use case 16: User can chat with a friend
Requirement: UI5.5 (C)
Actors: User
Goals: The user has exchanged a text message with a friend.
Preconditions: The user must have friends.
Steps:

Actor actions	*System responses*
1 Select friend	2 Highlight friend
3 Select chat command	4 Show chat window
5 Input message	6 Transfer and show message to recipient

Postconditions: The friend has been shown the message.

Use case 17: User can automatically generate a blog entry
Requirement: UI6.1 (C)
Actors: User
Goals: The user has generated a blog entry.
Preconditions: The user must be/have been somewhere, user must be logged in.
Summary: The user can generate a blog entry for famous places he has been. After having visited such a place, the user can use the 'generate blog' option to generate a short description of the place he has visited, when he has visited, etc.
Steps:

Actor actions	*System responses*
1 Select 'generate blog' command	2a Parse log file to gather landmarks known in DB 2b Generate text with timestamp and landmark and post to user's page

Postconditions: The blog entry has been generated and posted

Use case 18: User can modify the generated blog entry
Requirement: UI6.2 (C)
Actors: User
Goals: The user has modified his blog entry.
Preconditions: A blog entry bust have been generated.
Steps:

Actor actions	*System responses*
1 Select 'edit blog' command at desired entry	2 Show edit screen to user
3 Edit blog entry and submit	4 Save and display to user

Postconditions: The blog entry has been updated

Use case 19: User can upload the generated blog entry to another SNS
Requirement: UI6.3 (C)
Actors: User
Goals: The user has uploaded the blog entry to another, compatible SNS (i.e. MySpace).
Preconditions: User must be logged in, user must have account at external SNS
Steps:

Actor actions	*System responses*
1 Select 'export blog entry' command	2 Show export form to user
3 Fill in export form, submit	4 Send the blog entry to external SNS and post it there

Postconditions: The blog entry has been uploaded and posted

Use case 20: User can choose to upload generated blog entries automatically (real-time)
Requirement: UI6.4 (C)
Actors: User
Goals: The activities of the user are being uploaded to his blog continuously
Summary: A user can activate the automatic generation of blog entries on his page. A short, automated description of every landmark the user has visited will be posted to his blog in real-time.
Preconditions: User must be logged in, user must have switched on real-time location updating
Steps:

Actor actions	*System responses*
1 Select 'auto-upload blog entries' command	2a Notify the mode has been enabled 2b Upload blog entry whenever the user passes a landmark

Postconditions: The blog entries will have been generated and uploaded

Use case 21: Add photo to location.
Requirements: UI 7.1
Actors: User
Preconditions: User is logged in. The user must have a location selected.
Summary: User chooses to add a photo to the location; it will be uploaded to the site and displayed as a clickable thumbnail.
Steps:

Actor actions	*System responses*
1. Choose "add photo" command	2. Open add photo page
3a. Choose "browse" and select the photo file	4a. Return to main profile page
3b. Fill in a name and description	4b. Display photo name and thumbnail under the location
3c. Choose "upload" command	

Use case 22: View photo
Requirements: UI 7.1
Actors: User
Preconditions: User is logged in. A location of the user must be selected, and photo(s) must be added to that location.
Summary: User selects a thumbnail to see the full photo and description.
Steps:

Actor actions	*System responses*
1. Select a thumbnail	2. Open a new window with the full photo and description

Use case 23: Edit photo
Requirements: UI 7.2
Actors: User
Preconditions: User is logged in. One of the user's locations is selected; photo(s) are added to this location.
Summary: User chooses to edit the photo; the photo is updated on the site.
Steps:

Actor actions	*System responses*
1. Choose "edit photo" command	2. Open edit photo page.
3a. Change name and/or description	4a. Return to main profile page
3b. Choose "edit" command	4b. Display edited photo name

Use case 24a: Remove photo
Requirements: UI 7.2
Actors: User
Preconditions: User is logged in. A user's own location is selected; photo's are added to this location.
Summary: User chooses to remove a photo; the photo is removed from the site.
Related use cases:
Extension: Cancel removing photo
Steps:

Actor actions	*System responses*
1. Choose "remove photo" command	2. Open confirmation dialog
3. Choose "yes"	4. Remove photo and display profile page.

Use case 24b: Cancel removing photo
Related use cases:
Extension of: Remove photo
Steps:

Actor actions	*System responses*
1. Choose "remove photo" command	2. Open confirmation dialog
3. Choose "no"	4. Do nothing.

Use case 25: View someone else's photos
Requirements: UI 7.3
Actors: User
Preconditions: User is logged in. Someone else's location must be selected, and photo(s) must be added to that location. That person must have route sharing set to "everyone", or "friends" if the user is friends with that person.
Summary: User selects a thumbnail to see the full photo and description.
Steps:

Actor actions	*System responses*
1. Select a thumbnail	2. Open a new window with the full photo and description

Use case 26: Get sights summary
Requirements: UI 8.1
Actors: User
Preconditions: User is logged in. User must have a current position
Summary: User requests a summary of sights near him, system generates and displays it.
Steps:

Actor actions	*System responses*
1. Choose “sights summary” command	2. Loads sights into map
3. Choose a sight	4. Display popup with picture and description of sight.

Use case 27: Recommend sights
Requirements: UI 8.2
Actors: User
Preconditions: User is logged in. User must have a current position.
Summary: User requests a lists sights recommendations near him, system generates and displays it.
Steps:

Actor actions	*System responses*
1. Choose “recommend sights” command	2. Loads recommended sights into map.
3. Choose a sight	4. Display a popup with sight information and popularity

Use case 28: Search for a person
Requirement(s): UI 9.1
Preconditions: User is logged in.
Steps

Actor actions	*System responses*
1. User types a person’s name and gives the “search person” command	2. System shows all persons matching the name typed by the user

Use case 29: Search for a location
Requirement(s): UI 9.2
Preconditions: User is logged in.
Steps

Actor actions	*System responses*
1. User types a town name or famous spot and gives the “search location” command	2. System shows a list of all routes containing the town or famous spot.

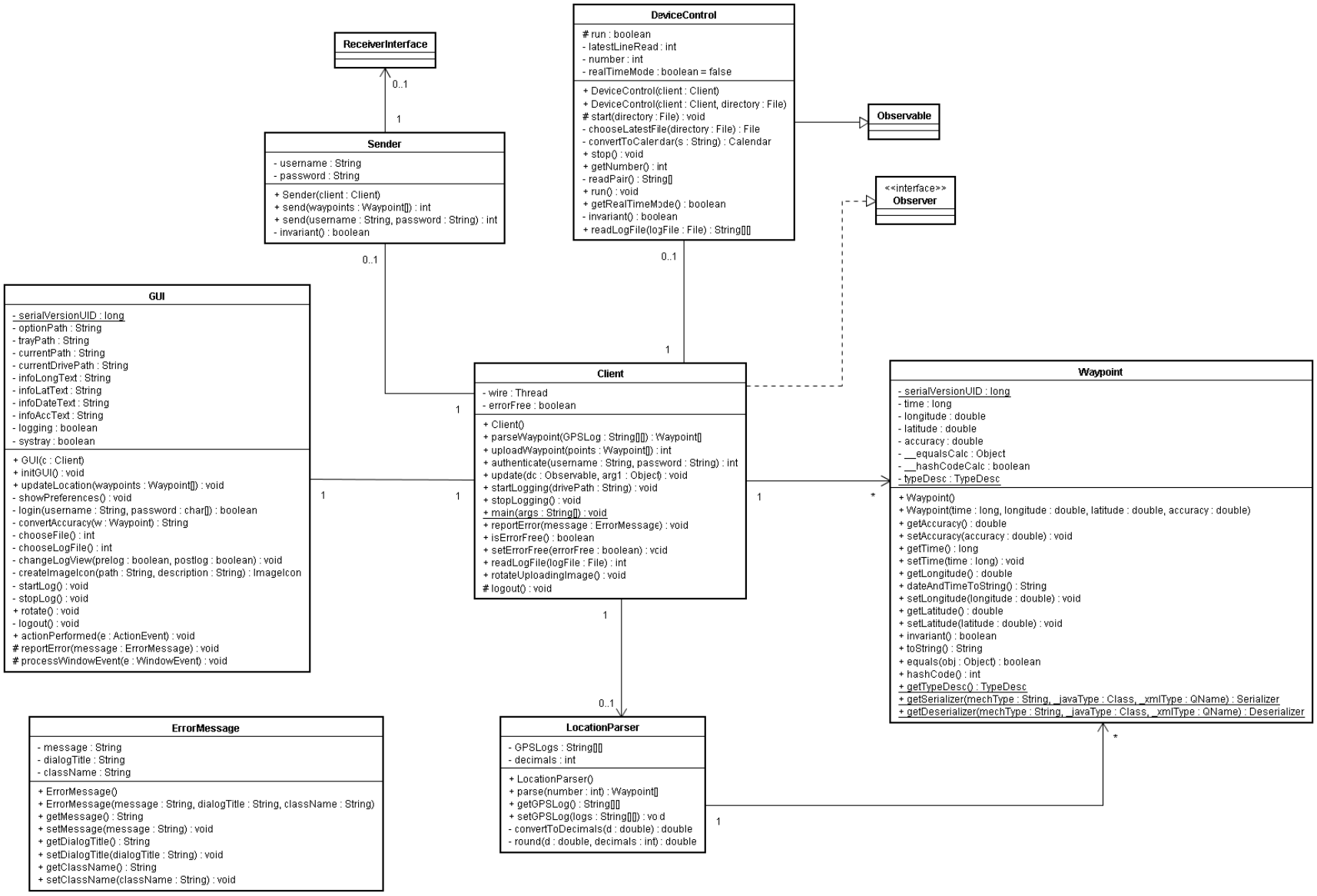
3 Class diagrams

This chapter contains the UML class diagrams for the Sococa system. The parts of the system are divided up into class diagrams: A client, a server, storables, a parser and the servlets. We will review them in this order.

3.1 Client

For real time logging of GPS data, a user has to have a client program installed on his computer. This client will contact the server and update the whereabouts of the user. The user can authenticate itself to the client program by supplying his OpenId name and the client password.

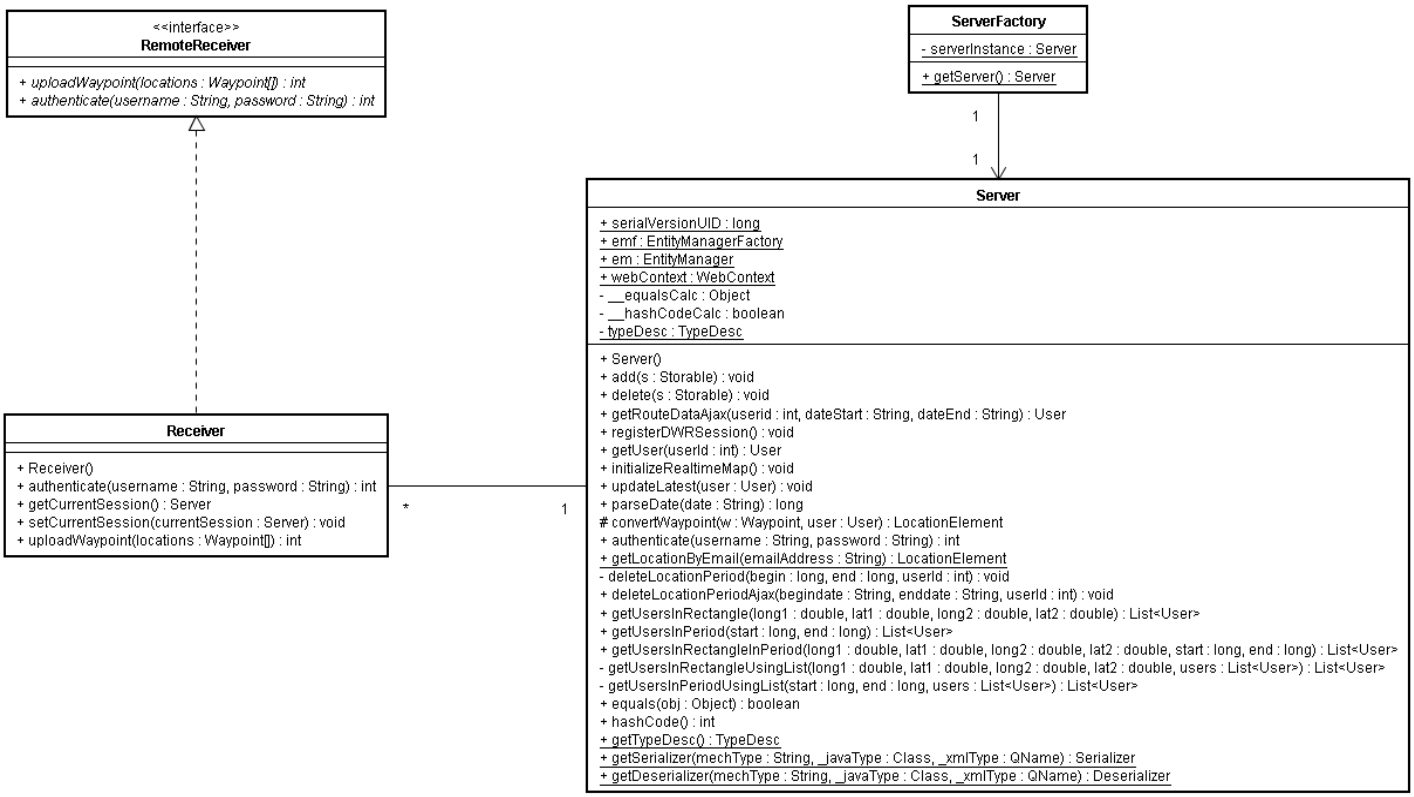
The client uses Apache Axis to communicate with the server, a technology based on JAX-RPC. Java objects will be represented as XML SOAP messages and sent to the server. At the server the messages are decoded into Java objects, so the server can update the user's location.


ReceiverInterface
0..1
1
Sender
- username : String
- password : String
+ Sender(client : Client)
+ send(waypoints : Waypoint[]) : int
+ send(username : String, password : String) : int
- invariant() : boolean
DeviceControl
run : boolean
- latestLineRead : int
- number : int
- realTimeMode : boolean = false
+ DeviceControl(client : Client)
+ DeviceControl(client : Client, directory : File)
# start(directory : File) : void
- chooseLatestFile(directory : File) : File
- convertToCalendar(s : String) : Calendar
+ stop() : void
+ getNumber() : int
- readPair() : String[]
+ run() : void
+ getRealTimeMode() : boolean
- invariant() : boolean
+ readLogFile(logFile : File) : String[][]
Observable
<<interface>>
Observer
0..1
0..1
1
GUI
- serialVersionUID : long
- optionPath : String
- trayPath : String
- currentPath : String
- currentDrivePath : String
- infoLongText : String
- infoLatText : String
- infoDateText : String
- infoAccText : String
- logging : boolean
- systray : boolean
+ GUI(c : Client)
+ initGUI() : void
+ updateLocation(waypoints : Waypoint[]) : void
- showPreferences() : void
- login(username : String, password : char[]) : boolean
- convertAccuracy(w : Waypoint) : String
- chooseFile() : int
- chooseLogFile() : int
- changeLogView(prelog : boolean, postlog : boolean) : void
- createImageIcon(path : String, description : String) : ImageIcon
- startLog() : void
- stopLog() : void
+ rotate() : void
- logout() : void
+ actionPerformed(e : ActionEvent) : void
# reportError(message : ErrorMessage) : void
# processWindowEvent(e : WindowEvent) : void
1
1
1
1
Client
- wire : Thread
- errorFree : boolean
+ Client()
+ parseWaypoint(GPSLog : String[][]) : Waypoint[]
+ uploadWaypoint(points : Waypoint[]) : int
+ authenticate(username : String, password : String) : int
+ update(dc : Observable, arg1 : Object) : void
+ startLogging(drivePath : String) : void
+ stopLogging() : void
+ main(args : String[]) : void
+ reportError(message : ErrorMessage) : void
+ isErrorFree() : boolean
+ setErrorFree(errorFree : boolean) : void
+ readLogFile(logFile : File) : int
+ rotateUploadingImage() : void
# logout() : void
1
1
*
Waypoint
- serialVersionUID : long
- time : long
- longitude : double
- latitude : double
- accuracy : double
- __equalsCalc : Object
- __hashCodeCalc : boolean
- typeDesc : TypeDesc
+ Waypoint()
+ Waypoint(time : long, longitude : double, latitude : double, accuracy : double)
+ getAccuracy() : double
+ setAccuracy(accuracy : double) : void
+ getTime() : long
+ setTime(time : long) : void
+ getLongitude() : double
+ dateAndTimeToString() : String
+ setLongitude(longitude : double) : void
+ getLatitude() : double
+ setLatitude(latitude : double) : void
+ invariant() : boolean
+ toString() : String
+ equals(obj : Object) : boolean
+ hashCode() : int
+ getTypeDesc() : TypeDesc
+ getSerializer(mechType : String, _javaType : Class, _xmlType : QName) : Serializer
+ getDeserializer(mechType : String, _javaType : Class, _xmlType : QName) : Deserializer
0..1
*
ErrorMessage
- message : String
- dialogTitle : String
- className : String
+ ErrorMessage()
+ ErrorMessage(message : String, dialogTitle : String, className : String)
+ getMessage() : String
+ setMessage(message : String) : void
+ getDialogTitle() : String
+ setDialogTitle(dialogTitle : String) : void
+ getClassName() : String
+ setClassName(className : String) : void
LocationParser
- GPSLogs : String[][]
- decimals : int
+ LocationParser()
+ parse(number : int) : Waypoint[]
+ getGPSLog() : String[][]
+ setGPSLog(logs : String[][]) : void
- convertToDecimals(d : double) : double
- round(d : double, decimals : int) : double
1

3.2 Server

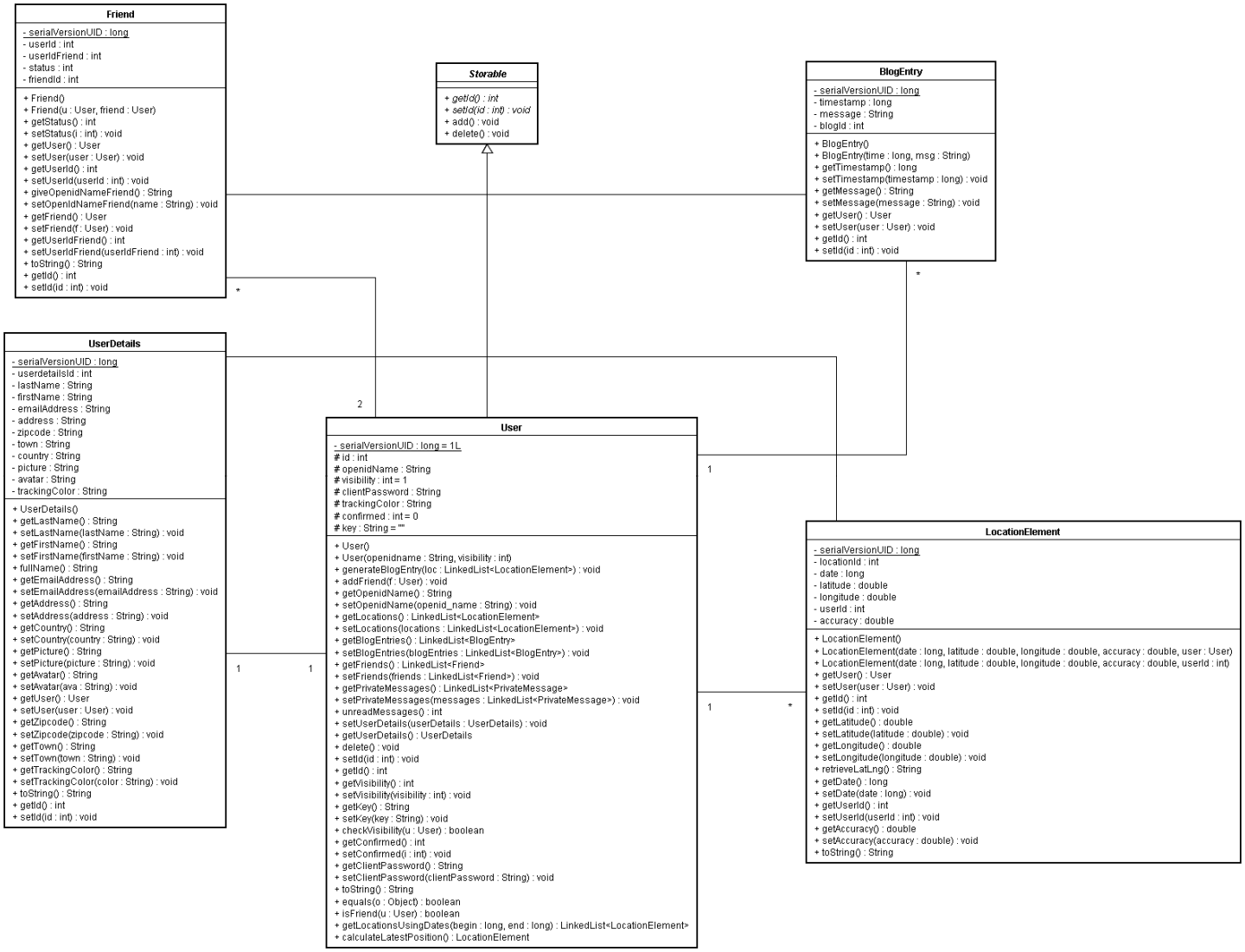
The server is responsible for all data management in the system. The server will be implemented using the Java Persistence API (JPA). Each persistence entity is a java class and represents a table in a relational database. Each instance of a class represents a row in a table. Adding an object to the database will now only take a *persist* method call instead of huge lines of code converting an object to an SQL-query. Typical JPA classes are Storables such as User, Friend and BlogEntry.

The system has a special class called the Receiver for handling the (real time) uploading of Waypoints. For every user connection a new instance of Receiver is made. All methods named in the RemoteReceiver interface can be called from remote systems. In our system remote systems can authenticate themselves and send waypoints to the server.



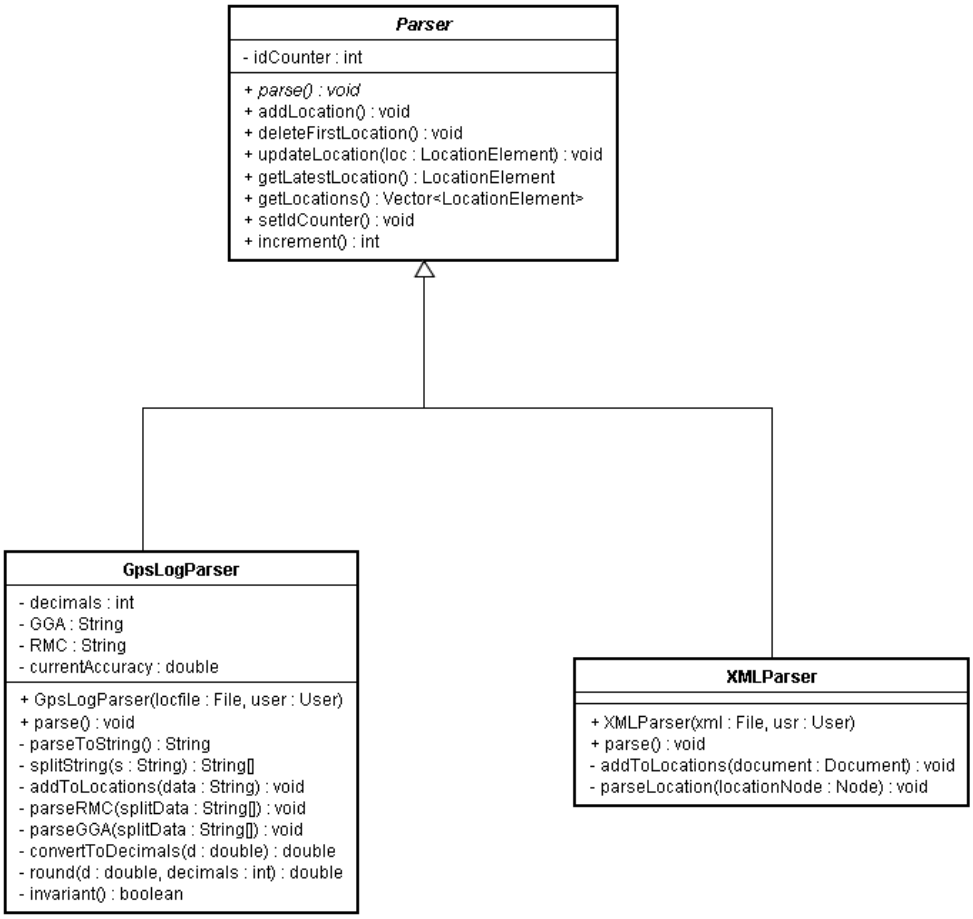
3.3 Storables

Each class which has to be stored in the database implements the Storable interface. This interface consists of two methods: add and delete. These two methods describe how an object of this type is supposed to be saved to the database. The server class also has these two methods with a Storable parameter. The server can now add and delete Storables to the database without having to know which subtype it's using. This is an example of polymorphism and enhances the system's extensibility.



3.4 Parser

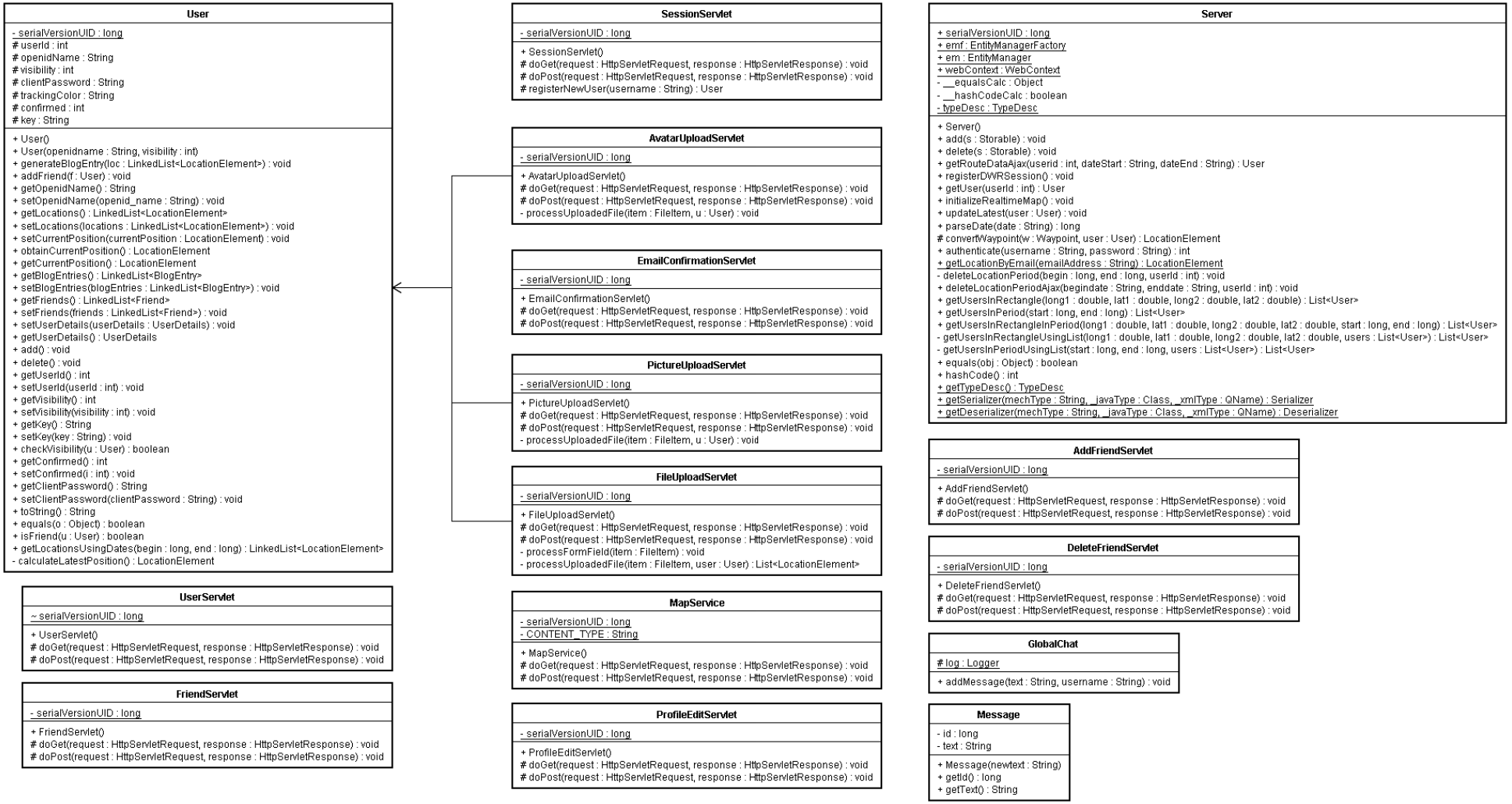
The parser has two subclasses: An XML-parser and a GPSLog-parser. Both have the abstract method parse implemented, so data can always be parsed by calling this method to the parser.



3.5 Servlets

Our system also has a web interface in which the user can set his preferences. The interface also contains a map which shows the users movement and the movements of others.
The interface is implemented using HttpServlets and JSP pages. These pages are basically a combination of Java, javascript and HTML. The map animations will implemented using Javascript and the Google API will provide the map itself.

Diagram on next page.

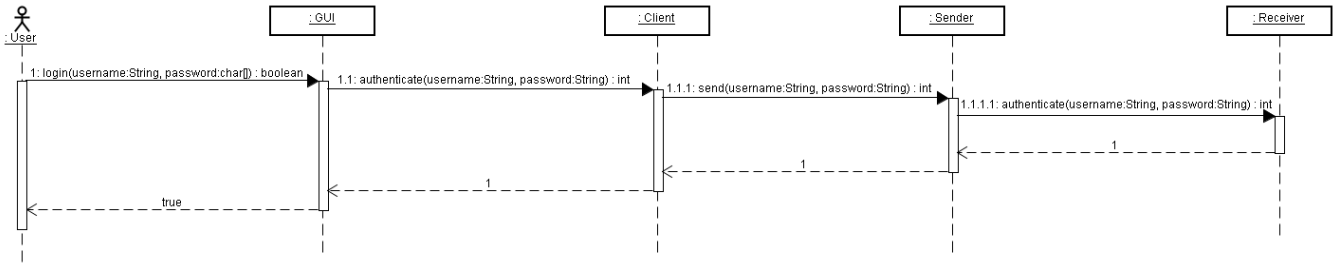

User
- serialVersionUID : long
userId : int
openidName : String
visibility : int
clientPassword : String
trackingColor : String
confirmed : int
key : String
+ User()
+ User(openidname : String, visibility : int)
+ generateBlogEntry(loc : LinkedList<LocationElement>) : void
+ addFriend(f : User) : void
+ getOpenidName() : String
+ setOpenidName(openid_name : String) : void
+ getLocations() : LinkedList<LocationElement>
+ setLocations(locations : LinkedList<LocationElement>) : void
+ setCurrentPosition(currentPosition : LocationElement) : void
+ obtainCurrentPosition() : LocationElement
+ getCurrentPosition() : LocationElement
+ getBlogEntries() : LinkedList<BlogEntry>
+ setBlogEntries(blogEntries : LinkedList<BlogEntry>) : void
+ getFriends() : LinkedList<Friend>
+ setFriends(friends : LinkedList<Friend>) : void
+ setUserDetails(userDetails : UserDetails) : void
+ getUserDetails() : UserDetails
+ add() : void
+ delete() : void
+ getUserId() : int
+ setUserId(userId : int) : void
+ getVisibility() : int
+ setVisibility(visibility : int) : void
+ getKey() : String
+ setKey(key : String) : void
+ checkVisibility(u : User) : boolean
+ getConfirmed() : int
+ setConfirmed(i : int) : void
+ getClientPassword() : String
+ setClientPassword(clientPassword : String) : void
+ toString() : String
+ equals(o : Object) : boolean
+ isFriend(u : User) : boolean
+ getLocationsUsingDates(begin : long, end : long) : LinkedList<LocationElement>
- calculateLatestPosition() : LocationElement
UserServlet
~ serialVersionUID : long
+ UserServlet()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
FriendServlet
- serialVersionUID : long
+ FriendServlet()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
SessionServlet
- serialVersionUID : long
+ SessionServlet()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
# registerNewUser(username : String) : User
AvatarUploadServlet
- serialVersionUID : long
+ AvatarUploadServlet()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
- processUploadedFile(item : FileItem, u : User) : void
EmailConfirmationServlet
- serialVersionUID : long
+ EmailConfirmationServlet()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
PictureUploadServlet
- serialVersionUID : long
+ PictureUploadServlet()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
- processUploadedFile(item : FileItem, u : User) : void
FileUploadServlet
- serialVersionUID : long
+ FileUploadServlet()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
- processFormField(item : FileItem) : void
- processUploadedFile(item : FileItem, user : User) : List<LocationElement>
MapService
- serialVersionUID : long
- CONTENT_TYPE : String
+ MapService()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
ProfileEditServlet
- serialVersionUID : long
+ ProfileEditServlet()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
Server
+ serialVersionUID : long
+ emf : EntityManagerFactory
+ em : EntityManager
+ webContext : WebContext
- __equalsCalc : Object
- __hashCodeCalc : boolean
- typeDesc : TypeDesc
+ Server()
+ add(s : Storable) : void
+ delete(s : Storable) : void
+ getRouteDataAjax(userid : int, dateStart : String, dateEnd : String) : User
+ registerDWRSession() : void
+ getUser(userId : int) : User
+ initializeRealtimeMap() : void
+ updateLatest(user : User) : void
+ parseDate(date : String) : long
# convertWaypoint(w : Waypoint, user : User) : LocationElement
+ authenticate(username : String, password : String) : int
+ getLocationByEmail(emailAddress : String) : LocationElement
- deleteLocationPeriod(begin : long, end : long, userId : int) : void
+ deleteLocationPeriodAjax(begindate : String, enddate : String, userId : int) : void
+ getUsersInRectangle(long1 : double, lat1 : double, long2 : double, lat2 : double) : List<User>
+ getUsersInPeriod(start : long, end : long) : List<User>
+ getUsersInRectangleInPeriod(long1 : double, lat1 : double, long2 : double, lat2 : double, start : long, end : long) : List<User>
- getUsersInRectangleUsingList(long1 : double, lat1 : double, long2 : double, lat2 : double, users : List<User>) : List<User>
- getUsersInPeriodUsingList(start : long, end : long, users : List<User>) : List<User>
+ equals(obj : Object) : boolean
+ hashCode() : int
+ getTypeDesc() : TypeDesc
+ getSerializer(mechType : String, _javaType : Class, _xmlType : QName) : Serializer
+ getDeserializer(mechType : String, _javaType : Class, _xmlType : QName) : Deserializer
AddFriendServlet
- serialVersionUID : long
+ AddFriendServlet()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
DeleteFriendServlet
- serialVersionUID : long
+ DeleteFriendServlet()
# doGet(request : HttpServletRequest, response : HttpServletResponse) : void
# doPost(request : HttpServletRequest, response : HttpServletResponse) : void
GlobalChat
log : Logger
+ addMessage(text : String, username : String) : void
Message
- id : long
- text : String
+ Message(newtext : String)
+ getId() : long
+ getText() : String

4 Sequence diagrams

This chapter contains sequence diagrams. The most import functions of the systems are described here. These are for the client: get data and parse data. And for the server: Uploading of waypoints, parsing a file, add a user, delete a location, add a friend, delete a friend and generate a blog entry.

4.1 Client: logging in

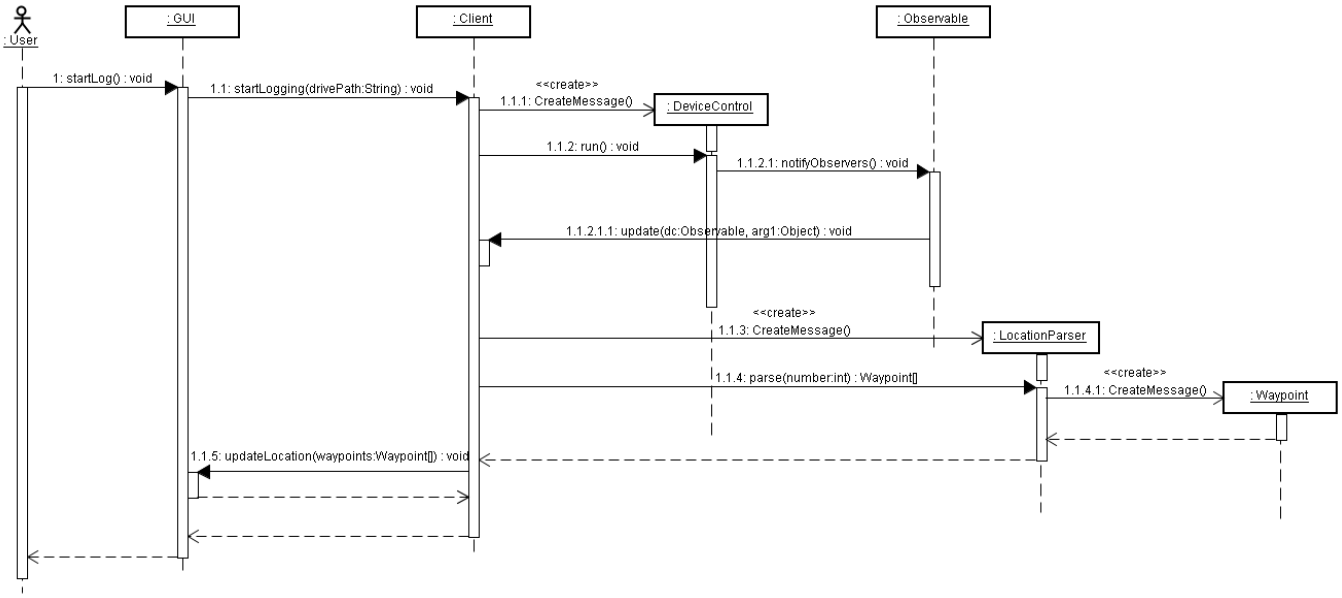
This diagram shows the user logs in to the client.



4.2 Client: Real-time logging

This diagram shows the real-time logging of waypoints. A DeviceControl thread is started which polls the GPS device. When new data is available, an update is sent to the Client. The Client then uses the Locationparser to parse the data and informs the GUI. How the waypoints are submitted to the server is shown in 2.3.

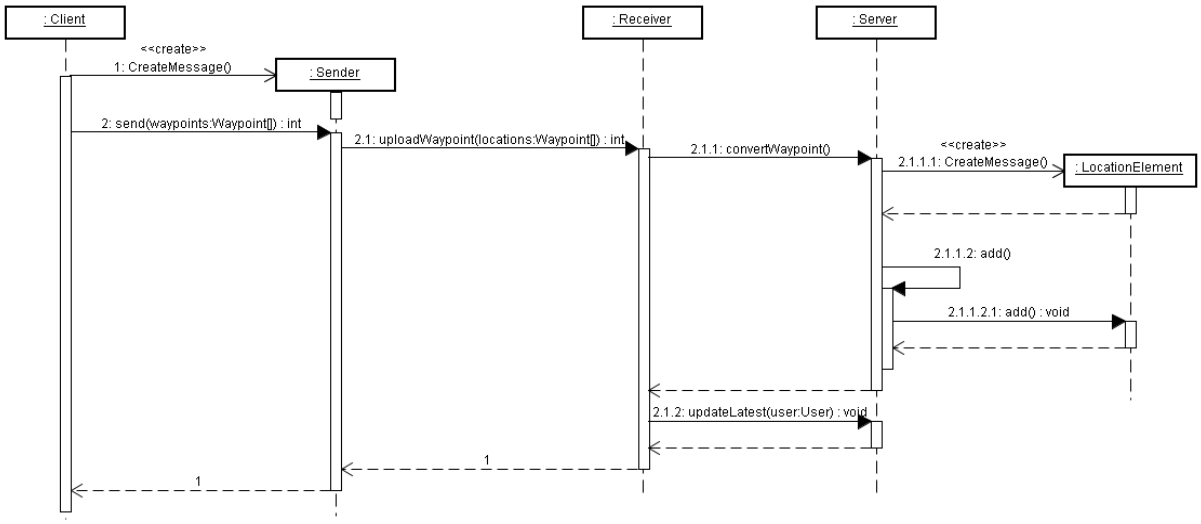
Involved use case: 1.



4.3 Uploading of waypoints

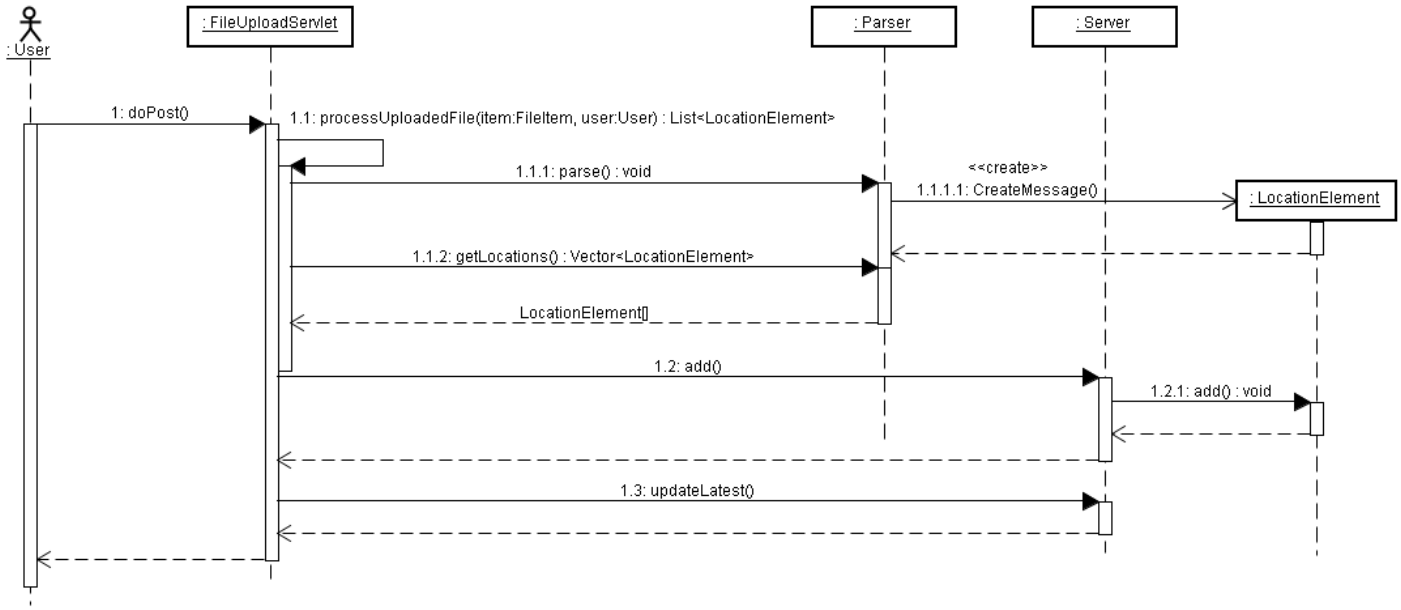
This diagram shows how waypoints are sent to the server and what happens when they arrive there. A sender is object is constructed, this object implements the RemoteReceiver interface of the server. Methods of the server can be called via sender as if it were a local object. Once the waypoints arrive at the server, they are converted to LocationElements and saved in the database by calling their add method. When the location is added to the database, the Receiver calls the updateLatest method. This method is used to directly update the map view of the website.

Involved use cases: 1,6.



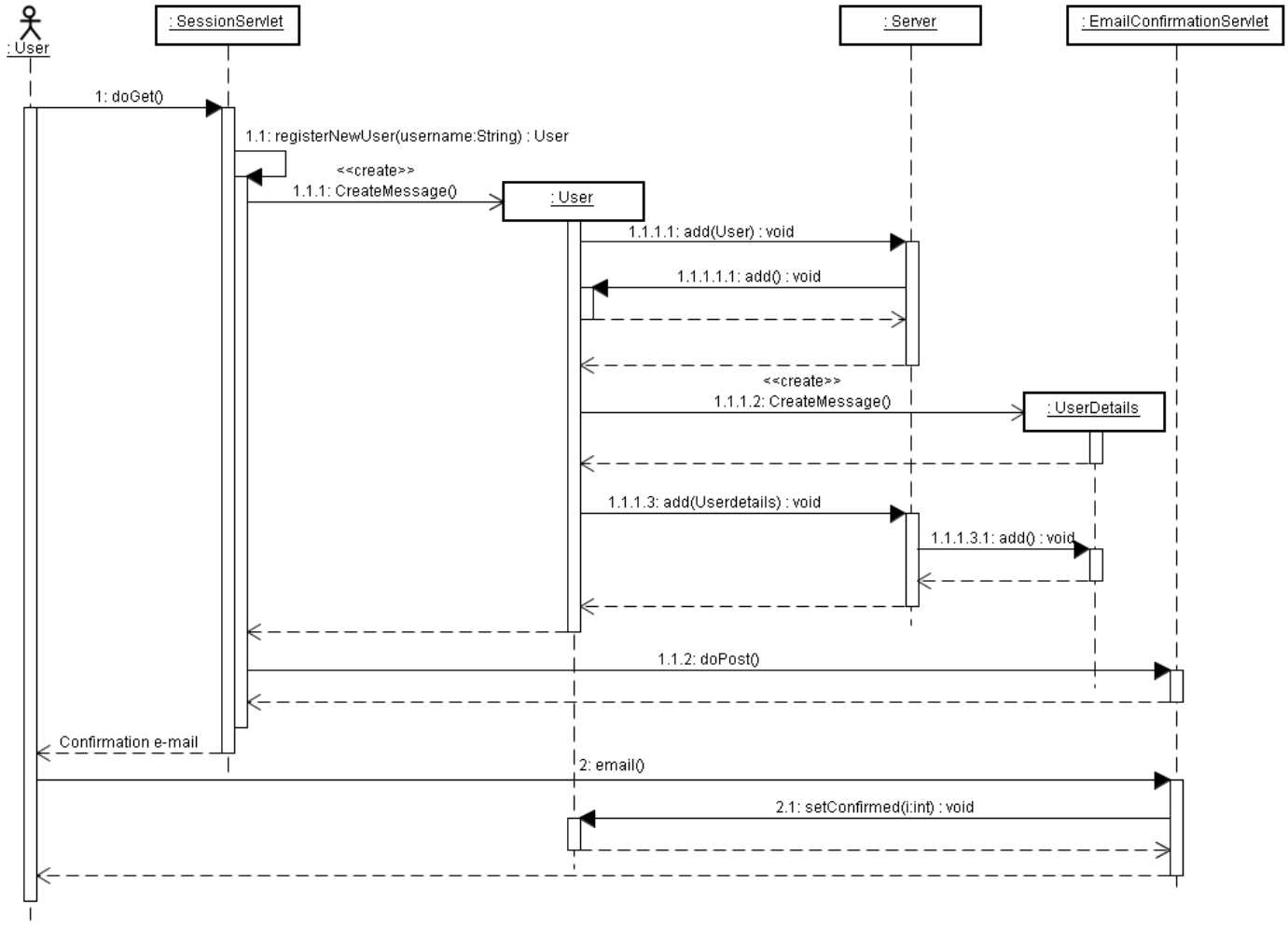
4.4 Parsing a file

This diagram shows how a file is uploaded to the server. The file is parsed to LocationElements and stored in the database. This diagram applies for both .log as xml-files, the difference is the parser which can either be a GPSParser (for .log files) or a XMLParser(for xml-files).
Involved use case: 6.



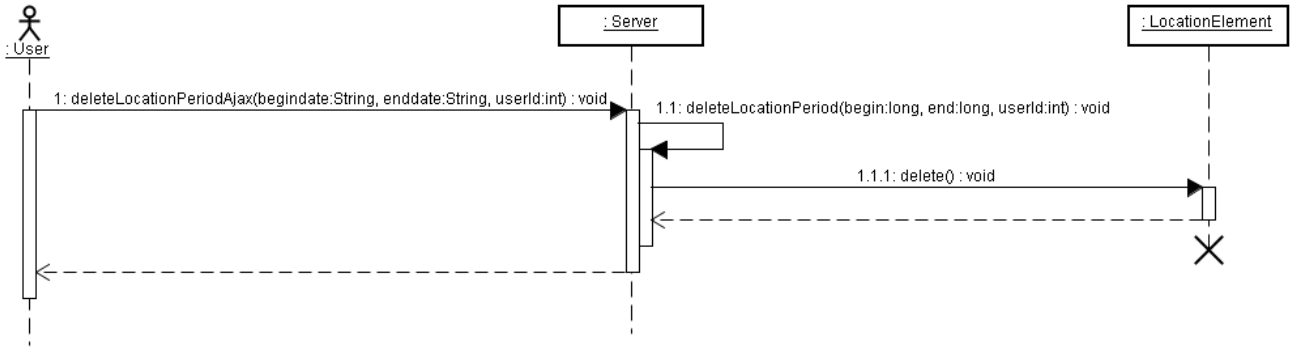
4.5 Add a user

This diagram shows a user registering on the website. The userServlet creates both the user object and the userdetails object. These objects are then given to the server which saves them to the database. When a user is constructed a confirmation e-mail is sent to his e-mail address, which the user has to answer to complete the registering process.



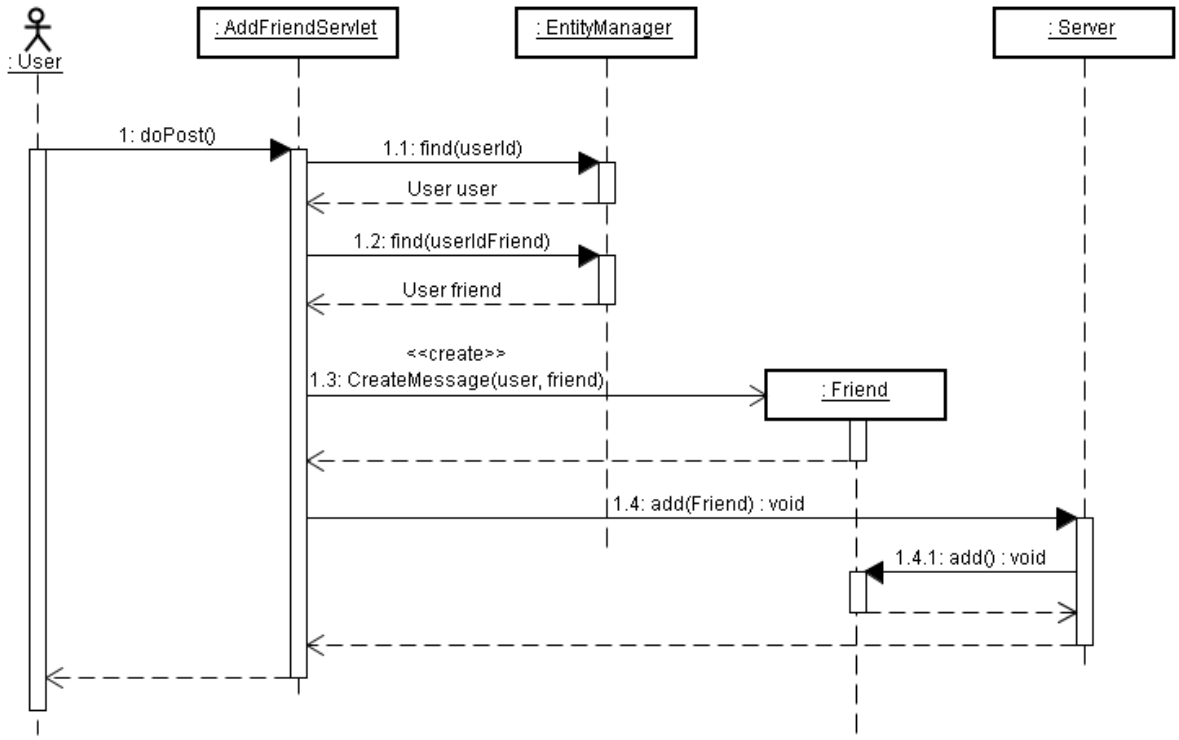
4.6 Delete a location

The user should be able to delete a location. This is done by selecting a location on the website and giving the delete command. The right location is then queried by the locationServlet, which passes the locationElement to the server. The server gives the final “delete” command.
Involved use case: 7.



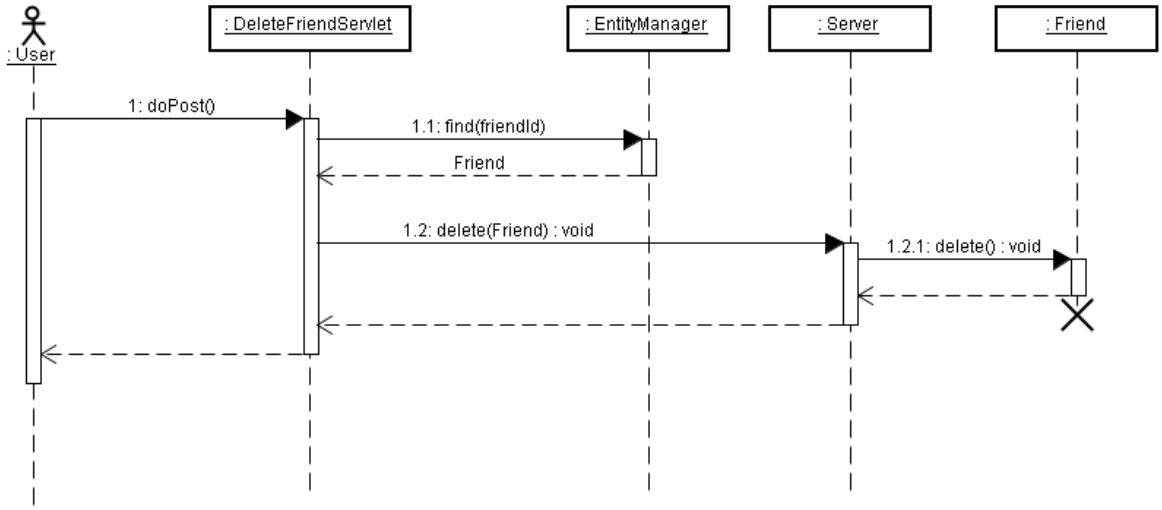
4.7 Add a friend

This diagram shows how a friend is added by a user. The user selects the user who he wants to become friends with and gives the “add friend” command. A new Friends object is created by the FriendServlet and passed on to the server. The server adds the friendship to the database and sets the status to unconfirmed. The status will be updated to “friends” when the “friend” user confirms the friendship.
Involved use case: 14.



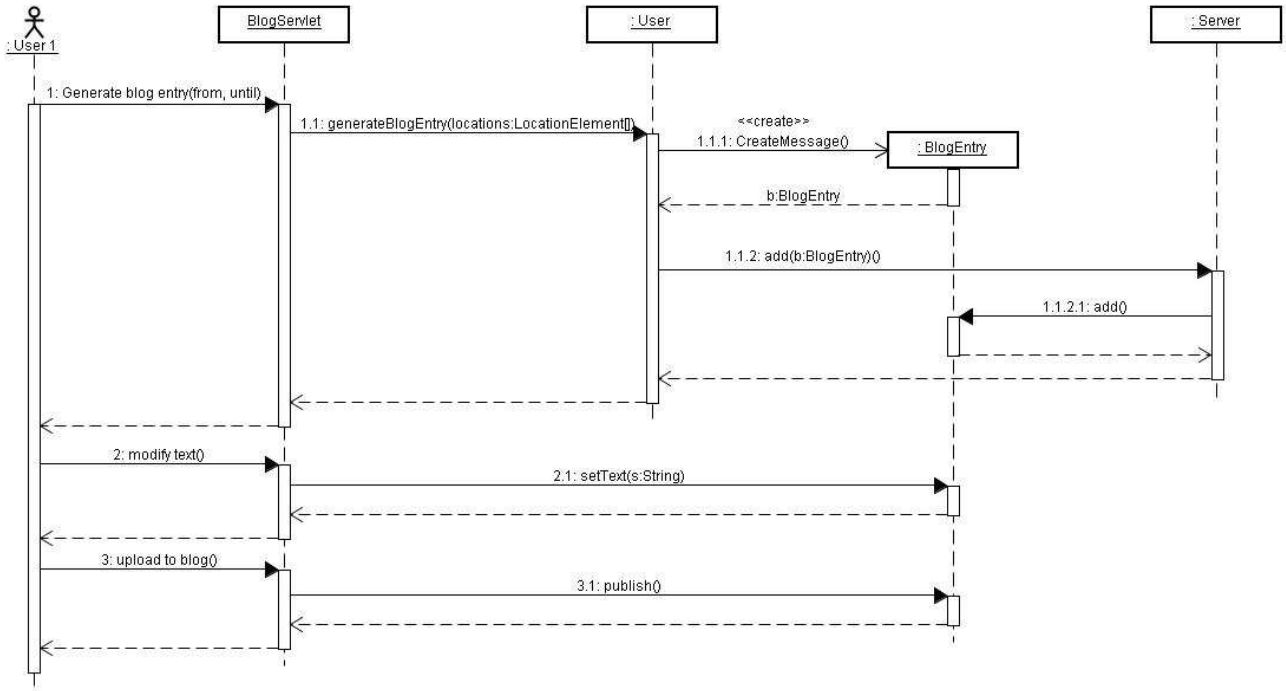
4.8 Delete a friend

Of course it should also be possible to undo a friendship. This is pretty straightforward: The friend is selected by the user and the delete command is given. The EntityManager queries for the right object which is then passed to the server. The server deletes the friend object from the database.
Involved use case: 15.



4.9 Generate blog entry

One of the sample applications is automatic blogging. This diagram shows how a blog entry is generated. The user selects a from and a till date. Using these dates the blog entry is generated. The blogEntry object is made by the blogServlet and passed to the server which saves it to the database. The user can choose to edit the generated blog entry before publishing it.
Involved use cases: 17, 18.

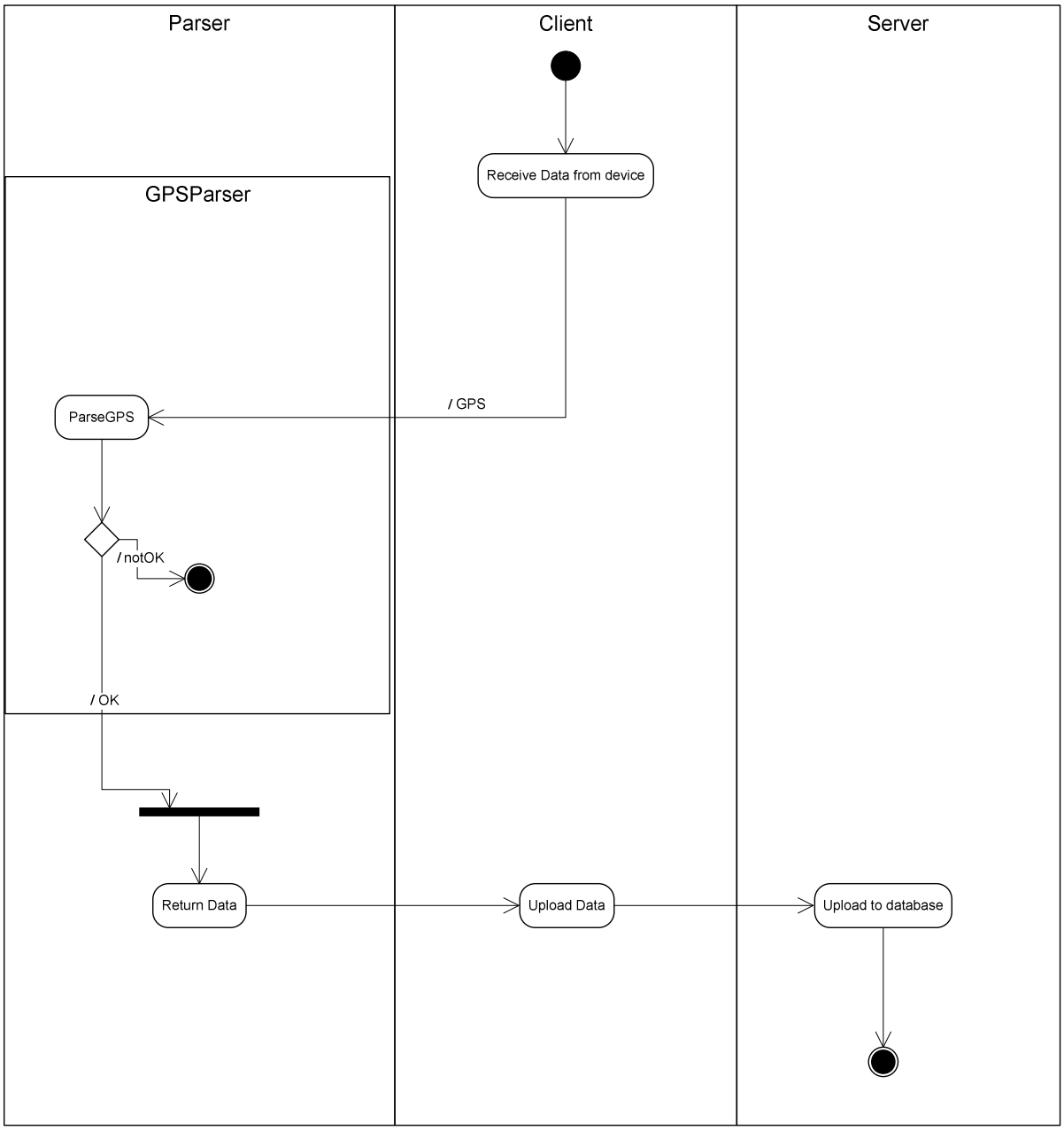


5 Activity diagrams

In this chapter you will find a couple of activity diagrams. These represent the data flow during the most complex activities of the system. In our system's case, this is the uploading of data through the client and through the web-interface.

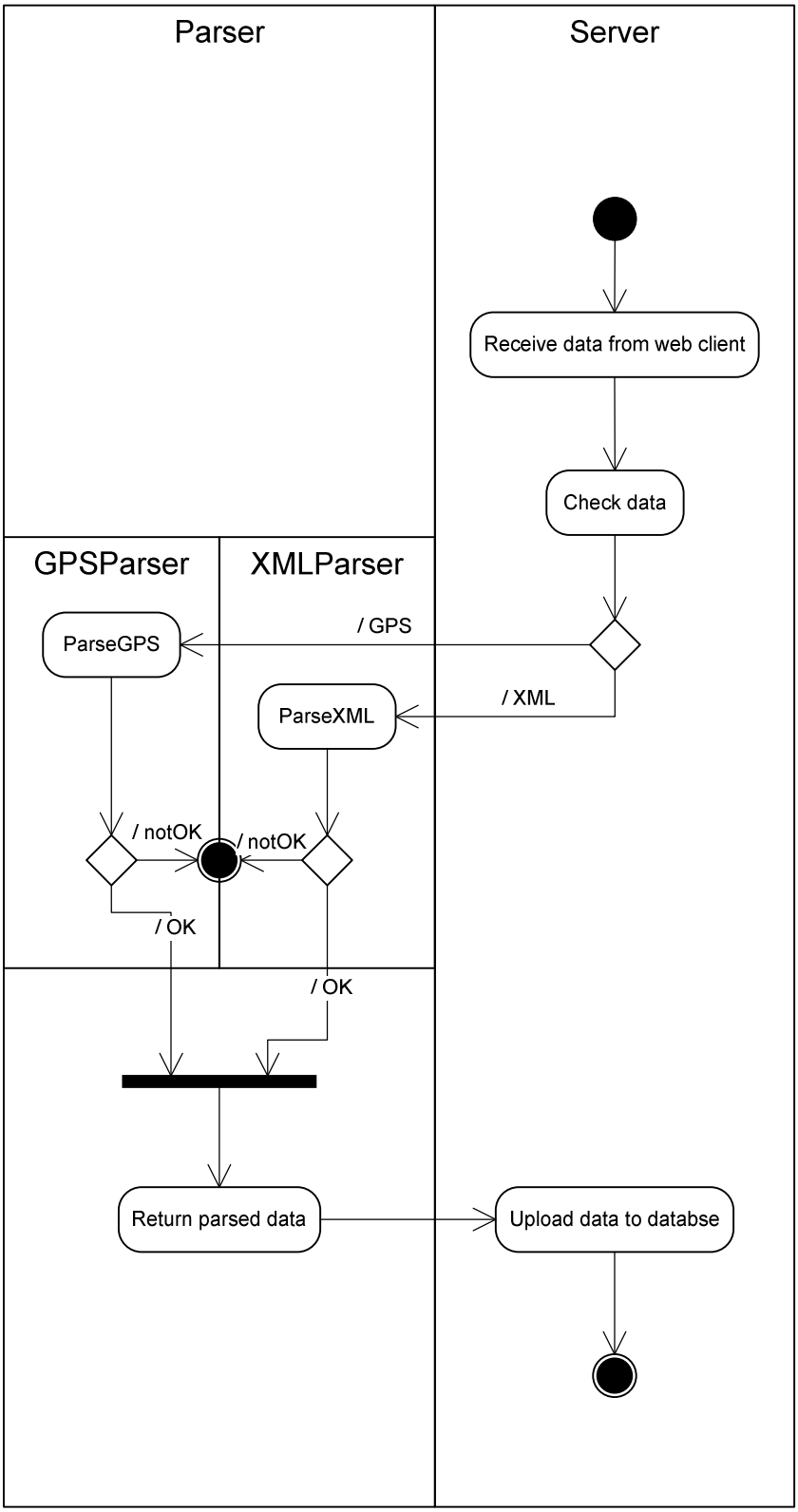
5.1 Uploading data through the client

Represented is the data flow through several classes when data is uploaded from a device using the client program.



5.2 Uploading data through the web-interface

In this activity diagram the data flow through several classes is represented when a user uploads data through the web-interface.



Appendix B
Prior research

Prior Research

Kristian Slabbekoorn, Gosse Bouma, Peter de Klerk

Version: 1.0

Status: final

Contents

1 OpenID

OpenID is an identity service which helps the user identify himself to various websites.
Users no longer have to use the traditional way of identifying themselves. In the traditional way users had to come up with a user name and a password, or use their email address. This information can be privacy sensitive and can be easily forgotten.
When using OpenID, you only have to register at one so called Identity provider. Users can choose the Identity Provider they trust to provide them an identity. This identity can then be used at every other website that supports OpenID. This can be a downside as well, using one OpenID provider gives this provider a lot of information of the serving habits of the user. In other words, the user's privacy is at stake here. If users don't trust there OpenID provider anymore, they can easily switch providers with their own login-in details. About ten thousand websites currently offer OpenID log-in to their users.

1.1 The owner

OpenID is an open-source project. This means it is not owned by anyone and is free to use by anyone. Also one doesn't have to register at an organization to use the software. Although the fact that OpenId is decentralized seems like an advantage, in practice the user still has to register at a centralized OpenID provider, he doesn't control.

1.2 How it works

While nothing in the protocol requires JavaScript or modern browsers, the authentication scheme works with "AJAX"-style setups. This means the user doesn't have to leave the website to prove his identity to the website.
OpenID Authentication uses only standard HTTP(S) requests and responses, so it does not require any special capabilities of the User-Agent or other client software. OpenID is not tied to the use of cookies or any other specific mechanism of OpenID Provider session management. Extensions to User-Agents can simplify the end user interaction, though are not required to utilize the protocol [1].

1.3 Using OpenID

It's quite easy for a user to use his own website as OpenID provider. There are several free programs on the web that facilitate this. Also it's not hard to be a OpenID provider yourself, because there are several recipes on the web which show step by step how to configure a web server to work with OpenID.

2 Fuzzy logic

Experts usually rely on common sense when they solve problems. They also use vague and ambiguous terms. However when this knowledge needs to be translated into a rule base, it's hard to represent these vague and ambiguous terms.

Fuzzy logic is the theory of fuzzy sets, sets that calibrate vagueness. Fuzzy logic is based on the idea that all things admit of degrees. Examples like: very fast, precise and short have a sliding scale that makes it impossible to distinguish members of a class from non-members.

2.1 Traditional Boolean logic

Traditional Boolean logic uses sharp distinctions, it always has a boundary. You are either a member of the set (1) or not a member (0). When you try to divide tall and small people for instance, one can say that a person is tall when he/she is taller than 180 cm. This makes a person of 180 cm tall (a member), but a person of 179 cm not (not a member); although their height only differs 1 cm.

2.2 Fuzzy set theory

The basic idea of fuzzy set theory is that an element belongs to a fuzzy set with a certain degree of membership. Thus, a proposition is not either true or false, but maybe partly true or partly false to any degree. The tallness example now shows a different result, when a person is 180 cm tall, it's membership's degree to the tallness set is 0.82. For the person of 179 cm, this is 0.80. As this example shows, there is no clear boundary if you belong to a set or not (the classic 0 or 1).

2.3 Fuzzy reasoning

Fuzzy logic is used in fuzzy reasoning. With fuzzy reasoning one can make a database with fuzzy rules. These rules are usually formed in IF-THEN statements. An example rule using our earlier height example:

IF male IS true AND height >= 1.8 THEN is_tall IS true; is_short IS fals

In traditional Boolean logic the rule would look like this. But by using fuzzy logic the rule doesn't make this sharp distinction:

IF height <= medium male THEN is_short IS agree somewhat

IF height >= medium male THEN is_tall IS agree somewhat

Using the fuzzy rules, the area around medium male is now called somewhat. This area makes sure values near a boundary are designated to their set(s) properly.

Another example, when speaking about age of people, the context can be $w = \langle 0, 45, 100 \rangle$ where people below 45 years are surely young, those around 45 are middle aged, and those over 100 are surely old. Ages below 45 are young in various degrees, those around 45 are middle aged in various degrees, and those over 45 are old in various degrees.

2.4 Hedges

In the previous examples a lot of non-numeric terms were used. Terms like medium, somewhat etc. are examples of hedges[3]. Hedges are used to weaken or strengthen the impact of statements. For example very, is a hedge to strengthen the impact of a statement. Because "very tall" is stronger than "tall", a new group can be created. The graph in figure 1 shows a red line which represents the "very tall"-set. Take, for instance, a person of 185 cm. He is a member of the "tall"-set, but not of the "very tall"-set, because 180 cm doesn't pass the red "very tall" boundary . A person of 190 is a member of the "very tall"-set, because 190 cm does pass the red boundary.

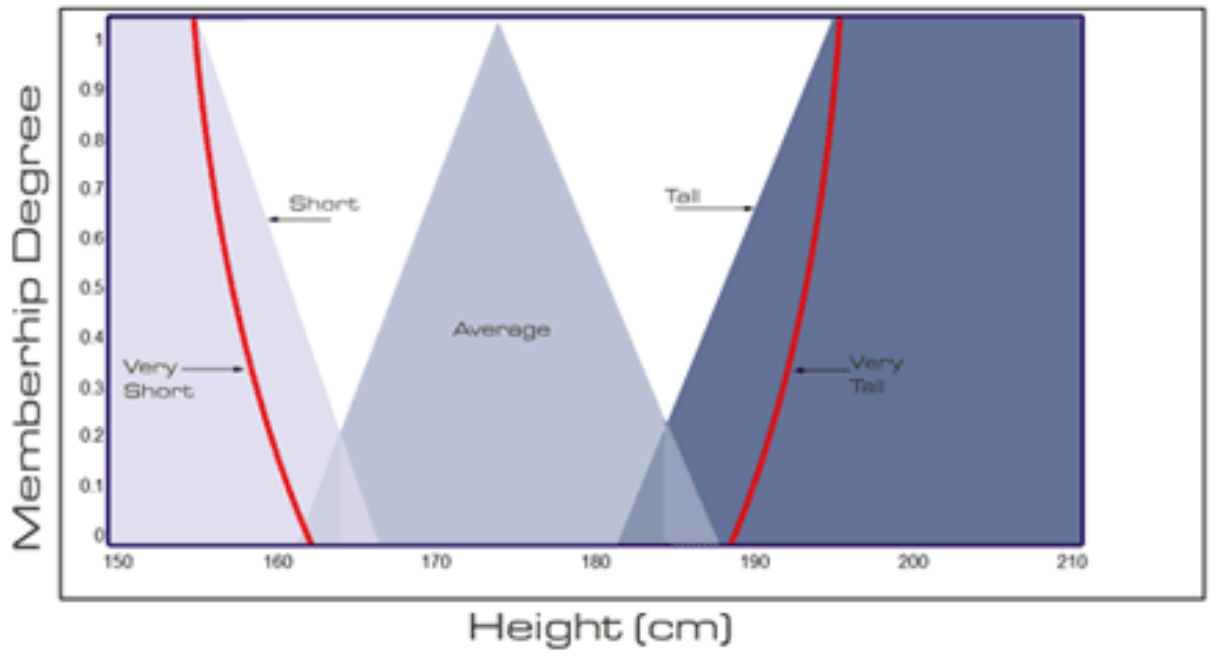
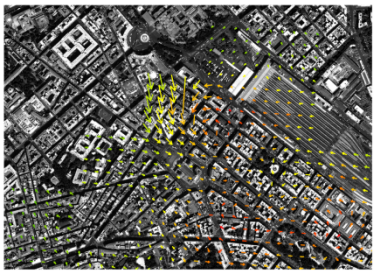


Figure 2.1Very tall example

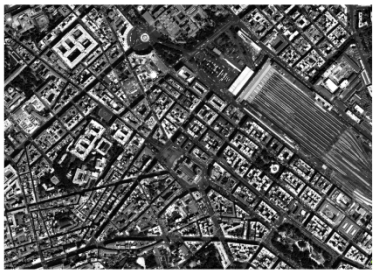
3 Real Time Rome

Real time Rome is a MIT project in order to display people movement in Rome. MIT's basis were the maps provided by Google via their Google Earth software. Because almost everyone has a cell phone these days, MIT used data provided by telephone companies to visualize people densities in Rome. Bus and Taxi information were also used to compute the efficiency of the public transport system in Rome.

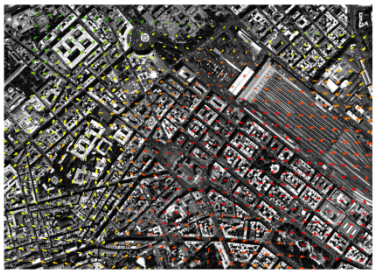
The images made by the software look like this:



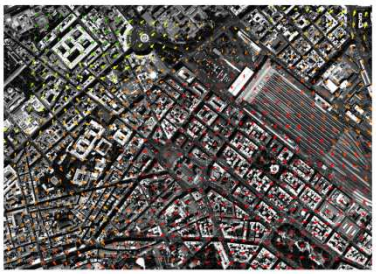
00 - 00 AM



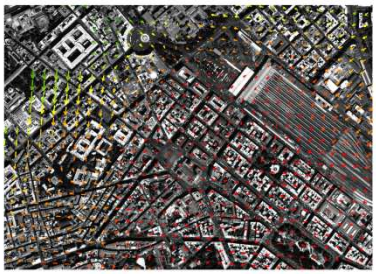
04 - 00 AM



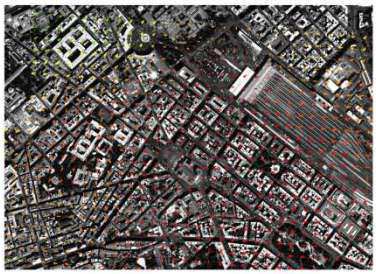
08 - 00 AM



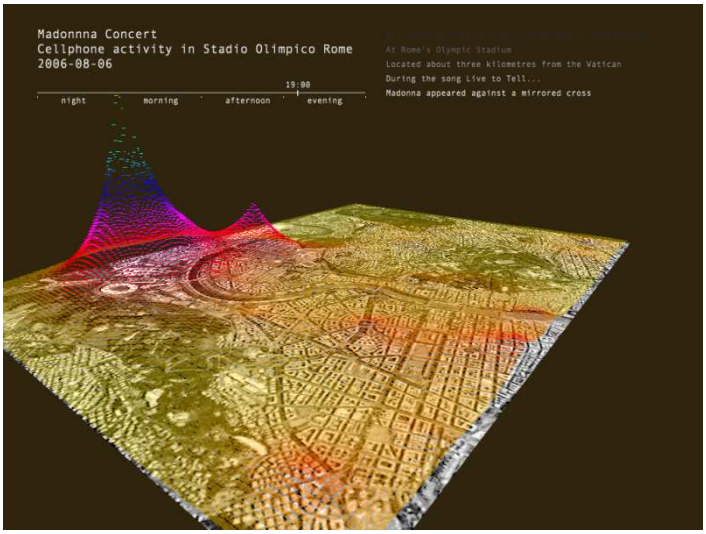
12 - 00 PM



04 - 00 PM



20 - 00 PM



These real-time maps show how neighborhoods are used in the course of a day, how the distribution of buses and taxis correlates with densities of people, how goods and services are distributed in the city, or how different social groups, such as tourists and residents, inhabit the city. With the resulting visualizations users can interpret and react to the shifting urban environment[4].

4 PlaceEngine

PlaceEngine is a core technology that enables a device equipped with Wi-Fi such as a laptop PC or smart phone to determine its current location. It can be used in conjunction with web sites that provide local area information to gain easy access to nearby services. An API is provided on the website, making it possible to use the technology in our own application. You must have the PlaceEngine Client running on your device in order to interface a local application with PlaceEngine. You can communicate with the PlaceEngine Client via HTTP. The following two steps are necessary to implement the "Get Location" function [5].

- Retrieve Wi-Fi information from the PlaceEngine Client (http://localhost:5448)
- Send the retrieved Wi-Fi information to the PlaceEngine Server (http://www.placeengine.com)

In order to access the PlaceEngine Client and PlaceEngine Server from your location application you must obtain an Application Key from the PlaceEngine site (http://www.placeengine.com/appk/indexe). You can then connect to the server using this key, and read the information from it in the following way:

```
HttpURLConnection urlc = (HttpURLConnection)url.openConnection();
urlc.connect();
BufferedReader reader = new BufferedReader(new
InputStreamReader(urlc.getInputStream(), "UTF-8"));
```

You can then read the information and process it.

Applications can communicate with the PlaceEngine Client by issuing HTTP GET requests to http://localhost:5448.

Using the LocalDB functionality, it is possible to find out your location without connection to the server. However, it is not possible to find out about the address, floor information, estimation accuracy and the number of access points of the location this way.

Applications can communicate with the PlaceEngine Server by issuing HTTP GET requests to http://www.placeengine.com/api/.

5 Mashup

A mashup is a web application that combines data from multiple sources into one, integrated tool [6]. Content is usually sourced from a third party API. The architecture of mashup web applications consists of 3 parts: the content provider, the mashup site and the user's web browser.

Several types of mashups are: consumer mashups, data mashups and business mashups. While the underlying technology is the same, there is some distinction in how this technology is used. Consumer mashups combine different forms of media from multiple sources and combine them into one, to provide a unique service. Data mashups combine similar types of media and data from multiple sources into a single graphical representation. Business mashups focus on data aggregation into a single presentation.

There are some key differences between mashups and portals. A portal approaches aggregation by splitting role of web server into two phases - markup generation and aggregation of markup fragments. A mashup uses APIs provided by different content sites to aggregate and reuse the contents in another way. A portal presents all content side-by-side, while in a mashup, individual content can be combined in any manner, resulting in arbitrarily structured hybrid content.

6 Social Network Services

For the assignment, the information that is being presented, needs to be translated into real terms and collected and presented on a social network service. For example, when information such as "person p - location x,y - time 1300" arrives at the system, it needs to update a social network page to show that Peter is in the cantina during lunch time. The data will be presented from another source, most likely an engine which translates the raw GPS and identification data into data the social network service can use. If all required information is present in this data, translating it to real (readable) terms is easy. However, the social networking part is more complicated. In order to get an overview of what social network services actually are, we will study them more carefully.

Social network services provide just that, a social network for its users. Users can communicate with each other on different levels. Social networks have become very popular recently, with some of the most popular sites on the web being social network services. With millions of people using them regularly, it seems social networks from an enduring part of everyday life.

Usually, a social network service provides some kind of e-mail and/or instant messaging service. With most social network services users have a self-description page where they can introduce themselves. Also users can keep a log of what they have been doing, where they've gone and so on. This will come in useful for our assignment; we hope to improve the 'where' part by automating the generation of a blog and automatically changing the users' location. However, this can not be done just like that, social networks services are mostly heavily protected and outside applications cannot access them easily. We need to find a way to access and submit data on social network services. Luckily, we found that Yokoyama et al. [7] already studied a similar subject, and came up with a solution. First though, we will briefly study why these problems exist and what kind of solution is needed.

The biggest pitfall developers of these social network services face is the privacy of the users. It is important that not all personal information of the users provided on a social network ends up all over the internet. Yet at the same time, users should be able to provide enough information so that their friends can communicate with them. Therefore, a strongly secured system is necessary. Another problem developers have to tackle is understanding what its users want. Which people do they want to find and communicate with over the social network? The information that can be provided by the users must be well thought out. If the wrong information ends up on the network, ill-intended people can exploit it (for example for spamming etc.) [7] The information must be what is called human-orientated.

If developers want to create an application which uses social network data, they will have to find a way to extract this human-oriented information. Yokoyama et al. provide a solution in the form of an idea for an API. However, this API does not support submitting data to an SNS, but this will be needed for the implementation of our assignment. The proposed API does extract a lot of data from the social network. Not all of this data might be relevant to our system, however with so much data available new ideas can spawn. For example, the API provides access to user's communities and it's members. This would provide possibilities for finding people with similar interest, beyond your own friends, that travel close to you. Perhaps there are even more way of interaction between users of the social network... Over the course of this assignment we hope to come up with more creative ideas.

We have to keep in mind though, that the API Yokoyama et al. propose does not actually exist, we have to find an alternative.

7 Open Social

A more relevant API for the system we will be building is the Open Social API. This is the most commonly used and state-of-the-art technology used by social networks nowadays. It was developed by Google in corporation with MySpace and some other social network services.

We will implement our system so that it can interact with applications adhering to the Open Social API (so called Open Social containers). Since most of the large social networks on the web (such as MySpace, Orkut, Mixi and others) are indeed Open Social containers, our system will be compatible with most of them. This chapter will contain a report about our research of the Open Social technology and API. When our research is finished, we will be better prepared for developing an application using the Open Social API.

According to the Open Social homepage [8] it can be defined as "a common API for social networks across multiple websites". It supports standard JavaScript and HTML. Developers using the Open Social standard are able to implement applications that can access social network's friends lists and update feeds.
This means that when someone uses an application on one social network, say Orkut, it will access only that person's Orkut friends. When he uses that same application at MySpace, it will only access his MySpace friends.

The most important data in an Open Social container are the viewer (logged in user), the owner of the application, and the viewer's or owner's friends. These data are easily fetched using a DataRequest and sending it to the social network.
Activities also play an important role in social networks. Open Social provides access to activities in the form of an activity stream. An activity stream is basically a feed which lists actions a user takes at certain times. An activity can be anything from changing the user's state to uploading a new picture. For our assignment, these are the most important data the system needs access to. The exact and complete list of functionalities of the API is well documented in the Open Social Developers Guide, so we will not go into further detail.

8 References

[1] = “OpenID authentication final 2.0”, http://openid.net/specs/openid-authentication-2_0.html
[2] = 'Artificial Intelligence', second edition, by Michael Negnevitsky, Addison Wesley, 2004, p30-33
[3] =Fuzzy Math, part 1, the theory. http://blog.peltarion.com/2006/10/25/fuzzy-math-part-1-the-theory/
[4] = Real time rome official website, http://senseable.mit.edu/realtimerome/
[5] “PlaceEngine Technical Information Page”, http://www.placeengine.com/doce
[6] ”Mashups: The new breed of Web app”, http://www.ibm.com/developerworks/xml/library/x-mashups.html
[7] “A Generic API for Retrieving Human-oriented Information from Social Network Services” by Teruaki Yokoyama, Shigeru Kashihara, Takeshi Okuda, Youki Kadobayashi, and Suguru Yamaguchi http://ieeexplore.ieee.org/iel5/4090056/4090057/04090098.pdf
[8] “Open Social homepage”, http://www.opensocial.org/