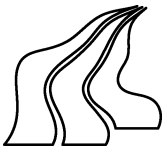
Route Guide Providing Route Guidance for Nomadic Users

Peter Mølgard Vinther *Henrik Olesen*
petermv@cs.auc.dk *nunu@cs.auc.dk*

Institute of Computer Science
Aalborg University

Aalborg University
Institute of Electronics
Frederik Bajersvej 7, 9220 Aalborg East
Computer Science Department



Route Guide Providing Route Guidance for Nomadic Users

PROJECT PERIOD:
1. February - 25. May 2001

PROJEKT GROUP:
E3-213a

PROJECT MEMBERS:

Peter Vinther

Henrik Olesen

SUPERVISOR:
Nectaria Tryfona

COPIES: 6

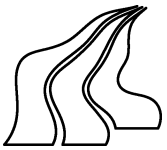
PAGES: 65

APPENDIX: 16

ABSTRACT:

The purpose of this project is to develop a Route Guide application prototype, providing route guidance for nomadic users. The idea is that nomadic users interact with the Route Guide from mobile - or desktop computers in order to create, modify or delete information stored in a User Profile. Information stored in the User Profile is used by the Route Guide to answer requests for route guidance. The actual route planning leading to route guidance is implemented by the use of ArcView, providing the road network. route guidance is provided as HTML, WML or Voice depending of the platform used by the nomadic user.

Aalborg Universitet
Institutet for Elektroniske Systemer
Frederik Bajersvej 7, 9220 Aalborg Øst
Computer Science Afdelingen



Rute Guide der tilbyder Rute Vejledning for Mobile Bruger

PROJEKT PERIODE:
1. februar - 25. maj 2001

PROJEKT GRUPPE:
E3-213a

MEDLEMMER:

Peter Vinther

Henrik Olesen

VEJLEDER:
Nectaria Tryfona

KOPIER: 6

SIDER: 65

APPENDIX: 16

ABSTRAKT:

Formålet med projektet er at udvikle en Rute Guide applikations prototype, som tilbyder rute vejledning til mobile bruger. Ideen er at mobile bruger skal kunne interagere med Rute Guiden fra mobile - eller desktop computere for at oprette, slette eller modificere information gemt i en Bruger Profil. Rute Guiden anvender den gemte information i Bruger Profilen til at besvare forespørgelser på rute vejledning. Den faktiske rute planlægning der resultere i rute vejledningen er implementeret vha. ArcView som stiller det nødvendinge vej-netværk tilrådighed. rute vejledning returneres som HTML, WML eller Tale afhængig af hvilken platform den pågældende bruger anvender.

CONTENTS

CHAPTER 1

Introduction

During the 90's the Internet have had a tremendous impact on the way we acquire information. In the beginning the Internet was mainly used by universities and large companies to exchange or share knowledge. Nowadays the Internet is used by many to access information about different topics. Hyper Text Markup Language (HTML) is the cornerstone of the Internet when it comes to providing human readable information. Technologies as Servlets, Java, Active Server Page (ASP), Hypertext PreProcessor (PHP) etc. have become important tools used to enrich interaction and the dynamic nature of the Internet.

Due to the prevalence of these technologies and recent advances and deployment within the area of mobile communication it is possible to develop application for the Internet providing services for users of mobile devices.

At the moment different applications have been developed for the Internet. Applications as Krak and MapQuest[1] provides applications that helps planning a route. The purpose of this project is to develop a Route Guide application prototype, providing route guidance for nomadic users.

To limit the scope of this project the next section points out the primary interest of the project.

[1]Krak: http://www.krak.dk, MapQuest: http://www.mapquest.com

1.1 Goal

Goal: The goal of the project is to develop a Route Guide providing route guidance for nomadic users.

This goal consist of the following sub goals:

- It is the task of the Route Guide to perform route planning based upon input as start- and end location, additional addresses and services required by the nomadic user as being a part of the route.
- The Route Guide should support the nomadic user when trying to follow route. This should be done by providing continues route guidance, informing the nomadic user to follow a route.
- The Route Guide should provide the nomadic user with persistent storage of personal information enabling the Route Guide to provide customized route guidance. Access to the persistent storage of personal information should be provided for different platforms, allowing the nomadic user to access it in different situations.

We believe it will enhance the usefulness of the Route Guide if the route guidance is presented as text, voice and maps, or different combinations of the three. Therefore, we will explore how text, voice and maps can be used to provide services within the concept of a Route Guide.

1.2 Outline

In chapter 2 a system architecture showing the parts of the hole system is presented. Then an analysis of the Route Guide Application is carried out, defining three components; User Profile[2] Management (UPM), User Request and Response Management (URRM) and Route Planning (RP). The analysis of the Route Guide leads to the design of the Route Guide. This is done by the help of a class diagram, showing relations between classes that are important to the Route Guide.

In chapter 3 the UPM component introduced in chapter 2 is analyzed in more detail. First the idea of UPM and User Profile (UP) is introduced, leading to an analysis of the requirements such an idea impose on the UPM and UP components. The result is a conceptual model (Entity Relational model) of the UP being transformed into an Extensible Markup Language

[2]For now, thing of a user profile (UP) as being a place where personal information such as address, user name, user id etc. are persistently stored

(XML) database schema.

The RP component is analyzed in chapter 4, where the geocoding process is presented together with solutions of how to calculate the optimal route (Route Planning) in ArcView when facilities/services specified by the nomadic user is an important part of the requirements.

Finally in chapter 5 implementation topics related to the three components described during chapter 2, 3 and 4 are presented. Chapter 5 deals with retrieval or information from the UP, communication between URRM and RP and transformation of internal representation of route guidance to HTML or WML.

The report ends with a conclusion (chapter 6) on the presented work in this report and give direction towards future work.

CHAPTER 2

Route Guide Application

During the introduction the idea of a Route Guide was introduced. This chapter gives a more detailed perspective on the Route Guide application by defining three components; User Profile Management (UPM), User Request and Response Management (URRM) and Route Planning (RP). Then input and output to the Route Guide application is introduced leading to the design of the Route Guide. But first a system architecture showing the parts of the whole system is presented.

2.1 System Architecture Analysis

In order to develop a Route Guide an understanding of the surrounding system is needed, the topic of this section is to develop a system architecture showing Route Guide interaction with central components of the overall system.

Moving from left to right on figure 2.1 we start by explaining the Client Device.

2.1.1 Client Device

The Client Device at figure 2.1 is used by the nomadic user[1] to interact with the Route Guide running at the Server Platform.

[1]Nomadic user vs User: Nomadic user is a user that is expected to move from time to time, whereas user is a more general term.

For client side navigation two standards and one upon coming, are available:

- Hyper Text Markup Language (HTML). Some of the benefits of HTML over Wireless Markup Language (WML) is the layout features and the amount of support. Further more, HTML 4 supports multimedia options, scripting languages, style sheets, printing facilities, and documents that are more accessible to users with disabilities. HTML 4 also takes great strides towards the internationalization of documents, with the goal of making the Web truly World Wide. HTML 4 is an Standard Generalized Markup Language (SGML) application conforming to International Standard from International Stardards Organization (ISO) 8879[4]

- The benefit of using WML is that more devices like Wireless Application Protocol (WAP) phones support this language. WML is a markup language based on the Extensible Markup Language (XML) and was developed for specifying content and user interface for narrow band devices such as cellular phones. Some of the benefit of WML is that it is designed for small display screens with low resolution. For example, most mobile phones can only display a few lines of text, and each line can contain only 8-12 characters. By the use of menus the WML is specially designed for small devices like mobile phones that typically only have numeric keypad and a few additional function specific keys[5].

- Voice Extensible Markup Language (VoiceXML). VoiceXML is a Web-based markup language for representing human-computer dialogs, just

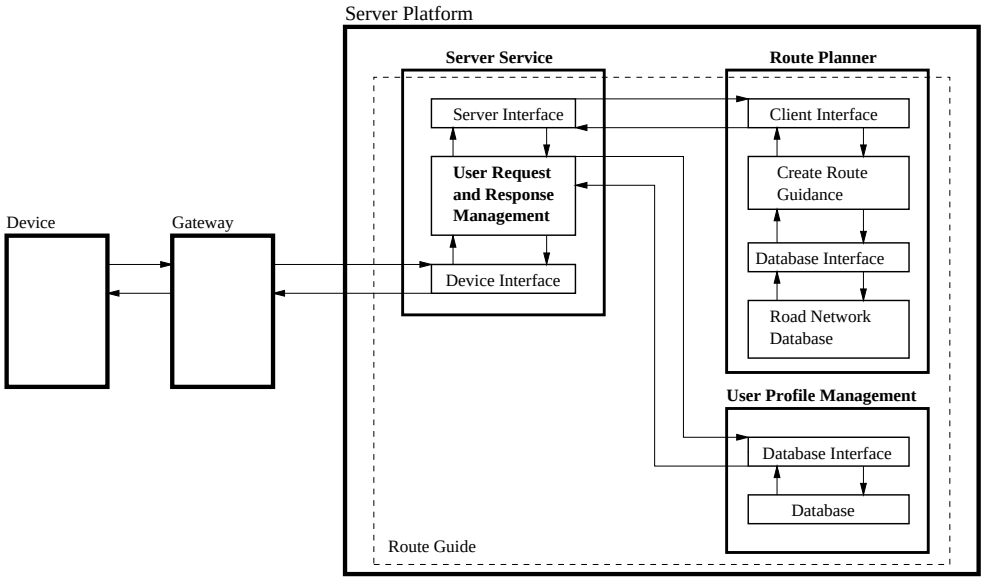


Figure 2.1: System Architecture

> like HTML. But while HTML assumes a graphical web browser, with display, keyboard, and mouse, VoiceXML assumes a voice browser with audio output (computer-synthesized and/or recorded), and audio input (voice and/or keypad tones). VoiceXML leverages the Internet for voice application development and delivery, greatly simplifying these difficult tasks[9].

As a result of the three markup languages the Route Guide should be able to provide route guidance as HTML, WML or VoiceXML based upon the Client Device platform. This feature can be provided by the use of XML documents being translated into appropiate format by the use of XSL. In this way the Route Guide is compatible with more client devices without using a format restricted to the intersection of the formats used of the users.

Another important issue, according to our perception of a Route Guide, is the ability to provide response as Voice, due to the fact that it is more traffic safe to get instructions as voice, instead of reading them on a small screen while driving.

The Client Device part of the architecture is one of the issues that we deal with during the project and for the prototype implementation we have chosen to use a WML emulator as Client Device.

2.1.2 Gateway

The architecture at figure 2.1 shows the Gateway between the Client Device and the Server Platform.

The Gateway connects the wireless network with the wired network. When the Gateway receives a Wireless Application Protocol (WAP) request it sends the request over the Internet as a HyperText transfer Protocol (HTTP) or Secure HyperText transfer Protocol (HTTPS) request.

With the architecture shown at figure 2.1, being a base for a world wide application it would be naturally to distribute the Gateway to the network operator. But this also means that only the customers of that operator can access the service. As an effect of this, co-operation agreements with more operators might be necessary to reach all the wanted customers. When distributing the Gateway it is also the task of the operator to collect billing information for WAP services and manage the billing of other services[6].

Regarding to the prototype implementation we have chosen not to work with the Gateway issue due to the fact that it is a service that is provided by others and not an area that contribute much to the goal of this project

(See section 1.1). Therefore the prototype implementation will not include the Gateway.

2.1.3 Server Platform

The Server Platform provides the execution environment for User Request and Response Management (URRM), the Route Planner (RP) and the User Profile Management (UPM) components. New Technology (NT) Workstation 4.0 with service pack have been chosen as Server Platform, due to it easy of use and administration and because the Nokia Activ Server runs on NT.

Server Service The Server service which is a part of NT allows people to add additional server functionality to NT. We have chosen to add the Nokia Activ Server as a service running on NT. The Nokia Activ Server is able to host servlets that can provide dynamic content and handle interaction with clients.

Device Interface When developing online services like a Route Guide to be used through a web browser or a WAP terminal, it usually requires server side programming. Different technologies are nowadays used to handle client requests from mobile devices and provide dynamic content. The server side has to provide a mobile device interface to receive a HTTP request from the Gateway. The HTTP request has to be unmarshalled and handed over for further computation.

Technologies such as Common Gateway Interface (CGI) scripts, Active Server Page (ASP) and Java Applets are widely used in the development of dynamic services. Nowadays, however Java Servlets are becoming widely used as an efficient platform and server independent technology used to provide dynamic content. The main benefits of using Java Servlets for server-side programming are: persistence, performance and portability [3].

The Route Guide is implemented as servlets, using the servlet technology (Servlet Application Programmer Interface (Servlet API)) as interface between the servlets and the Gateway. Actually due to the fact the Gateway is not included in the prototype implementation, the interface is between the Route Guide and the WML emulator and is implementet as a servlet.

Server - and Client Interface The Server - and Client interfaces between the URRM and the RP provides functionality that allows them to communicate with each other. In the setup of figure 2.1 the RP should be thought of as a "*server*" providing functionalities as Create Route Guidance

(CRG) to the URRM which should be thought of as being the ″client″.

The idea is to implement the URRM and UPM components from 2.2 as servlets running on top of the Nokia Activ Server (Server Service). The RP will be implemented by the use of ArcView, do to the ease of using data and functionality already established in ArcView. In order to allow the URRM to communicate with ArcView the JavaDDE interface[8] will be used to implement the Server Interface. JavaDDE provides functionalities that allow the URRM to connect and disconnect to ArcViews DDE-Server. Besides connecting and disconnecting to the DDE-server, JavaDDE contains functionalities that allows the URRM to execute commands and pass parameters to the RP component.

Just as the Server Interface allows the URRM to communicate with ArcView, the Client Interface allows ArcView to respond to requests received from the URRM.

Route Planner Besides the Client interface described above, the RP consist of three components; CRG, Database Interface and Road Network Database.

Avenue[2] provides[7] functions that allows the programmer to access database data in order to retrieve valuable data, needed for the creation of route guidance. In order to create route guidance the CRG needs access to data about roads (street, houseNr, postal code etc.). This information is found in the Road Network Database being a part of the RP.

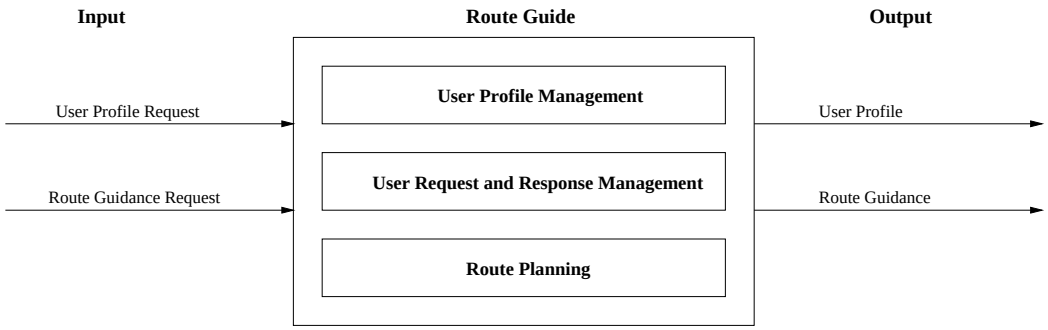


Figure 2.2: Route Guide

[2]Script language provided by ArcView

2.2 Route Guide Analysis

In the previous section the system architecture showing the Route Guide interaction with other parts of the whole system where presented. The following section is an analysis of the the Route Guide consisting of three components (see figure 2.2) identified during the analysis of the system architecture.

User Profile Management concerns management of personal information in the user profile enabling the Route Guide to provide customized route guidance. In order to do this the UPM component provides external functionalities allowing users to create, modify and delete personal information. Additional internal functionality enables retrieval and storage of the same personal information (see figure 2.3).

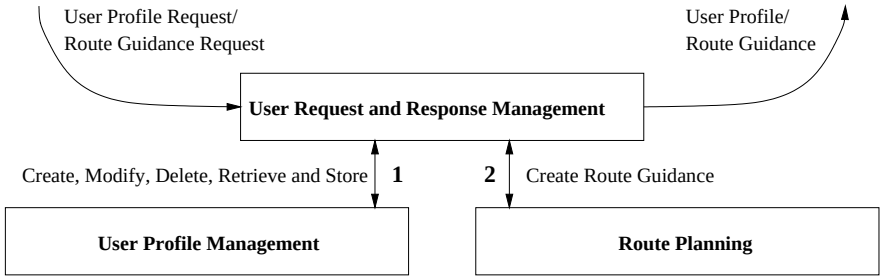


Figure 2.3: Service Hierarchy

User Request and Response Management provides functionality that allows the Route Guide to handle incoming requests (Input) from users and to answer requests by sending response back (Output). There exist a cooperation between the UPM and URRM components. The URRM component handles the Input/Output of the Route Guide and use functionality provided by UPM to access personal information (see figure 2.3, arrow labeled 1). Furthermore the URRM component contains functionality allowing cooperation with the Route Planner (see figure 2.3, arrow labeled 2).

Route Planning is based upon information the URRM have received directly from the user or personal information retrieved by the help of the UPM. According to this information it is the task of the RP to create route guidance (see figure 2.3, arrow labeled 2) and return this route guidance to the URRM. When the URRM receives the route guidance it translates the internal representation of route guidance to an external representation that applies to the platform of the user.

2.2.1 Route Guide Input and Output

The tasks of the three components (UPM, URRM and RP) have been established and it is time to turn to towards Input and Output to the Route Guide.

User Profile Request The first input type of figure 2.2 is a User Profile Request (UPR), this type of input refers to all requests from users regarding creation, modification and deletion (see figure 2.4, arrow $D.1, D.3, D.7$ and $D.11$) of personal information in the user profile.

By looking at figure 2.4 the following approaches regarding creation, modification and deletion can be found:

- **Create User Profile**, Start at the user side, follow arrow $D.1$ to the process Initial Request (1.1). This process returns the user profile interface ($D.2$), needed by the user before he can perform a creation, modification or deletion. Then upon choosing creation of a user profile ($D.3$), process 1.2 retrieves a user profile template from the user profile storage and returns the template ($D.4$) to the user. Upon receiving the template, the user enters all necessary information, which is returned ($D.5$) to process 1.3. Process 1.3 creates the user profile in the user profile storage and returns either a success or failure message ($D.6$) to the user.
- **Modify/Delete User Profile**, when the UPR is for a modification the approach is: $D.1, 1.1, D.2, D.7, 1.4, D.8, D.9, 1.5$ and $D.10$. The approach for a deletion is as follows: $D.1, 1.1, D.2, D.11, 1.4, D.8, D.12$, 1.6 and $D.13$.

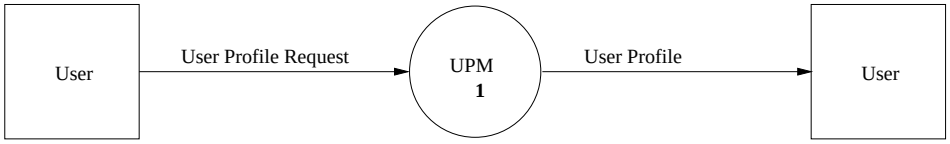
User Profile The User Profile (UP) arrow of figure 2.2 contains the following data flows of figure 2.4: $D.2, D.4, D.6, D.8, D.10$ and $D.13$, indicating different kinds of response to the user.

Route Guidance Request and Route Guidance The Route Guidance Request arrow of figure 2.2 refer to all request from users regarding route guidance. A similar data flow diagram exist as that of figure 2.4, by starting at the Route Guidance Request arrow, similar processes exist to handle such a request, leading to Route Guidance as output from the URRM component.

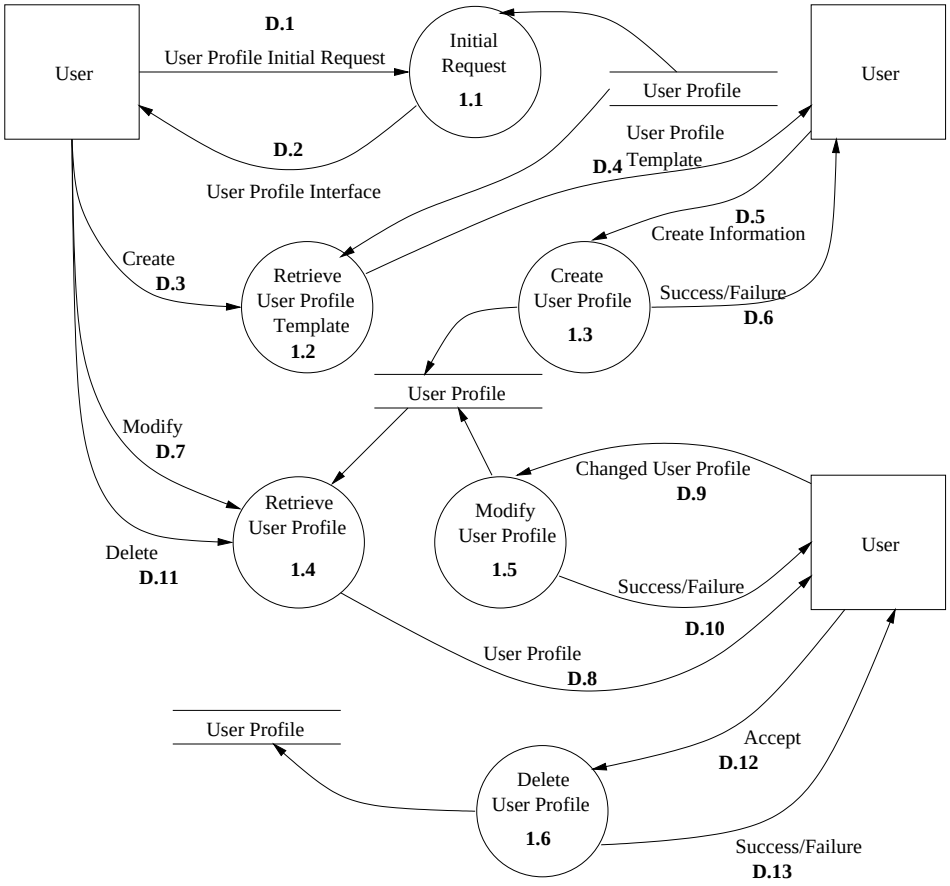
2.3 Route Guide Design

The design of the Route Guide is the topic of the folowing section. It is carried out by the help of a class diagram showing relations between classes important to the Route Guide.

HTTPServlet During the analysis it was found that the Route Guide should be implemented as a servlet. Therefore the class diagram of figure 2.5 consist of the Hyper Text Transfer Protocol (HTTP) Servlet package



(a) Process Level 1



(b) Process Level 2

Figure 2.4: User Profile Request Data Flow Diagram

(javax.servlet) from SUN Microsystems Inc. This package provides functionality that allows the Route Guide to accept client calls. In this case from the nomadic user. When a servlet receives a call, two objects are created: ServletRequest and ServletResponse, these object are used to communicate with the client. HTTP servlet provides classes to be sub-classed to create a HTTP servlet suitable for a Web site. The two subclasses Input and Output of figure 2.5 extends classes from the HTTP servlet package and overriding the doGet, doPost and Init functions of the HTTPServlet class in order to

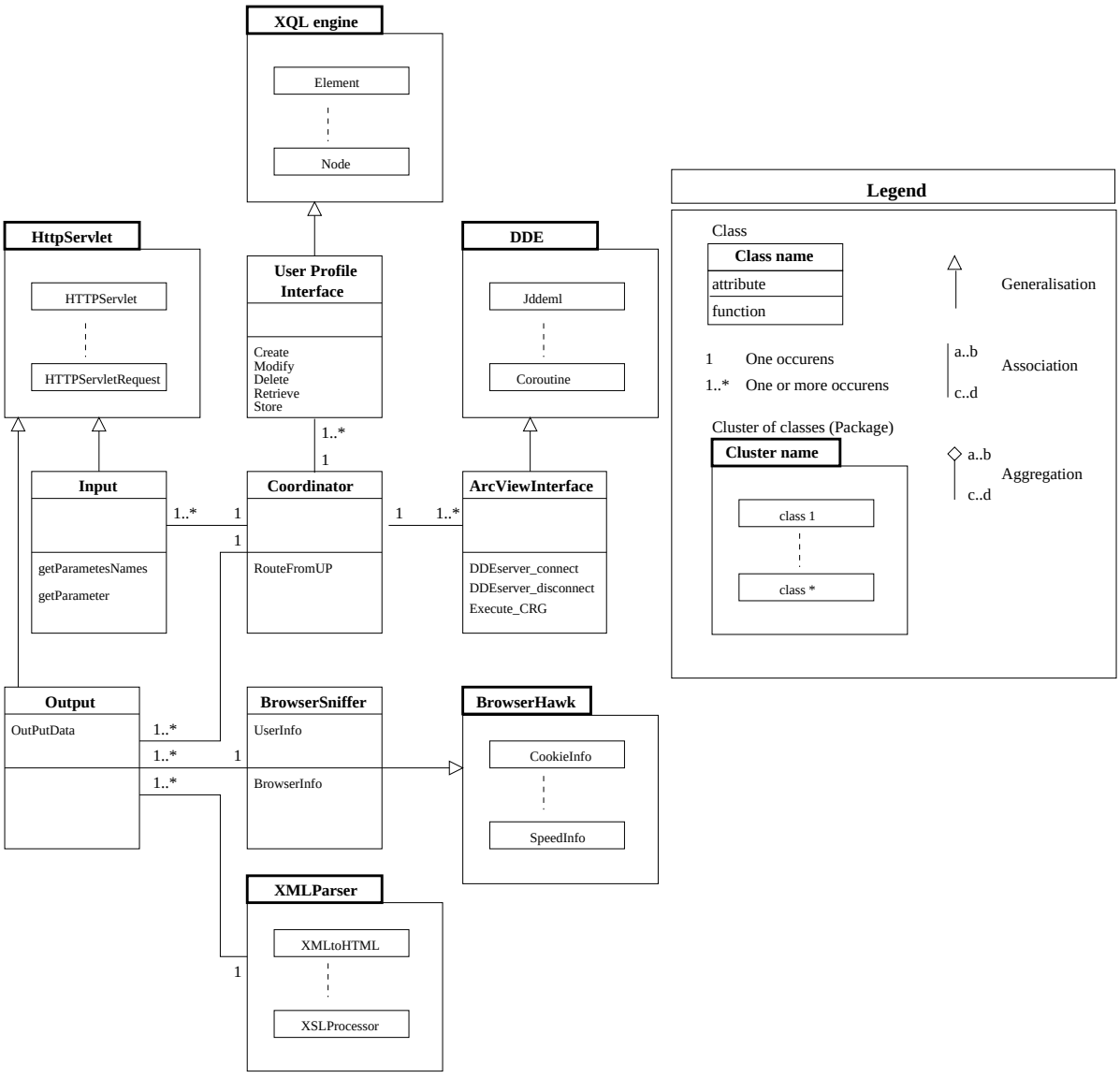


Figure 2.5: Route Guide Class Diagram

communicate with the client.

Input This class handles all Input from the nomadic user. The main task of this class is to receive parameters from the client in the form of a HTTP request. When a HTTP request is received the Input class uses the getParameterNames() and getParameter() functions to fetch parameters from the HTTP request message. These parameters are then passed on to the Coordinator class.

Coordinator The task of the Coordinator class is to, based upon parameters received from the Input class to decide weather the request is for one of the five functions the user profile interface (UPI) class provides. In the case of a request regarding UPM the Route Guide decides which of the five functions provided by the UPI class to use. If the request concerns execution of an already defined route, it is the task of the Coordinator class to fetch route information from the user profile by using the retrieve function provided by the UPI class. This information is then handed over to the RP or more precise to the CRG by using the ArcViewInterface class. When the Coordinator receives a planned route from the RP it provides this information to the Output class.

User Profile Interface The UPI class provides five functions: modification, deletion, retrieval and storing of information in the user profile and creation of a new user profile. The UPI class uses functionality provided by the XQL Engine package to access the UP. Each of the UPI functions are described in greater detail later.

Dynamic Data Exchange The Dynamic Data Exchange (DDE) package[3] consist of classes that provides functionality enabling the Coordinator to communicate with the RP. The most important functions provided by this package is connect, disconnect and execute which allows a DDE client to establish a connection to a DDE server, in our case the RP (see figure 2.1), in order to perform some actions by the use of the DDEPoke command and finally to close the connection again by using the disconnect function.

ArcViewInterface The ArcViewInterface class extends the DDE package by implementing exception handling and providing avenue scripts to activate the CRG being a part of the RP (see figure 2.1).

Output The Output class determines the target client platform (Browser specific testing), hence deciding weather the response to the nomadic user

[3]JavaDDE from Neva Object Technology Inc [8].

should be in the form of HTML, WML or Voice. The Output class uses functionality provided by the BrowserSniffer class to determine the target platform. Having decided the platform it is the task of the Output class to select the appropriate style-sheet and transform the internal representation of route guidance into HTML, WML or even Voice. The transformation is done by the help of functionality provided by the XML parser package. Furthermore, it is the task of the Output class to provide continuous route guidance. This means that the response should be divided into smaller parts, where each part represent e.g. one street. To accomplish this task, the Output class should be aware of the location of the user. The idea is that the mobile devices used by nomadic users are able to provide upon request there GPS location to the Route Guide. It is the task of the Output class to request the GPS location from the mobile device when providing continuous route guidance. Research about tracking locations of mobile devices ([18] and [19]) have already been carried out, and therefore not considered a part of the report.

2.3.1 Component and Interaction Diagram

In section 2.3 classes of the Route Guide and the relations between them were presented by the class diagram. In this section an Component and Interaction diagram (see figure 2.6) is used to show the required interaction between classes within a component and between class from different components[4], in order to provide route guideance.

At the Client Device a request is made by the user to the Uniform Resource Locater (URL) address of the Route Guide. The request from the user is received by the Input class (#1) and handed over to the Coordinator (#2), coordinating the hole process of providing route guidance.

To access the UP the Coordinator makes use of the UPI class (#3). The UPI class provides functionality necessary to retrieve information from the user profile (#4).

The user also has the option to request route guidance. To accomplish this the Coordinator use the ArcViewInterface (#5). The ArcViewInterface is used to establish a connection to the RP (#6). The RP contains functionality for route planning (CRG) (#6a) and the road network representation (#6b) used by CRG.

When the Coordinator receives a result from the UPI or ArcViewInterface the Coordinator passes the result to the Output class (#7).

[4]Recall that these components are UPM, URRM and RP

The Output class is responsible of the presentation of the response to the user. The presentation depends on the type of task performed by the Route Guide. If the needs of the user is to get information from the UP it should be returned as readable information either as WML or HTML.

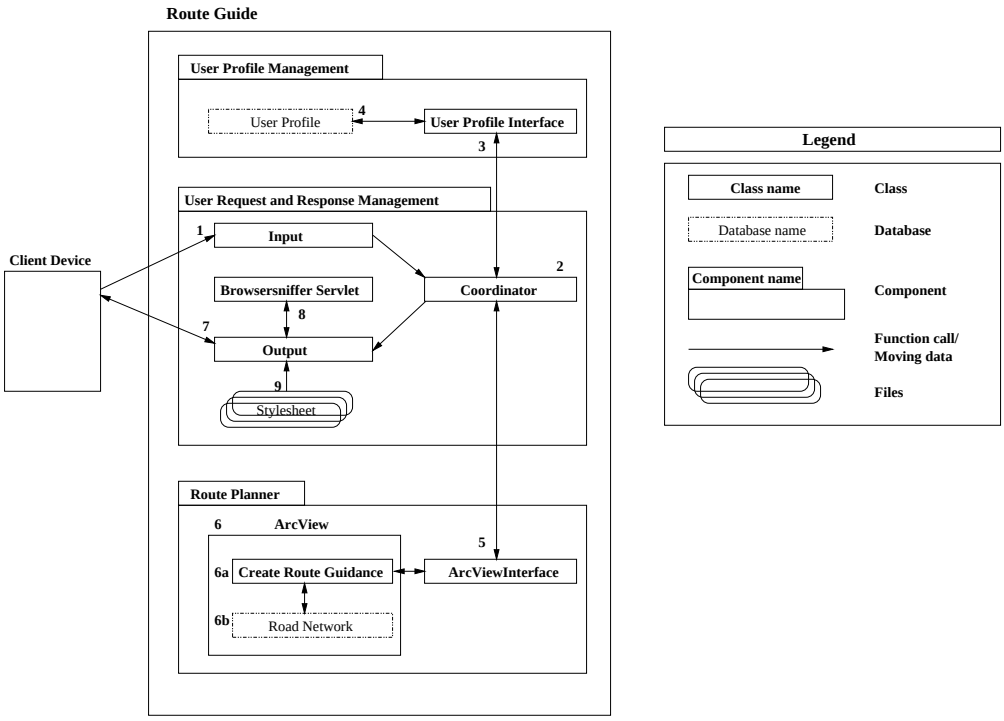


Figure 2.6: Component and Interaction Diagram

In order to communicate with users using different browsers and text format, the format of the current user should be detected. This is accomplish by the BrowserSniffer (#8). The BrowserSniffer detects the platform used on the client device. This knowledge is used by the Output class to decide the response format (HTML, WML or Voice). In order to respond in the format used by the user the XML document containing the information (e.g route guidance) meant for the user should be converted. This conversion is done by the XML parser. The input to the XML parser is the XML document containing the response to the user and a reference to a stylesheet. The Extensible Stylesheet Language (XSL) is used to express the intention about how the structured content should be presented, that is, how the layout of the source content should be styled. There are two aspects of this presentation process. First, constructing a result tree from the XML source tree

this is done by the use of Extensible Stylesheet Language Transformation (XSLT). Second, interpreting the result tree to produce formatted results suitable for presentation. The first aspect is called tree transformation because it transform the struktur of the document and the second is called formatting.

If voice is used as response format it is the task of the Output class to manage the discourse between the Route Guide and the user. By providing suitable voice subparts of the hole route guidance and interpretation of next commands issued by the user as indication of the need for the next voice subpart the Output class is able to provide route guidance as voice.

2.4 Chapter Summary

First a system architecture (section 2.1) showing interaction with other parts of the whole system was presented, leading to an analysis (section 2.2) of the Route Guide application by defining three components. Then Input and Output to the Route Guide application was analyzed leading to the design (section 2.3) of the Route Guide. During the design a class diagram were constructed and at the end of the design a Component and Interaction diagram combining the components of figure 2.2 with the class diagram of 2.5 was constructed and interaction between classes was described all the way from a request to a response.

The following chapters will concentrate on the UPM and RP components introduced in this chapter. Chapter 3 provides a detail analysis of the UPM component and chapter 4 provides an analysis of the RP component.

CHAPTER 3

User Profile Management

The Route Guide and the system architecture was presented in chapter (2). Three components where found: RP, URRM and UPM. The last one being the topic of this chapter together with an extension of the user profile (UP) concept from the previous chapter. First the idea behind UPM and UP is presented, leading to analysis of required functionality and data foundation. The result is a conceptual model of the UP being transformed into an XML database schema in the last section of the chapter.

3.1 Idea

Most Route Guides of today (e.g. www.krak.dk and www.mapquest.com etc.) provides standard route guides telling the nomadic user how to get from location A to location B, allowing the nomadic user to provide information like preferred highways, intersections etc. when the route is created. Afterwards the nomadic user can choose between fastest -, shortest - and avoid highway routes.

Our idea is that the above mentioned information that enables the Route Guide to provide the nomadic user with route guidance should have some supplementary information that allows the guidance to be customized according to the requirements of the nomadic user. On flaw with most route guides is their inability to remember information provided by the nomadic user, resulting in a tedious repetitions of information over and over. The combination of providing persistent storage and access from different platforms could be used to plan a route from the desktop at home and benefit from the larger screen and keyboard of the desktop computer. Later the user can access the route from a mobile computer.

We propose supplementary functionality (This is UPM) to the ordinary Route Guide that enables storing of personal information (here after referred to as a *user profile*) The following is an analysis of needed functionality, organization and content of such a UP.

3.2 Functionality and Data Analysis

In order to handle customized route guidance the Route Guide needs to be extended with functionality for modification, deletion, retrieval and storing of information and creation of UPs that is going to be used by the Route Guide when it creates new routes for nomadic users.

Creation, modification and deletion directly involves nomadic user interaction, these will be categorized as *external*, the last two will be categorized as *internal* due to the fact that they do not involve nomadic user interaction.

- **External**
    1. *Creation*: The Route Guide should provide an interface that allows first time nomadic users to create a new UP.
    2. *Modification*: Functionality that allows the nomadic user to add or change the content of the UP.
    3. *Deletion*: Provides an interface allowing the nomadic user to either delete the hole UP or parts of it.
- **Internal**
    1. *Retrieval*: Functionality is needed that allows the Route Guide to retrieve information stored in the UP
    2. *Storage*: Functionality that provides stable storage of UPs.

Before a more detailed analysis of the required functionality, lets look at the content and organization of the UP. By using the Entity-Relationship (ER) model from [11] a conceptual schema for the user profile is developed. From figure 3.1 it can be seen that a UP consist of five entities; Route, RouteType, NomadicUser, RoutePoint and Service. Lets look at each of the entities in more detail.

Route The UP allows the nomadic user to store routes for later use. A route is uniquely identified by the *RouteID* and has an associated name called *RouteName* allowing the nomadic user to give names like ″Easter Holiday″ to the route.

RouteType The RouteType entity is used to hold information about the route type, that a nomadic user have specified for a specific Route. A RouteType is uniquely identified by the *RouteTypeId* and the *RTname* attribute having one of the following values;

$$Route\,Type = \{Shortest, Fastest, Scenic\,Drive\}$$

The cardinality between RouteType and Route is one-to-many, because one RouteType may belong to many Routes and one Route can only have one RouteType.

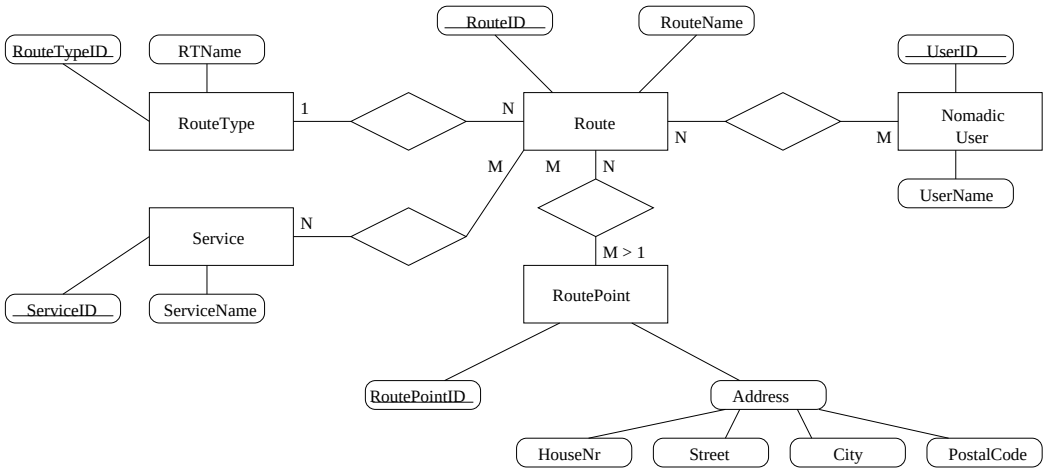


Figure 3.1: User Profile E-R diagram

NomadicUser Each nomadic user is uniquely identified by the use of the NomadicUser entity, consisting of a *UserId* and a *UserName*. By having this entity the concept of a "User Profile" actually becomes a set of α nomadic users having a set of β routes stored. Because one Route can belong to many NomadicUsers and because a NomadicUser can have many Routes defined, the cardinality between Route and NomadicUser is many-to-many. Notice that the NomadicUser entity easily could have several other attributes describing the nomadic user. Attributes that could be used by the Route Guide to provide customized route guidance. For the sake of simplicity we will settle with *UserID* and *UserName*.

RoutePoint By supplying the composite attribute *Address* consisting of *HouseNr*, *Street*, *City* and *PostalCode* the nomadic user is allowed to specify RoutePoints that the Route should consist of. This means that when the Route Guide (or more precise the RP) is calculating the route, the route

should consist of RoutePoints the nomadic user have specified as being a part of the route. Each RoutePoint is uniquely identified by a *RoutePointID*. The cardinality between Route and RoutePoint is many-to-many with the additional constraint that the relationship between the two entities must contain at least two RoutePoints. This means that one Route have two of more RoutePoints[1] and one RoutePoint can belong to many Routes.

Service Finally the nomadic user can specify Services, e.g. Gas Station, that should be available on the Route. Each Service is uniquely identified by the *ServiceID* and has an associated attribute *ServiceName* with values like;

$$Service\ Name = \{Gas\ Station, Gas\ Station\ (Repair\ Shop) \\ Gas\ Station\ (Repair\ Shop, Food)\}$$

Because one Route can have relations to many Services that should be a part of the Route and because Service (e.g. Gas station) can belong to many Routes, the cardinality between Service and Route is many-to-many.

Figure 3.1 shows each of the entities of the UP and relations between these entities. Lets return to the needed functionality and describe each of them in more detail.

3.2.1 Creation

This functionality should provide the nomadic user with an easy to use option menu that enables the nomadic user to create new routes, by specifying *RouteName* and *RouteType*. The nomadic user is also allowed to enter α sets of $\{HouseNr,\ Street,\ City,\ PostalCode\}$ indicating RoutePoints that must be a part of the new Route. The option menu also allows the nomadic user to select which Services should be part of the new Route. Please notice that the amount of Services is determined by the provider of the Route Guide. This is due to the fact, that allowing the nomadic user to enter e.g. his own Service could result in Services that does not exist or can not be associated with a geographical location on a map. Furthermore, being able to provide additional information about services chosen by the nomadic user depends on this information being available to the service provider. One can not provide information on repair discounts given by a Gas Station if this information is not available online (see appendix C for a scenario describing this problem).

[1] Having only one RoutePoint would not make sense, because one can not create a route from point A to point B, when only point A exist.

3.2.2 Modification

By using the modification functionality the nomadic user is allowed to change settings for each of his predefined routes. E.g change *RouteName* ″Easter Holiday″ to ″Spring Holiday″ or change Service from Gas Station to Gas Station with Repair Shop.

3.2.3 Deletion

This functionality allows the nomadic user to either delete a route or delete one or more services associated with a route. Notice that when a route is deleted all associated RoutePoints and Services are also deleted. On the other hand deletion of e.g. one Service does not infect the Route in other ways than the Service is no longer a part of the route.

3.2.4 Retrieval

When the Route Guide is asked to provide route guidance to a nomadic user, one of two things can happen; 1) The nomadic user have requested an already defined route stored in the UP, or 2) The nomadic user have requested a new route by providing start - and end location.

Case 1 The Route Guide (UPM and URRM in cooperation) retrieves all the route information (RouteName, RoutePoints, ServiceName etc.), from the UP and by the help of the RP, the Route Guide creates the route and returns route guidance to the nomadic user.

Case 2 The URRM passes the start - and end location to the RP, which returns route guidance that is returned to the nomadic user.

3.2.5 Storage

The last functionality needed in order to provide customized route guidance is storage of the UP. This functionality should provide stable storage and easy retrieval of UPs created by a nomadic user.

We propose to use an Extensible Markup Language (XML) database for storage of UPs. Storing UPs in an XML database makes it easy to translate the data content by the use of Extensible Style Language (XSL)/Extensible Style Language Transformation (XSLT) to either WML, HTML or VoiceXML. Furthermore XML databases supports structured querying by the help of Extensible Query Language (XQL), allowing retrieval of information stored in the database. If it is required that the UP should be able to follow a nomadic user, e.g. move to another location, then the use of XQL against the XML

database can based upon e.g. user id create a new XML document containing all information associated to the user id. The resulting XML document can then be moved to a new location[2].

Surely the same features as mentioned above could be accomplished by the use of an ordinary relational database like Oracle, Dbase, Sybase etc. One drawback is that the internal representation of data in these database either have to be 1) converted/translated to XML or 2) extracted in other ways to the target platform. Case 1: data stored in the database have to be translated to XML in order to gain the same benefits as with the XML database. Case 2, e.g. Hypertext Preprocessor (PHP)[3] or Active Server Page (ASP) could be used as part of HTML pages in order to display the content of a UP stored in one of these databases. But then, what about client devices only supporting WML?. Of course these tasks can be accomplished as mentioned or by other means, but the important message is, that these functionalities are already provided by the XML database.

Another drawback is the size of these databases compared to an XML database which in reality is an ordinary text file containing data and metadata (Document Type Definition, (DTD)).
Furthermore, when using an XML document as the database the DTD can be used to exchange data with other applications. Because the DTD works as a grammar for the XML document describing document content and structure.

To conclude, it is found that an XML database fulfills the requirements; it is small, provides easy retrieval of data and most important, by storing data as XML they can easily by transformed to WML, HTML or VoiceXML.

3.3 User Profile Design

In section 3.2 the ER model was used to develop a conceptual model of the user profile. In this section the ER model is transformed to an XML database schema by going through three steps: 1) Mapping the Conceptual Schema to a Relational Database Schema, 2) Normalization of the Relational Database Schema and 3) Transformation of the Relational Database Schema to a XML Database Schema.

[2]This feature could for instance be used to replicate the UP to a client device for off-line browsing or modification.

[3]PHP is an HTML-embedded scripting language. Much of its syntax is borrowed from C, Java and Perl with a couple of unique PHP-specific features thrown in. The goal of the language is to allow web developers to write dynamically generated pages quickly.

3.3.1 Mapping Conceptual Schema to Relational Database Schema

By following seven steps from [11] a relational database schema is derived from the ER model by using ER-to-Relational Mapping.

Step 3.3.1 *For each regular entity type E in the conceptual schema (ER-model), create a relation R that includes all the simple attributes of E. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of E as primary key.*

From the ER-model five relations; RouteType, Route, NomadicUser, Service and RoutePoint, is created (See figure 3.2) to correspond to the regular entity types RouteType, Route, NomadicUser, Service and RoutePoint. RouteType ID, RouteID, UserID, ServiceID and RoutePointID are choose as primary keys. Notice that the composite attribute *Address* becomes *Street, HouseNr, City* and *PostalCode* in the RoutePoint relation.

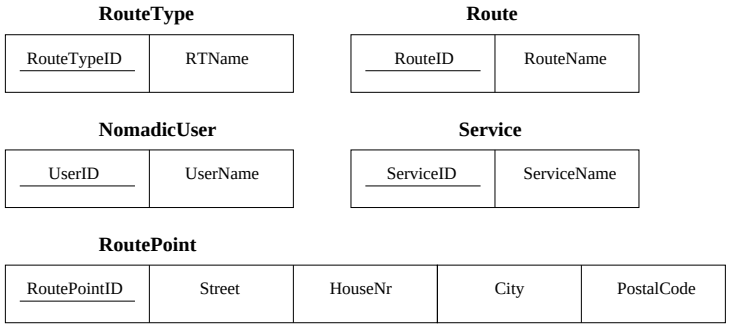


Figure 3.2: R relations with primary key and simple attributes

Step 3.3.2 *For each weak entity type W in the conceptual schema with owner entity type E, create a relation R, and include all simple attributes of W as attributes of R. In addition, include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s)*

No additional relations R are created from this step, due to the fact that the conceptual schema (ER model) does not contain weak entity types W.

Step 3.3.3 *For each binary one-to-one relationship type R in the ER schema, identify ...*

As with step 3.3.2 there does not exist any binary one-to-one relationship type R, hence step 3.3.3 does not result in any additions to the five relations of figure 3.2

Step 3.3.4 *For each regular binary one-to-many relationship type R, identify the relationship S that represents the participating entity type at the $N-side$ of the relationship type. Include as foreign key in S the primary key of the relation T that represent the other entity type participating in R. Include any simple attributes of the one-to-many relationship type as attributes of S.*

Between RouteType and Route a binary one-to-many relationship exist, therefore, according to step 3.3.4, the primary key RouteTypeID of entity RouteType is included as foreign key at the $N-side$ of the relationship (Route) together with any simple attributes of the one-to-many relationship. The new relational database schema resulting from step 3.3.4 can be seen at figure 3.3.

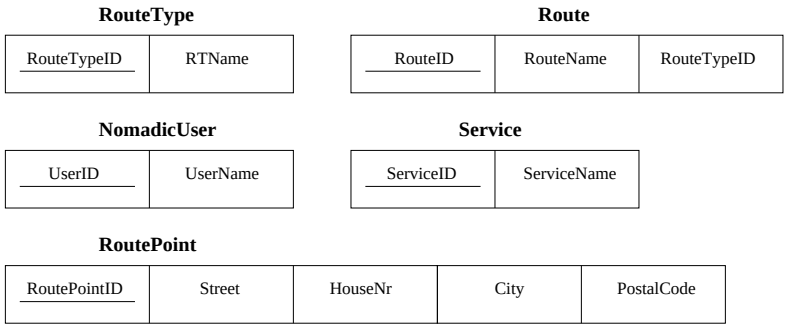


Figure 3.3: R relations after step 3.3.1 . . . 3.3.4

Step 3.3.5 *For each binary many-to-many relationship type R, create a new relation S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S. Also include any simple attributes if the many-to-many relationship type as attributes of S.*

Three binary many-to-many relationships exist on the ER model. These three relationships are mapped to three new relations; RouteHasService, NomadicUserOwnRoute and RouteConsistOfRoutePoint, shown at figure 3.4.

Step 3.3.6 *For each multi-valued attribute S, create a new relation R that . . .*

As figure 3.1 does not include any multi-valued attributes this step does not add any thing to the relational database schema and it remains as shown at figure 3.4.

Step 3.3.7 *For each n-ary relationship type R, $n > 2$, create a new relation S to represent ...*

Neither does figure 3.1 include any n-ary relationships, so figure 3.4 becomes the final relational database schema, after the seven steps from [11] have been used to map the conceptual schema.

By the use of normalization [11] the quality of the relational database schema is measured and improved by removing composite attributes, checking functional dependency etc. And finally the relations are on Boyce-Codd normal form.

3.3.2 Relational Database Schema to XML Database Schema

We have arrived at the final step, the transformation of the relational database schema to a XML database schema. The transformation will be based upon eleven rules from [12][13].

Rule 3.3.1 *Choose the Data to Include. Based on the business requirement the XML database document will be fulfilling, decide which tables and columns from our relational database will need to be included in our documents.*

Well, all information from the relations will be needed in order to create a Route Guide providing customized route guidance, therefore the whole relational database schema is included.

Rule 3.3.2 *Create a Root Element. Create a root element for the document. Add the root element to our DTD*[4]*, and declare any attributes of that element*

[4]Think of the DTD as the XML database schema, data describing data (meta-data)

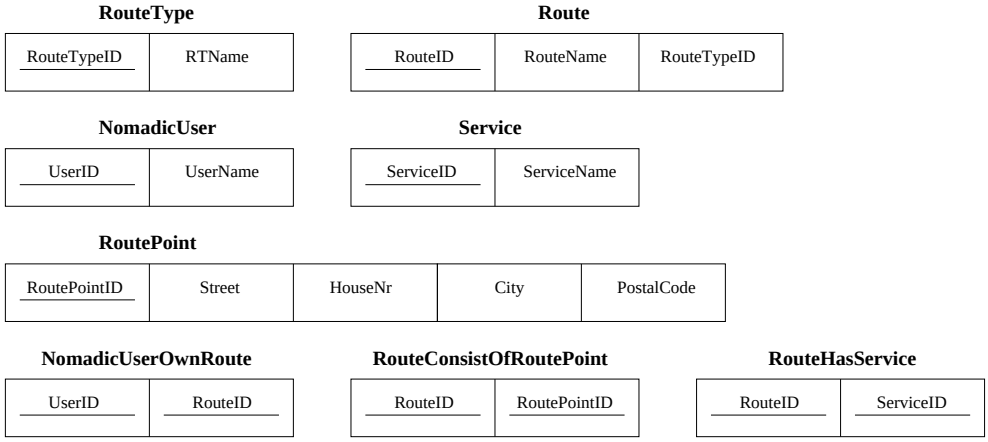


Figure 3.4: Many-to-many relationships as new Relations

that are required to hold additional semantic information. Root element's names should describe their content.

To represent the relational database schema (see figure 3.4), a root element called <UserProfile> is created, holding the other elements that will be created:

```
<!ELEMENT UserProfile EMPTY>
```

The root element does not have an associated attribute list.

Rule 3.3.3 *Model the Content Tables. Create an element in the DTD for each content table*[5] *we have chosen to model. Declare these elements EMPTY for now.*

All relations from figure 3.4 are content tables according to rule 3.3.3, and are add as new elements to the DTD.

```
<!ELEMENT UserProfile EMPTY>
<!ELEMENT RouteType EMPTY>
<!ELEMENT Route EMPTY>
<!ELEMENT NomadicUser EMPTY>
<!ELEMENT Service EMPTY>
<!ELEMENT RoutePoint EMPTY>
<!ELEMENT RouteHasService EMPTY>
<!ELEMENT NomadicUserOwnRoute EMPTY>
<!ELEMENT RouteConsistOfRoutePoint EMPTY>
```

Rule 3.3.4 *Modeling Non-foreign Key Columns. Create an attribute for each column we have chosen to include in our XML document (except foreign key columns). These attributes should appear in the !ATTLIST declaration of the element corresponding to the table in which they appear. Declare each of these attributes as CDATA, and declare it as #IMPLIED or #REQUIRED depending on whether the original column allowed nulls or not.*

Adding all attributes that are not foreign keys, leads to the following DTD. Notice that there are not any attributes associated with the root element <UserProfile>.

```
<!ELEMENT UserProfile EMPTY>
<!ATTLIST UserProfile>
<!ELEMENT RouteType EMPTY>
<!ATTLIST RouteType
RouteTypeID CDATA #REQUIRED
RTName CDATA #REQUIRED>
<!ELEMENT Route EMPTY>
<!ATTLIST Route
RouteID CDATA #REQUIRED
RouteName CDATA #REQUIRED>
<!ELEMENT NomadicUser EMPTY>
<!ATTLIST NomadicUser
UserID CDATA #REQUIRED
UserName CDATA #REQUIRED>
<!ELEMENT Service EMPTY>
<!ATTLIST Service
ServiceID CDATA #REQUIRED
ServiceName CDATA #REQUIRED>
<!ELEMENT RoutePoint EMPTY>
<!ATTLIST RoutePoint
RoutePointID CDATA #REQUIRED
```

[5]According to [12] content tables are tables that simply contain a set of records (e.g all customer addresses for a certain company), notice that relating tables like NomadicUserOwnRoute on figure 3.4 is treaded as content tables.

```
Street CDATA #REQUIRED
HouseNr CDATA #REQUIRED
City CDATA #REQUIRED
PostalCode CDATA #REQUIRED>
<!ELEMENT RouteHasService EMPTY>
<!ATTLIST RouteHasService
RouteID CDATA #REQUIRED
ServiceID CDATA #REQUIRED>
<!ELEMENT NomadicUserOwnRoute EMPTY>
<!ATTLIST NomadicUserOwnRoute
UserID CDATA #REQUIRED
RouteID CDATA #REQUIRED>
<!ELEMENT RouteConsistOfRoutePoint EMPTY>
<!ATTLIST RouteConsistOfRoutePoint
RouteID CDATA #REQUIRED
RoutePointID CDATA #REQUIRED>
```

Rule 3.3.5 *Add ID Attributes to the Elements. Add an ID attribute to each of the elements we have created in our XML structure (with the exception of the root element). Use the element name followed by ID for the name of the new attribute, watching as always for name collisions. Declare the attribute as type ID, and as #REQUIRED*

Using rule 3.3.5 and remembering not to create an ID for the root element <UserProfile>, we change RouteTypeID, RouteID, UserID, ServiceID and RoutePointID defined as type CDATA (according to rule 3.3.4) to type ID, instead of adding yet another ID attribute for each element. We leave the primary keys of RouteHasService, NomadicUserOwnRoute and RouteConsistOfRoutePoint as they are for now, because these are handled as foreign keys later. We add new ID attributes to each of these three elements.

```
<!ELEMENT UserProfile EMPTY>
<!ATTLIST UserProfile>
<!ELEMENT RouteType EMPTY>
<!ATTLIST RouteType
RouteTypeID ID #REQUIRED
RTname CDATA #REQUIRED>
<!ELEMENT Route EMPTY>
<!ATTLIST Route
RouteID ID #REQUIRED
RouteName CDATA #REQUIRED>
<!ELEMENT NomadicUser EMPTY>
<!ATTLIST NomadicUser
UserID ID #REQUIRED
UserName CDATA #REQUIRED>
<!ELEMENT Service EMPTY>
<!ATTLIST Service
ServiceID ID #REQUIRED
ServiceName CDATA #REQUIRED>
<!ELEMENT RoutePoint EMPTY>
<!ATTLIST RoutePoint
RoutePointID ID #REQUIRED
Street CDATA #REQUIRED
HouseNr CDATA #REQUIRED
City CDATA #REQUIRED
PostalCode CDATA #REQUIRED>
<!ELEMENT RouteHasService EMPTY>
<!ATTLIST RouteHasService
RouteHasServiceID ID #REQUIRED
RouteID CDATA #REQUIRED
ServiceID CDATA #REQUIRED>
<!ELEMENT NomadicUserOwnRoute EMPTY>
<!ATTLIST NomadicUserOwnRoute
NomadicUserOwnRouteID ID #REQUIRED
UserID CDATA #REQUIRED
RouteID CDATA #REQUIRED>
<!ELEMENT RouteConsistOfRoutePoint EMPTY>
<!ATTLIST RouteConsistOfRoutePoint
```

```
RouteConsistOfRoutePointID ID #REQUIRED
RouteID CDATA #REQUIRED
RoutePointID CDATA #REQUIRED>
```

Rule 3.3.6 *Representing Lookup Tables. For each foreign key that we have chosen to include in our XML structures that references a lookup table:*

- *Create an attribute on the element representing the table in which the foreign key is found.*
- *Give the attribute the same name as the table referenced by the foreign key, and make it #REQUIRED if the foreign key does not allow NULLS or #IMPLIED otherwise.*
- *Make the attribute of the enumerated list type. The allowable values should be some human-readable form of the description column for all rows in the lookup table.*

When using 3.3.3 it was found that all tables in the User Profile are content tables, and therefore rule 3.3.6 does not contribute any thing to the DTD.

Rule 3.3.7 *Adding Element Content to Root elements. Add a child element to the allowable content of the root element for each table that models the type of information we want to represent in our document.*

As a result of rule 3.3.1 it was concluded that the whole relational database schema should be included, therefore the content model for the root element <UserProfile> consist of all the elements created by the use of rule 3.3.3. The DTD is extended as follows:

```
<!ELEMENT UserProfile (RouteType*, Route*, NomadicUser*, Service*,
RoutePoint*, RouteHasService*, NomadicUserOwnRoute*,
RouteConsistOfRoutePoint*) >
<!ATTLIST UserProfile>
...
```

With figure 3.4 eight relations where identified. Because of the addition of the relations: NomadicUserOwnRoute, RouteConsistOfRoutePoint and RouteHasService as modeling many-to-many relationships from the initial conceptual model of figure 3.1. The result is eight relations having only one-to-many or many-to-one relationships, see figure 3.5.

As an example, look at the NomadicUser relation of figure 3.5, this relation has a one-to-many relationship to NomadicUserOwnRoute, which has a many-to-one relationship to Route. This is how the many-to-many relationships of figure 3.1 is modeled [17]. By looking at figure 3.5 arrows can be seen showing the navigation direction of relationships. These navigation directions have to be established because they determine where the ID-IDREF or parent-child relationships in rule 3.3.8 and 3.3.9 should be. For example, in this case it is much more likely to navigate from NomadicUser to Route when searching/querying for information, than the other way around.

Rule 3.3.8 *Adding Relationships through Containment. For each relationship we have defined, if the relationship is one-to-one or one-to-many in the direction it is being navigated, and no other relationship leads to the child within the selected subset, then add the child element as element content of the parent element with the appropriate cardinality.*

No one-to-one relationships exist at figure 3.5, but the following one-to-many relationships exist; NomadicUser-to-NomadicUserOwnRoute, Route-to-RouteHasService and Route-to-RouteConsistOfRoutePoint. This means that the many-side of these relationships are added as child elements of the parent elements (one-side of the relationships):

```
...
<!ELEMENT Route (RouteHasService*, RouteConsistOfRoutePoint*)>
<!ATTLIST Route
RouteID ID #REQUIRED
RouteName CDATA #REQUIRED>
<!ELEMENT NomadicUser (NomadicUserOwnRoute*)>
<!ATTLIST NomadicUser
UserID ID #REQUIRED
UserName CDATA #REQUIRED>
...
```

Rule 3.3.9 *Adding Relationships using IDREF/IDREFS. Identify each relationship that is many-to-one in the direction we have defined it, or whose child is the child in more than one relationship we have defined. For each of these relationships, add an IDREF or IDREFS attribute to the element on the parent side of the relationship, which points to the ID of the element on the child side of the relationship.*

At figure 3.5 there exist the following many-to-one relationships:
NomadicUserOwnRoute-to-Route, RouteHasService-to-Service, Route-to-RouteType and RouteConsistOfRoutePoint-to-RoutePoint. Therefore we

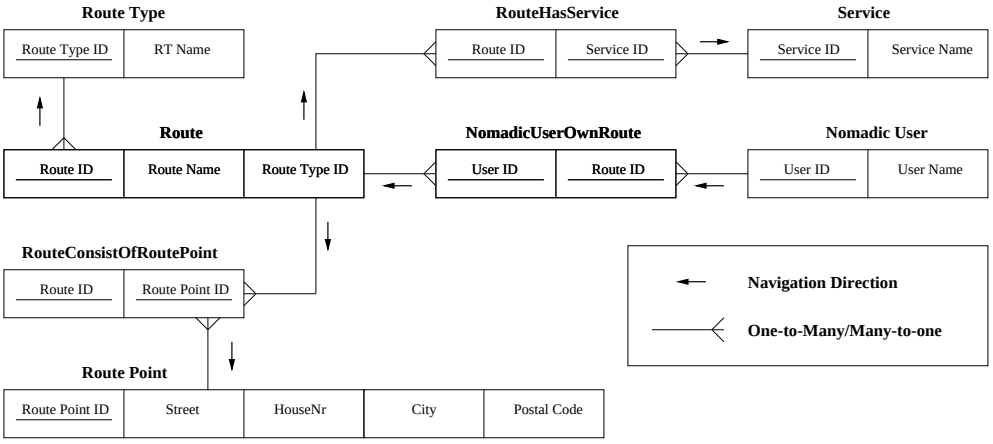


Figure 3.5: Relationships between Relations

add an IDREF attribute to the parent element (the many-side of the relationship) which points to the ID of the child element (the one-side of the relationship):

```
...
<!ELEMENT Route (RouteHasService*, RouteConsistOfRoutePoint*)>
<!ATTLIST Route
RouteID ID #REQUIRED
RouteName CDATA #REQUIRED
RouteTypeIDREF IDREF #REQUIRED>
...
<!ELEMENT RouteHasService EMPTY>
<!ATTLIST RouteHasService
RouteHasServiceID ID #REQUIRED
RouteID CDATA #REQUIRED
ServiceID CDATA #REQUIRED
ServiceIDREF IDREF #REQUIRED>
<!ELEMENT NomadicUserOwnRoute EMPTY>
<!ATTLIST NomadicUserOwnRoute
NomadicUserOwnRouteID ID #REQUIRED
UserID CDATA #REQUIRED
RouteID CDATA #REQUIRED
RouteIDREF IDREF #REQUIRED>
<!ELEMENT RouteConsistOfRoutePoint EMPTY>
<!ATTLIST RouteConsistOfRoutePoint
RouteConsistOfRoutePointID ID #REQUIRED
RouteID CDATA #REQUIRED
RoutePointID CDATA #REQUIRED
RoutePointIDREF IDREF #REQUIRED>
```

Rule 3.3.10 *Add Missing Elements. For any element that is only pointed to in the structure created so far, add that element as allowable element content of the root element. Set the cardinality suffix of the element being added to *.*

In rule 3.3.7 (Adding Element Content to Root elements) all relations where add to the root element as allowable element content of the root element <User Profile>, therefore if there existed elements that where only pointed to in the structure (see rule 3.3.10), they would already have been add as allowable element content of the root element. Rule 3.3.10 therefore does not add anything to the DTD.

Rule 3.3.11 *Remove Unwanted ID Attributes. Remove ID attributes that are not referenced by IDREF or IDREFS attributes in the XML structures.*

On review, the UserID, RouteHasServiceID, NomadicUserOwnRouteID and RouteConsistOfRoutePoint attributes are not referenced by IDREF or IDREFS attributes, all are removed except UserID, which is going to be used to uniquely identify each nomadic user.

The final XML database schema (DTD) can be seen in appendix D.

3.4 Chapter Summary

First the idea behind UPM and UP were presented, leading to analysis of required functionality and data foundation. The result was a conceptual model (ER model) of the UP being transformed into an XML database schema (DTD) in the last section of this chapter. The five functions create, modify, delete, retrieve and store needed to provide UPM were also identified.

In chapter 4 the RP component is analyzed, providing insight into important topics of RP requirements.

CHAPTER 4

Route Planning

In chapter 2 and 3 the UPM and URRM components were analyzed. In this chapter the last component (Route Planning (RP)) provided by the Route Guide is analyzed.

First the Road Network used by the RP is presented together with the idea behind the geocoding process. This is continued with the analysis of the RP.

4.1 Road Network

In order for the RP to plan a route it should have access to information representing the road network. In this project the road network is represented by an ArcView theme, in this way ArcView is used as a database containing the road network.

4.1.1 Facilities and Geocoding

Features[1] can be related to geographical locations by their address. This is done by the help of geocoding [14], which means joining a table with features with a table representing street data (Road Network). The joining attribute of both tables are address (see figure 4.1). By performing the geocoding process features becomes facilities[2] that can be displayed on a view. Becoming a facility means that normal avenue functions as FindPath() and FindClosestFacility() can be used to perform route planning.

[1]Examples of features could be: Hospital, Gas Station, Harbour and Train Station etc.

[2]The service entity from figure 3.1 is an example of a facility (e.g. Gas Station) that needs to be represented as a facility

During the geocoding process, ArcView creates a geocoded theme in the form of an ArcView shapefile (see figure 4.1). This shapefile is used to store the cells of each record in the feature table. Some of the cells holds the XY coordinates of the successfully matched records. These XY coordinates are the geographical location of the facility.

Based on the shapefile, from the geocoding process, it is possible to see roads and facilities on the same view. Views that can be exported and sent to the nomadic user as maps showing a route.

During the geocoding process ArcView also creates files like e.g. <name>.idx (see figure 4.1), these are not meant to be human readable. The purpose of these files are for internal use by ArcView to maintain geographic relations between items like roads and facilities.

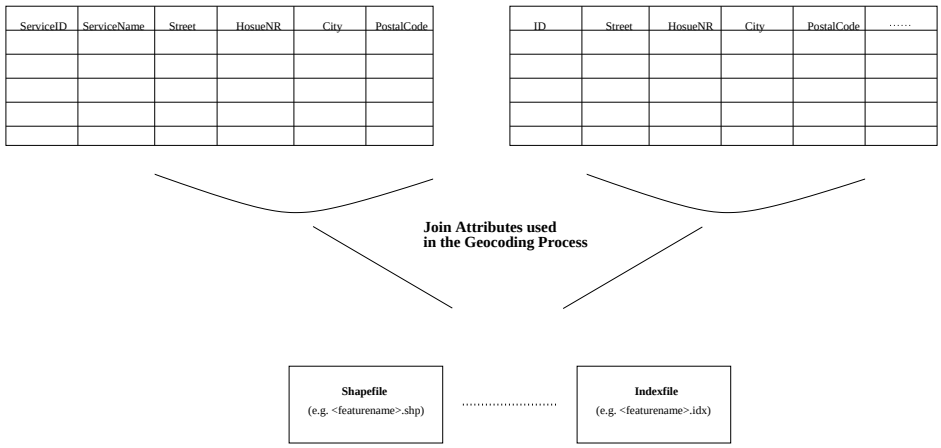


Figure 4.1: Illustration of the Geocoding Process

4.2 Route Planning Analysis

When developing the RP different requirement should be realized to accomplish the needs of the user. The simplest need to accomplish is to plan a route from point A to B. In the case where ArcView is used to host the road network the route planning of a route from point A to B is done by the use of a function called FindPath.

A requirement from the user regarding route planning is the need of a route from point A to B that pass a particular facility. In this case valid

facilities are all addresses in an ArcView theme. This task can also be accomplished by the FindPath function in ArcView by specifying a point-list[3] of addresses the route should pass.

This solution has the drawback that the nomadic user has to know the addresses of facilities. Therefore the RP should provide functionality allowing the nomadic user to specify facility names (E.g Gas Station) as well as the street address of the facility. To accomplish the use of facility names as an option instead of addresses additional data requirement and functional requirement should be considered. When using the facility name instead of the street address of the facility it is the task of the Route Guide to find the location of the facility. The idea is to find the facility with the location that gives the optimal route.

4.2.1 Data Requirement

When using facility names instead of addresses the RP need data about the location of facilities. These data are included in the ArcView database by the following steps:

- Make e.g. a file containing facility name, Street, HouseNr, City and Postal Code etc., in dbase format (.dbf).
- The addresses of the facility is then geocoded by the help of the table containing street addresses (Road Network). The geocoding process joins facilities to street addresses, by using the addresses of both tables as join attribute (see figure 4.1).

This process of making a .dbf file and geocode it with the street addresses makes it possible to find the address of a facility by the use of facility name. Furthermore the geocoding process makes it possible for the FindPath function to locate a facility on the view. When the street address is known the route can be planned as before from point A to B or A to B through C, where C is a facility. But now the nomadic user can specify the facility name instead of the street address of the facility. By specifying a service[4] in the User Profile (see chapter 3) the nomadic user can specify a service that should be a part of a route.

4.2.2 Functional Requirement

In this section, solutions to the task of route planning is presented. The solutions shows issues relevant to route planning meant for use within the

[3] A point-list is an ordinary list, with the exception that each element is a geographical point on a map

[4] Service name in UP equals facility name in ArcView

Route Guide prototype. In the case of commercial products this algorithm should be review for further enhancement, e.g. efficiency.

Consider a Route Guidance Request (see figure 2.2) that specifies the needs of a route from point A to B through C. Where C is a facility name.

The following solution to the task above is presented where only one facility name (e.g Gas Station) is given. At the end of the section an extension will be made to the solution that allows an unlimited number of facility names (e.g Gas Station, Hospital, Habour and Train Station) to be specified.

If the facility name equals Gas Station, the task of the RP is to make one Gas Station a part of the Route from A to B that gives the optimal route. To solve this task some options are available:

- **Option One**: Plan α routes from location A to B using one facility each time. For each route calculate travel distance, pick the route with the lowest cost (see figure 4.2). This option will find the optimal route but the drawback is extensive computation if many facilities exist. Another drawback is computation power used on routes as Route one of figure 4.2, a route where facility D leads to an route where the first part from A to D goes in the wrong direction[5], resulting in a route that should not be a part of the set of α routes calculated because it from the start does not have any change of becoming the optimal route. Somehow it should be possible to limit the area of facilities to an area with facilities only leading to routes that moves in the right direction, e.g. towards B.
- **Option Two**: Limit the area of facilities to those that result in routes that moves towards the end location B. The problem of option two is to determine the shape and size of such an area, without leading to an area with no facilities.

[5]Wrong direction means a direction resulting in a route moving away from B, instead of towards B on figure 4.2

4.2.3 Route Planning with Facilities

A solution that combines the effectiveness and efficiency of option one and two to find the optimal route that goes trough one of more facilities is shown at figure 4.3

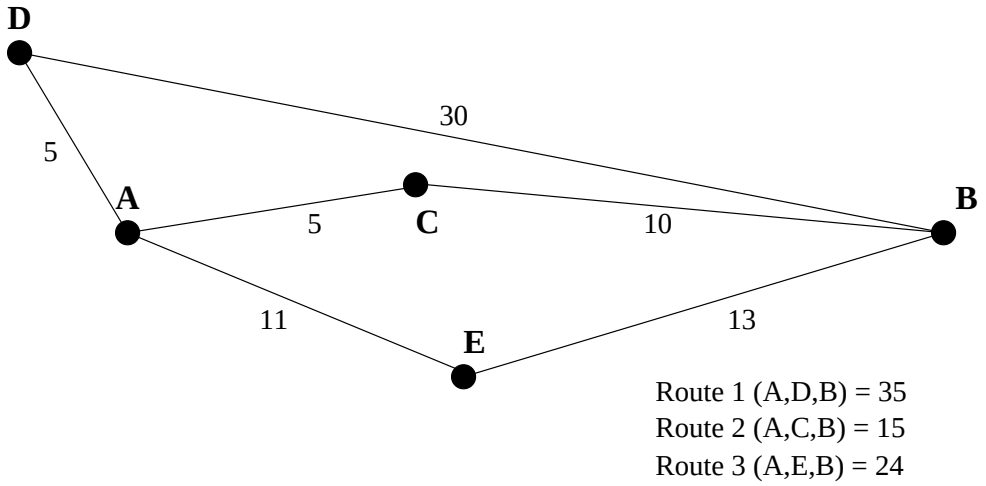


Figure 4.2: Option 1; α routes are calculated

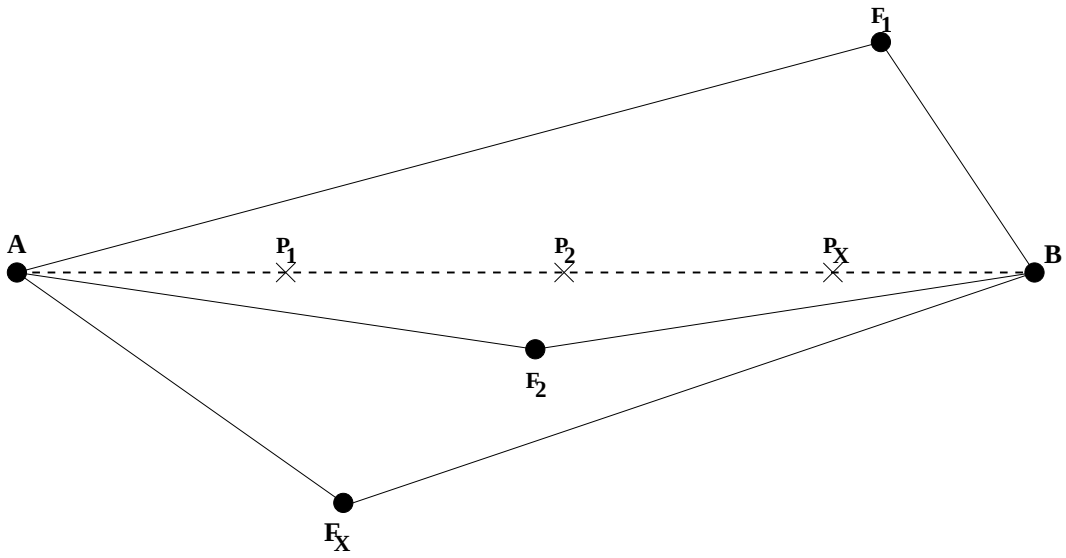


Figure 4.3: Optimal route with facilities

The idea shown at figure 4.3 is as follows:

Algorithm 4.2.1

1. *Find the optimal route from A to B.*
2. *Place x facility points* ($P = \{p_1, p_2 \ldots, p_x\}$) *on the optimal route from A to B with equal distance.*
3. *Locate the f closest facilities from each facility point* p_x *where* $f \subseteq F$ (*allfacilities*) *and* $F = \{f_1, f_2 \ldots, f_x\}$.
4. *Determine the cost from A to B through each facility* f_x.
5. *Choose the facility* f_x *that gives the optimal route.*

The first step of algorithm 4.2.1 is to find the optimal route from point A to B. This is done by the use of the **FindPath** function provided by the avenue script language in ArcView. At figure 4.3 the optimal route from A to B is represented by the dashed line.

In the second step of algorithm 4.2.1, P facility points[6]. are placed on the optimal route from A to B with equal distance. The facility points are temporary marks used to spread the search of facilities along the optimal route. In this way the search for the closest facilities are limited to the area (Option two) between A and B and close to the optimal route.

The third step is to find the f closest facilities to each facility point p for that purpose the **FindClosestFacility** function in avenue is used.

If the route should be planned to pass more than one type of facility, f facilities should be found for each kind of facility.

In the fourth step, routes from A to B are planned through all f facilities from each facility point p and the costs are determined. This means that $f * p$ routes have to be planned.

If the route should pass more than one type of facility, routes with combinations of the different types of facilities should be planned. Each route with one of each facility. In this case the number of routes to plan are: $(F * P)^T$ where T is the number of different facility types. As a consequence of this, the number of facilities for each facility point and the number of facility point itself should be kept at a minimum.

[6]They are called facility points because they are going to be points from where facilities are located

In the fifth step the planned routes are compared and the optimal one is selected as the optimal route from A to B passing the facility.

The optimal number of facility points P and facilities F depends on the road environment. Because the road environment change depending on the place it is hard to choose optimal number of facility points P and facilities F.

A modification to the described solution would be to determine the amount of facility points P based upon the length of the optimal route planned in the first step. For instance one facility point for each 10 km.

As long as the purpose of this project is to develop a prototype of a Route Guide it will be out of scoop to determine the optimal numbers of facility points P and facilities F.

One might argue that if this development was not based on ArcView the RP should be designed different. But we think that this solution still is worth considering. Because it would be natural to build a function like the **findShortestRoute** in ArcView to reach a higher abstraction and base further development on such a function (eg. specification of facility name instead of address).

4.3 Chapter Summary

By the use of the algorithm presented above and the geocoding functionality from ArcView the Route Guide can accomplish the task of planing a route from point A to B through C, where C is a facility given by the facility name.

The algorithm presented here is only meant for use within the prototype. Even through the algorithm is simple it shows ideas of how to minimize the cost when planing a route from A to B through C, where more options are available for the location of C. The primary benefit of the solution presented is the ability to limit the area in which the facility should be found.

The next chapter presents implementation topics related to the three components described during chapter 2, 3 and 4.

CHAPTER 5

Route Guide Implementation

Until know the focus have been on the components of the Route Guide, each of the three components identified in chapter 2 have been analyzed and design have been carried out. The final step is the implementation of a prototype of the Route Guide. The topic of this chapter is to present central issues related to the implementation.

First the implementation of the Retrieve function of the UPI class is presented. This part of the implementation allows the Coordinator of figure 2.6 to retrieve information from the UP. The retrieved information can then be passed on to the RP (the CPR function) for further processing. The exchange of information between the Coordinator and the CPR function of figure 2.6 is illustrated in the second part of this chapter. Finally in the last part the transformation of XML Route Guidance to HTML or WML is presented.

5.1 User Profile Management

The UPM component provides functionality for modification, deletion, retrieval and storing of information and creation of UPs. The following is a illustration of how the Retrieve function is implemented.

5.1.1 The Retrieve Function

In order to provide retrieval of information from a UP the UPI class implements the XQL engine package of figure 2.5. Three things are needed in order to perform a retrieval of information[1] from a UP. 1) XML database,

[1]Recall that information is stored as a XML database

2) Document Object Model (DOM) and 3) a XQL query.

The XML database was create in chapter 3 and the resulting XML database schema can be seen in appendix D.

The DOM is create by the use of the DOMUtil class from the XQL engine package of figure 2.5. This class provides the XMLParser function that creates a DOM from a XML source.

The DOM created is used to create a XQL query that can be executed by the help of the execute function from the XQL class (also a part of the XQL engine package). The result of the execution of the query is a XQLResult object contain the result of the executed query.

A prototype implementation of the Retrieve function from the UPI class can be found in appendix E.

5.2 Server and Client Interface Implementation

During the analysis of the system architecture in chapter 2 the need for a interface between the URRM and RP (or more precise the Coordinator and CRG parts of figure 2.6) were identified (see figure 2.1). The RP component is implemented by ArcView providing a DDE server. In order for the URRM to communicate with the RP a DDE client is needed. By extending the JavaDDE from [8] the ArcViewInterface class from figure 2.5 implements the DDE client providing the URRM with three functions; DDEserver_connect(), DDEserver_disconnect and Execute_CRG(). The following shows the implementation of the Execute_CRG function.

5.2.1 Execute Create Route Guidance

The Execute_CRG() function takes a byte array and a integer as input parameters. The integer specifies the size of the byte array containing parameters to be used by the CRG function implemented in ArcView by the help of avenue. The parameters passed in the byte array to the CRG function is either retrieved from the UP by the help of the UPM component or received directly from the user. Examples of parameter values are; Servicename, RT-name and a set α addresses, where $\alpha = \{HouseNr, Street, City, PostalCode\}$.

The cl.ddePoke statement is were it all happens (see the following code). The first parameter is a string holding a av.run(crg) statement. This string tells the DDE server (ArcView) to run the CRG avenue script that creates

```
  public String Execute_CPR(byte [] data, int dataLength) throws DdeException
  {
    try
    {
      int timeout=5000;
      com.neva.DdeClient cl=new com.neva.DdeClient();
      // Answers the standard clipboard format in which the data item
      // is being requested. Jddeml supports CF_TEXT data format
      int format=com.neva.DdeUtil.CF_TEXT;

      this.DDEServer_connect("ArcView", "System");
      //Execute the avenue script
      cl.ddePoke("av.run(\"crg\", byte [] data, dataLength, timeout)

      this.DDEServer_disconnect();
    }
    catch(DdeException e){return e.getMessage();}
    catch(InterruptedException e){return e.getMessage();}

    return "believe it works!";
  }
}
```

route guidance. The second parameter is the byte array holding values as Servicename. The dataLength parameter specifies the size of the byte array and the timeout parameter determines how long the DDE client (Execute_CRG) will wait for an answer.

5.3 Implementation of User Request and Response Management

In the previous two sections (section 5.1 and 5.2) the prototype implementation of the Retrieve function from the UPI class and the interface between the URRM and RP were described. In this section the part of the output class that translates the internal representation (XML) of Route Guidance to HTML or WML is shown.

5.3.1 Translating XML Route Guidance to HTML/WML

The initial idea was to make the transformation of XML to HTML or WML a functionality provided by the Output class. It turns out to be easier and more flexible to implement the transformation as a separate servlet that can be activated by the Output class whenever a transformation is needed. The transformation is implemented as a servlet called XMLParser (see the class diagram of figure 2.5). The Output class activates the XMLParser by

issuing the URL: http://loda15:8080/servlet/XMLParser?OutPutData.xml, where the query part of the URL[2] is passed in as parameters to the stylesheet. The stylesheet used by the XMLParser at the moment is specified as an init parameter to the servlet.

At the moment the XMLParser returns a HTML page or a deck of WML cards (Depends on the stylesheet used as init parameter to the servlet). A future extension would be to extend the query part of the URL to hold the stylesheet as well as the XML document. This requires a modification of the XMLParser servlet in order for it to be able to receive both an XML document and a stylesheet as parameters and produce a response. A second extension would be to modify the XMLParser in a way that allows the XML-Parser to send the response directly to the nomadic user instead of through the Output class. In order for this to work, the Output class should provide the URL address of the nomadic user as yet another parameter to the XML-Parser.

A prototype implementation of the XMLParser can be seen in appendix F.

5.4 Chapter Summary

In section 5.1 a prototype implementation of the Retrieve funtion from the UPI class were presented. The Retrieve function takes a XML document as input, creates a DOM of the input and executes a XQL query against the DOM. Then in section 5.2 the interface between the URRM and RP were presented. The interface was exam-plied by the Execute_CRG function. Finally the translation of route guidance to HTML or WML was described in section 5.3. The translation of route guidance is done by the XMLParser servlet activated by the Output class. Two extension were proposed: 1) stylesheet as a part of the URL query part and 2) Nomadic user address as a part of the URL, allowing the XMLParser to return response directly to the nomaidc user.

With this implementation chapter the report have come to an end, only the conclusion presenting the main points of the report remain together with suggestion for future work.

[2]The query part, is the subset of the URL after the ?

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

During the project, we have gained experience in several areas related to the development of a Route Guide. To understand issues of a Route Guide an architecture, a class diagram and a component- and interaction diagram have been developed, showing components used to provide route guidance for nomadic user.

We have found that XML is a technology providing a flexible way of handling data to be exchanged between various part of a Route Guide. Furthermore XML and related technologies as XSL and XQL have been used as the foundation to communicate with users at different platforms in an effective way. This has been accomplish by the use of different stylesheets, one for each platform.

To determine the stylesheet to be used for a particular user, browsersniffing was included. Browsersniffing means to detect e.g. the platform and browser version used by the user. Based on this knowledge the appropriate stylesheet is chosen.

The User Profile introduced in the report is based on XML technology allowing easy transformation of information in the User Profile to WML, HTML and Voice depending on the needs of the user. Furthermore the User Profile based on XML has the benefit of being a tree structure allowing efficient retrieval of data by the use of XQL.

To execute the route planning the ArcView application has been used.

The implementation of the route planner with ArcView has been accomplish by the use of the avenue script language providing access to the functionalities within the ArcView application. Besides giving access to the functionalities provided by ArcView, additional functionalities were designed (chapter 4) and implemented in avenue to accomplish the goal of deciding which facility location among many to use in order to plan an optimal route.

The issue of tracking the location of the mobile user is not included in this project. It is and important issue but the technology is already well establish and available. Furthermore, the use of this technology has been reported from work elsewhere [18] and [19]. In this project the client side is simulated on a desktop computer using fictive locations of the user. Therefore the task of locating the user has not been a part of this project.

6.2 Future Work

In this section we give direction towards future work. One direction is the extension of the service level of the Route Guide, another is the use of Voice Discourse and the last is movement of the road network from ArcView to e.g. an Oracle database.

6.2.1 Extending the Service Level of the Route Guide

Besides more long-term changes (road repairs, road rebuilding etc.) of the road network, other situations as heavy traffic load, queues and accidents could be included in the determination of the optimal route. Actually, the development of a database containing the information mentioned above is an ongoing work where EUMAN is participating [21].

In order to include the above-mentioned situations in the route planning, they have to be registered. This registration could be accomplish in different ways. One approach could be that people driving on a road have the ability to register the state of the road [21]. Another approach would be to allow Ambulances and Falck units to send there location when they are on the road and the traffic should be directed in another direction.

Adding information to the road network database as road repairs, traffic load, queues and accidents could provide for an extension of the service level provided by the Route Guide, in the sense that these informations are available to the users.

6.2.2 Voice Discourse

Another interesting issue of this project is to improve communication between the user and the Route Guide by using voice. In this area a lot of improvement can be made to the Route Guide, in order to handle a discourse with the user. The difficult part is the speech recognition. With that in mind the user interaction could be limited to normal use of the presented keyboard and using text-to-speech synthesis at the server-side to give route guidance in the form of voice. The ideal extension would be to allow the user to talk with the Route Guide and receive response as Voice.

6.2.3 Moving from ArcView to Oracle

In this project route planning have been implemented by the use of ArcView, a future extension would be to implement route planning on top of e.g. a Oracle database with spatial data representing the road network of interest. This would lead to a more stable and realistic system and provide for the use of XML as interface to the database instead of the DDE interface which have it's limitations. With the User Profile already implemented as a XML database and the ability to exchange data between the Oracle database and the Route Guide in the form of XML, this could result in a system were all internal representation of information to be moved between various components is done by the use of XML. Having all internal information represented as XML creates a good foundation for building a flexible Route Guide, providing transformation of the internal representation of information to various formats as HTML, WML and Voice.

BIBLIOGRAPHY

[1] **W3C Scalable Vector Graphics (SVG)**
http://www.w3c.org/Graphics/SVG/Overview.htm8

[2] **What is Oracle Spatial?**
http://www.oracle.com/products/spatial/

[3] **Nokia Programmers Guide**
Nokia Activ Server API API version 1.2 October 25 2000
Included in the nokia toolkit at:
http://forum.nokia.com/wapforum/main/1,,1_1_30_2_3,00.html

[4] **HTML 4.01 Specification**
http://www.w3.org/TR/REC-html40/

[5] **WML Reference** version 1.1,
http://forum.nokia.com/wapforum/main/1,6668,1_1_30_3_1,00.html

[6] **Deployment Guide for Corporate WAP Services**,
http://www.nokia.com/corporate/wap/pdf/
activserver_deployment_guide.pdf

[7] **Avenue scripts for accessing metadata while using ArcView**
http://www.mp.usbr.gov/geospat/mdext/avmdprog.html

[8] **JavaDDE**, Neva Object Technology Inc., 11 Charity Street, Suite A, Irvine, Ca 92612 USA, homepage: http://www.nevaobject.com

[9] **VoiceXML Forum**, http://www.voiceXML.org/tutorials/intro2.html

[10] **WAP White Paper**, When time is of the essence..., February 1999, AU-System

[11] **Fundamentals of Database Systems**, Ramez Elmasri and Shamkant B. Navathe, Second Edition, 1994, ISBN 0-8053-1753-8

[12] **XML structures for Existing Databases**, Eleven rules for moving a relational database to XML, Kevin Williams and others, January 2001, http://www-106.ibm.com/developerworks/library/x-struct/

[13] **Professional XML Databases**, Wrox Author Team, December 2000, ISBN 1861003587

[14] **ArcView help files**, ArcView GIS Version 3.1.1

[15] **Personal Agent Providing Customized Information for Nomadic Users**, Peter Vinther & Henrik Olesen, Aalbog University, Denmark, December 2000

[16] **GMD IPSI XQL Engine**, version 1.0.2, http://xml.darmstadt.gmd.de/xql/indel.html

[17] **Access FAQ, Relationship Answers**, http://www.lmu.ac.uk/lskills/TLLS/FAQs/AccessRelationshipAns.html#Many

[18] **Location Based Services - The Underlying Technology**, http://www.sli.unimelb.edu.au/research/publications/IPW/4_01Smith.pdf

[19] **Privacy vs Location Awareness**, http://www.hut.fi/ slevijok/privacy_vs_locationawareness.htm

[20] **XT**, Version 19991105, Copyright (c) 1998, 1999 James Clark, http://www.jclark.com/xml/xt.html

[21] **EUMAN**, A company that we have exchange ideas with and discussed certain issues http://www.EUMAN.com

APPENDIX A

Abbreviations

ASP	Active Server Page
CGI	Common Gateway Interface
CRG	Create Route Guidance
DDE	Dynamic Data Exchange
DOM	Document Object Model
DTD	Document Type definition
ER model	Enitity Relation model
HTML	Hypertext Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPS	Secure HyperText Transfer Protocol
ISO	International Standards Organization
NT	New Technology
PDA	Personal Digital Assistant
PHP	Hypertext PreProcessor
RP	Route Planner
Servlet API	Servlet Application Programmer Interface
SGML	Standard Generalized Markup Language
UP	User Profile
UPI	User Profile Interface
UPM	User Profile Management
UPR	User Profile Request
URL	Uniform Resource Locator
URRM	User Request and Response Management
VoiceXML	Voice Extensible Markup Language
WAP	Wireless Application Protocol
WML	Wireless Markup Language
XML	Extensible Markup Language
XQL	Extensible Query Language
XSL	Extensible Stylesheet Language
XSLT	Extensible Markup Language Transformation

APPENDIX B

Code Example: Browser Sniffer

This appendix is a code example presenting the ideas of browsersniffing. The code is based upon the BrowserHawk4J (JavaBean) package from cyScape Inc. (http://www.cyspace.com/company/) for use on any platform from a server-side Java environment such as Java Server Pages, Servlets, server-side JavaScripts and Java applications. The only system requriement is a Java Virtual Machine, version 1.1 or higher.

The BrowserHawk4J package provides classes with functionality enabling e.g. a servlet to detect information as: Browser, Cookie, JavaScript, JavaApplet, WAP, Personal Digital Assistant (PDA), Connection speed, screen size resolution on client, Reverse DNS Lookup and Client Operating System etc.

BrowserHawk parses the user agent on the client platform, in order to dynamically determine platform, operating system details and version information.

The following code is an example of a browsersniffer servlet that detects information by using the BrowserHawk.getBrowserInfo(req) function.

```
import java.lang.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.cyscape.browserhawk.*;

public class browsersniffer extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    res.setContentType("text/html");
    EncryptResponse encryptResponse = new EncryptResponse(req, res);
    PrintWriter out = encryptResponse.getWriter();
    BrowserInfo b = null;
    try {
      b = BrowserHawk.getBrowserInfo(req);
    }
    catch (BrowserHawkException e) {} // Runtime error, here.......
    String currFileName = "browserResult.txt";
    try
    {
      // Open a file of the current name.
      File file = new File (currFileName);
      // Create an output writer that will write to that file.
      FileWriter fileout = new FileWriter(file);
      // Returns the common name associated with the
      // browser such as "Netscape" and "IE"
      fileout.write(b.getBrowser());
      // Returns the entire version of the browser,
      // including all major and minor numbers and letters, if any.
      fileout.write(b.getFullversion());
      // Returns details on operating system the visitor is using.
      fileout.write(b.getOSDetails());
      // Returns more general information (as compared to getOSDetails())
      // about the user's platform.
      fileout.write(b.getPlatform());
      // Contains the model of the WAP device if known.
      fileout.write(b.getWAPDeviceModel());
      fileout.close();
    }
    catch (IOException e) {}
  }
}
```

APPENDIX C

Service Concept: A Philosophical View

In chapter 3 the concept of service was introduced as a entity type called Service. Gas Stations of various kinds was used to illustrate the idea of a service the nomadic user would like include as part of his Route. In this chapter we will take a philosophical view on the service concept and focus on problems the service concept might introduce.

C.1 Service Concept

Imagine a service called Musical, the nomadic user is allowed to specify in his user profile that he has an interest in musicals. This service can be interpreted in may ways, lets look at Musical service in the following way:

Algorithm C.1.1

1. *If an entry exist in the user profile called Musical then*
2. *When nomadic user pass by a musical unit on his route from A to B then*
3. *Notify user, by displaying a message on the screen of the mobile device*
4. *And retrieve, upon request, additional information about the musical unit.*

It is not a very complex algorithm at the first glance, it is easy to read and cope, but when we look closer, many problems will occur if one tries to implement this algorithm.

C.1.1 Musical Service: Problems

The first line of the algorithm is fairly easy to implement, one just have to locate the user profile, make e.g. an SQL query like this:

select *service name*
from *service*
where *service name* = *musical*

Problem 1 The ″pass by″ phrase in line two of the algorithm introduce some problems. First of all what is meant by the phrase ″pass by″, is it when I pass by a musical house within 10, 100 or 1000 meters (distance limit). The ″pass by″ phrase should somehow be converted into a measure, the difficult part is to find the right measure, a measure that is common enough to fit most situations and accurate enough to be useful. Imagine a distance limit at 1000 meters, how does the algorithm cope with the distance limit when I am in Copenhagen or in London, clearly a distance limit at 1000 meters is to much when I am downtown London, but okay when I am downtown Copenhagen.

Problem 2 Second how should a museum with an musical exhibition be treaded when I pass by the museum. The problem here is how to define Musical as a service or more precisely how to limit the amount of information that can be associated with a service like Musical. Should the nomadic user only receive information when he pass by a Musical House or does the word musical also include exhibitions, events and written material about Musicals.

Okay, one approach could be not to allow a nomadic user to define a service name as broad as Musical, only allowing the user to specify musical names like Cats, Panthon of the Opera etc. But on the other side, how should I as first time tourist in London know anything about musical names in London.

Problem 3 Furthermore, how does the Route Guide discover that the nomadic user is about to pass by a Musical House on his route through London. The information must be available online on the Internet, and the never ending circle of problems expands with a new problem; Where does the Route Guide locate the information. One approach could be to have information agents moving around looking for information, introduction the problem of inter-agent communication and cooperation [15]. A more feasible solution could be a centralized information source for London providing all necessary

information about musicals. Clearly this information source have to contain geographical coordinates of each musical unit, so that these coordinations can be compared to the location of the nomadic user in order to activate line two and three of the algorithm.

C.2 Service Concept: Conclusion

The above scenario illustrates some of the problems associated with the idea of providing additional services as a part of e.g. a Route Guide. The first problem concern how to make a correct measure for the ″pass by″ phrase of the algorithm, in order to make it possible for the Route Guide to perform checks that allows it to register when a user is passing by a relevant service. One approach could be to compare GPS locations, simply compare the current nomadic user GPS location with the stored location of the service, if they match the Route Guide can conclude that the nomadic user is passing by the service of interest. But then assume a Mall with eleven stores and a food court. One of the stores sell books about musicals online on the Internet and the food court have special food deals online as well. If a user have defined musical and food deals as services in his user profile, a new problem occur. Both services have the same GPS location, and the Route Guide will have two services that match the location of the nomadic user. Then, should the nomadic user have information about both services, if so, what happens when the Route Guide find five services that match the same nomadic user location.

The second problem concerns how to chose the right service name, so that it a the same time describes the service and makes it possible to limit the amount of information that the nomadic user should receive.

The last of the three problems that we have chosen to focus on, concerns how to discover information. Should this be done by mobile agents moving around the Internet looking for information, or by dedicated web sites that accumulate specific information about a few services.

The three problems above helps to illustrate the difficult task that one face when trying to provide services. On the other hand if these problems can be solved one stands at the edge of creating something new. Something that will change the use of mobile devices and enhance the all ready widely use of the Internet with something new.

APPENDIX D

Final XML Database Schema

The final XML database schema derived from the relational database schema from section 3.3.1 by using eleven rules from [12][13].

<!ELEMENT UserProfile (RouteType*, Route*, NomadicUser*, Service*,
RoutePoint*, RouteHasService*, NomadicUserOwnRoute*,
RouteConsistOfRoutePoint*) >
<!ATTLIST UserProfile>
<!ELEMENT RouteType EMPTY>
<!ATTLIST RouteType
RouteTypeID ID #REQUIRED
RTname CDATA #REQUIRED>
<!ELEMENT Route (RouteHasService*, RouteConsistOfRoutePoint*)>
<!ATTLIST Route
RouteID ID #REQUIRED
RouteName CDATA #REQUIRED
RouteTypeIDREF IDREF #REQUIRED>
<!ELEMENT NomadicUser (NomadicUserOwnRoute*)>
<!ATTLIST NomaidcUser
UserID ID #REQUIRED
UserName CDATA #REQUIRED>
<!ELEMENT Service EMPTY>
<!ATTLIST Service
ServiceID ID #REQUIRED
ServiceName CDATA #REQUIRED>
<!ELEMENT RoutePoint EMPTY>

<!ATTLIST RoutePoint
RoutePointID ID #REQUIRED
Street CDATA #REQUIRED
HouseNr CDATA #REQUIRED
City CDATA #REQUIRED
PostalCode CDATA #REQUIRED>
<!ELEMENT RouteHasService EMPTY>
<!ATTLIST RouteHasService
RouteID CDATA #REQUIRED
ServiceID CDATA #REQUIRED
ServiceIDREF IDREF #REQUIRED>
<!ELEMENT NomadicUserOwnRoute EMPTY>
<!ATTLIST NomadicUserOwnRoute
UserID CDATA #REQUIRED
RouteID CDATA #REQUIRED
RouteIDREF IDREF #REQUIRED>
<!ELEMENT RouteConsistOfRoutePoint EMPTY>
<!ATTLIST RouteConsistOfRoutePoint
RouteID CDATA #REQUIRED
RoutePointID CDATA #REQUIRED
RoutePointIDREF IDREF #REQUIRED>

APPENDIX E

User Profile Interface: Retrieve Function

The following code is a prototype implementation of the UPI function Retrieve. The Retrieve function takes a xml file name and a XML Element tagname as input parameters. Based upon the xml file the retrieve function starts by creating a Document Object Model (DOM) of the XML document. Then a simple query is executed, retrieving the id of all elements specified by the tagname. The result of the Retrieve is a array of id's (result) returned to the calling function.

```
import org.w3c.dom.*;
import de.gmd.ipsi.xql.*;
import de.gmd.ipsi.domutil.*;
import java.io.*;

class UserProfileInterface {

  static Document testDoc;

  // xmlFile = name of xmlFile to perform XQL query on.
  // tagname = name of Element to perform XQL query on.
  public Object Retrieve(String xmlFile, String tagname)
  {
    XQLResult queryresult = new XQLResult();
    Object result[] = {};
    // Generates a DOM from XML source given as InputStream
```

```
    try
    {
      testDoc = DOMUtil.createDocument();
      // testDoc = The DOM document which serves as node factory and
      // to which child nodes are added.
      DOMUtil.parseXML(new BufferedInputStream(
      new FileInputStream(xmlFile)),testDoc,true,
      DOMUtil.SKIP_IGNORABLE_WHITESPACE);
    }
    catch ( DOMParseException ex ) { ex.printStackTrace(); }
    catch ( FileNotFoundException ex ) { ex.printStackTrace(); }

    // *** Iterate over the query set ***
    // Returns a NodeList of all the Elements with a given tag name in the
    // order in which they would be encountered in a preorder traversal
    // of the Document tree.
    NodeList queries = testDoc.getDocumentElement().
getElementsByTagName( "xqltest:"+tagname );
    for ( int i=0; i < queries.getLength() ; i++ )
    {
      // Get single query and its ID
      // The node at the indexth position in the NodeList
      String query = queries.item(i).getFirstChild().getNodeValue();
      // and store id attribute value of the node in the parameter id.
      String id = ((Element) queries.item(i)).getAttribute( "id" );

      // Execute Query on sourceDoc and store the result
      // of the Query in the queryresult object.
      queryresult = XQL.execute( query, (Node) testDoc);
      result[i]=queryresult;
    }
    return result;
  }
}
```

APPENDIX F

XML Parser Servlet

Prototype implementation of XML Parser. The implementation is based upon the XT package from [20].

```
package com.jclark.xsl.sax;

import com.jclark.xsl.*;

import java.io.IOException;
import java.io.File;
import java.io.Writer;
import java.net.URL;
import java.util.Enumeration;
import javax.servlet.*;
import javax.servlet.http.*;
import org.xml.sax.*;

public class XSLServlet extends HttpServlet {
  private XSLProcessor cached;

  public void init() throws ServletException {
    String stylesheet = getInitParameter("stylesheet");
    if (stylesheet == null)
      throw new ServletException("missing stylesheet parameter");
    cached = new XSLProcessorImpl();
    cached.setParser(createParser());
```

```
try {
      cached.loadStylesheet(new InputSource(getServletContext().
                getResource(stylesheet).toString()));
    }
    catch (SAXException e) {
      throw new ServletException(e);
    }
    catch (IOException e) {
      throw new ServletException(e);
    }
  }

  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    File inputFile = new File(request.getPathTranslated());
    if (!inputFile.isFile()) {
      inputFile = new File(request.getPathTranslated() + ".xml");
      if (!inputFile.isFile()) {
        response.sendError(HttpServletResponse.SC_NOT_FOUND,
                           "File not found: " + request.getPathTranslated());
        return;
      }
    }
    XSLProcessor xsl = (XSLProcessor)cached.clone();
    xsl.setParser(createParser());
    for (Enumeration e = request.getParameterNames(); e.hasMoreElements();) {
      String name = (String)e.nextElement();
      // What to do about multiple values?
      xsl.setParameter(name, request.getParameter(name));
    }
    OutputMethodHandlerImpl outputMethodHandler = new OutputMethodHandlerImpl(xsl);
    xsl.setOutputMethodHandler(outputMethodHandler);
    outputMethodHandler.setDestination(new ServletDestination(response));
    try {
      xsl.parse(fileInputSource(inputFile));
    }
    catch (SAXException e) {
      throw new ServletException(e);
    }
    while(System.in.read() != 'q') {}
  }
```

```
static Parser createParser() throws ServletException {
    String parserClass = System.getProperty("com.jclark.xsl.sax.parser");
    if (parserClass == null)
      parserClass = System.getProperty("org.xml.sax.parser");
    if (parserClass == null)
      parserClass = "com.jclark.xml.sax.CommentDriver";
    try {
      return (Parser)Class.forName(parserClass).newInstance();
    }
    catch (ClassNotFoundException e) {
      throw new ServletException(e);
    }
    catch (InstantiationException e) {
      throw new ServletException(e);
    }
    catch (IllegalAccessException e) {
      throw new ServletException(e);
    }
    catch (ClassCastException e) {
      throw new ServletException(parserClass + " is not a SAX driver");
    }
  }

  /**
   * Generates an <code>InputSource</code> from a file name.
   */

  static public InputSource fileInputSource(File file) {
    String path = file.getAbsolutePath();
    String fSep = System.getProperty("file.separator");
    if (fSep != null && fSep.length() == 1)
      path = path.replace(fSep.charAt(0), '/');
    if (path.length() > 0 && path.charAt(0) != '/')
      path = '/' + path;
    try {
      return new InputSource(new URL("file", "", path).toString());
    }
    catch (java.net.MalformedURLException e) {
      /* According to the spec this could only happen if the file
 protocol were not recognized. */
      throw new Error("unexpected MalformedURLException");
    }
  }

}
```