Diplomarbeit

Flexible Content Management for the LoL@ UMTS Application

ausgeführt am

Institut für Informationssysteme
Abteilung für Verteilte Systeme
der Technischen Universität Wien

unter Anleitung von
o.Univ.-Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
und
Univ.-Ass. Dipl.-Ing. Clemens Kerer
als betreuendem Assistent

von

Elke Michlmayr
Brigittagasse 11/4
1200 Wien
Matr.Nr. 9402411

Wien, im Mai 2002 _____

Kurzfassung

Der mobile Touristenführer LoL@ (Local Location Assistant) ist ein *location-based service* für UMTS (Universal Mobile Telecommunications System, [Ric00]). *Location-based services* nutzen den Zugriff auf die vom mobilen Netzwerk zur Verfügung gestellten Positionsdaten, um BenutzerInnen Informationen und Dienste, die mit ihrer momentanen Position in Zusammenhang stehen, anzubieten. LoL@ wurde für TouristInnen konzipiert, die zu Fuß im ersten Bezirk von Wien unterwegs sind und ihr mobiles Endgerät verwenden, um Multimediainformationen zu Sehenswürdigkeiten abzurufen. LoL@ bietet Unterstützung beim Finden des Weges zu bestimmten Sehenswürdigkeiten: Das System kann Straßenkarten und textuelle Beschreibungen von Wegen generieren. Ein zusätzliches Feature ist das Tourtagebuch, in dem Notizen zu den Sehenswürdigkeiten gespeichert sowie Multimediainformationen gesammelt werden können.

Beim Design von Applikationen für mobile Endgeräte ist es notwendig, die Eigenschaften der Geräte (kleine Bildschirmgrößen, begrenzte Eingabemöglichkeiten, weniger Rechenleistung und begrenzte Batterielaufzeit) zu berücksichtigen. Da die Möglichkeiten der Interaktion zwischen BenutzerInnen und System durch diese Faktoren – besonders durch die Bildschirmgröße – begrenzt sind, muss großer Wert auf einfach zu bedienende Benutzerschnittstellen gelegt werden. Wichtige Anforderungen bezüglich Benutzbarkeit sind: Stabilität, konsistente Navigation, Scrolling vermeiden, Anzahl der Klicks minimieren.

Diese Diplomarbeit behandelt das Design und die Implementierung der LoL@ Server Core Applikation. Die LoL@ Server Core Applikation basiert auf XML und HTTP und stellt die *business logic* und die *presentation logic* für die Applikation LoL@ zur Verfügung. Das System ist dafür zuständig, die im Netzwerk verteilten heterogenen LoL@ Datenquellen, die Datenbestände über Sehenswürdigkeiten sowie BenutzerInnendaten (Positionsinformationen, Profile) enthalten, zu einem Ganzen zu integrieren und zu gegebenem Zeitpunkt Datenabfragen an diese Datenquellen zu richten. Die Anforderungen an das System sind Erweiterbarkeit, Integration von bereits bestehenden Datenquellen und einfache Wartbarkeit.

Andererseits muss die LoL@ Server Core Applikation eine breite Palette von Ausgabegeräten und damit auch verschiedene Ausgabeformate unterstützen können. Die Grundanforderung für Geräteunabhängigkeit ist, Inhalt (Daten über Sehenswürdigkeiten) und Form (Layoutinformation) der Applikation strikt getrennt voneinander zu verwalten. An das Gerät auszuliefernde Daten werden entsprechend den Fähigkeiten und Eigenschaften des Geräts (z.B. Bildschirmgrösse) aufbereitet. Die Aufbereitung ist der letzte von mehreren Schritten, die zur Generierung der Benutzerschnittstelle abgearbeitet werden: Regeln, die die Form definieren, werden auf den Inhalt angewendet. So werden dynamisch Daten generiert, die auf dem Endgerät angezeigt werden können. Für das Endgerät des LoL@ Demonstrators wird HTML erzeugt.

Diese Diplomarbeit wurde im Rahmen des Forschungsprojekts C1 am Forschungszentrum Telekommunikation Wien (ftw.) ausgeführt. Das C1 Projekt dient als Fallstudie für den Prozess der Entwicklung einer UMTS Anwendung.

Abstract

The mobile tourist guide LoL@ (Local Location Assistant) is a location-based application for the Universal Mobile Telecommunications System (UMTS, [Ric00]). Location-based applications take advantage of the knowledge of the user's physical position to enhance the information presented to the user. LoL@ is designed for pedestrian tourists who walk along a pre-defined tour through the first district of Vienna and interact with their mobile device to get multimedia information about tourist attractions and assistance in finding them. LoL@ can generate street maps and textual descriptions of where the user should walk to reach a desired destination. In this way, LoL@ can provide information that is tailored to the users' current location. In addition, users can keep notes and collect information in a tour diary that can be downloaded to a PC after finishing the tour.

Application design for mobile devices must consider their characteristics which impose limits on display size, input possibilities, computing resources, and battery power. The small displays of mobile devices affect the types of interaction possible and desirable. Especially for small screens and limited input methods, a clear and consistent screen design and a well-designed site structure, including navigational aspects, is crucial. Other important usability requirements are: avoidance of scrolling, low number of clicks to perform a certain action, stability, and non-modal interactions.

This thesis presents the design and the implementation of the LoL@ Server Core application, a content delivery system for heterogeneous data sources via XML and HTTP. The LoL@ Server Core application provides the business and the presentation logic for the LoL@ service. For the design of both components, Web service engineering design patterns were used. The business logic component is responsible for the integration of and content retrieval from the LoL@ data sources which are distributed over the network and provide tourist-related information as well as information about the users, like location information and user profiles. The main requirements for the business logic component are extensibility, integration of legacy data, and maintainability.

The key requirement for the presentation logic component is device independence: It must support a potentially wide range of output devices and – consequently – various output formats. Before delivering content data to the user's mobile device, it must be prepared according to the capabilities of the viewing device, such as display size. To achieve this, the application's content data must be strictly separated from any layout information. The presentation logic component, which is the last of several entities involved in the server's page generation process, applies layout information to content data and hence generates result pages that are suitable for displaying them on the client's viewing device. For the demonstrator, HTML code is used.

This thesis is carried out within the frame of research project C1 at the Telecommunications Research Center Vienna (Forschungszentrum Telekommunikation Wien, ftw.). The C1 project provides a case-study on UMTS application development.

Acknowledgements/Danke

First of all, I want to thank Martina Umlauft and Clemens Kerer for being great advisors.

Thanks to my colleagues here at ftw. I enjoy being a part of a scientific, multilingual, and multicultural community. Special thanks to Christoph Mecklenbräuker and Florian Hammer for useful hints and support.

Vielen Dank an alle Freundinnen und Freunde, die mich in den letzten Monaten unterstützt und mir Arbeit abgenommen haben. Es war schön zu erfahren, wie viele Menschen bei Bedarf für mich da sind.

Am meisten danke ich meinen Eltern Christine und Leopold und meinen Geschwistern Birgit und Anton. Für alles.

Contents

List of Figures

List of Tables

1 Introduction

After a tremendous increase in the number of subscribers to various cellular networks over the last few years, mobile voice communication is widely established all over the world. The principle of *mobile computing* is to extend this technology to allow transmission of data across cellular networks, without having to be connected to a fixed physical link. Mobile computing and its synonym *ubiquitous computing* are umbrella terms used to describe technologies that enable people to access network services anyplace, anytime, and anywhere. In these days, data transmission and information access via mobile devices are hampered by low available bandwidth and poor connection maintenance. The emerging next generation mobile network technology *UMTS (Universal Mobile Telecommunications System)* will provide data rates between 144 kbit/s and 384 kbit/s and therefore remove these shortcomings. Improvements in portable computing devices – ranging from very small notebook computers to 'ubiquitous devices' with embedded processing power – will allow full exploitation of the telecommunication networks' power.

Application design for *mobile devices* must consider their characteristics. On the one hand, they impose limits in computing resources, battery power and display size. On the other hand, mobile computing removes constraints of desktop computing: Users can walk around and thus change their location and their context. The possibility to determine the location of an individual who carries a mobile phone allows the development of *location-based services* that meet the demands of the user by providing information that is – considering his or her current location – at this moment most important for the user. Consumer acceptance of location-based services will depend on many factors. Law and market (as defined in [Les99]) are challenged to ensure that the subscriber's identity and privacy is protected. Reasonable use of context information and a strong focus on the service's usability are the key factors for success when designing a UMTS application.

The integration of powerful telecommunication networks and enhanced mobile devices creates challenges both in *software design* and *human-computer interaction design*. Software techniques must be used that accommodate the diversity and integration of devices, network services and applications. It is necessary to build manageable and scalable architectures that deliver content to a wide variety of mobile devices. *Device independence* is achieved by uncoupling application data and presentation of that data. Depending on the capabilities of the viewing device, different presentations of the same data are delivered to clients. A large body of research concerning this topic has already been done in the field of Web applications.

1.1 Goals and Scope of the Thesis

This thesis is carried out within the frame of research project C1 at the Telecommunications Research Center Vienna[1] (ftw., Forschungszentrum Telekommunikation Wien). In this project, we develop LoL@ (Local Location Assistant), a mobile tourist guide. It is a demonstrator for location-based UMTS applications. The main goal of this thesis is the design and implementation of the business and presentation logic for the proposed application. For the design of both components, Web service engineering design patterns are used.

- The *business logic* component is responsible for content management and content retrieval. It must integrate data sources like a content database or the mobile network which provides location data. For content retrieval, the data flow within the application is expressed in a form that is both human- and machine-readable.
- The *presentation logic* component must allow support for various display sizes. To achieve this, it is necessary to dedicate a separate entity of the system to this task.

The requirements are laid out in Section 1.2.

As mentioned above, the outcome of this thesis is not a standalone application, but a component incorporated in the LoL@ application. Hence, the scope of the thesis needs to be clearly defined. Chapter 2 provides an overview on the complete project which is the outcome of our group's collaborative work. The situation is the same with the human-computer interface which is presented in Chapter 3. Chapter 4 and Chapter 5 focus on my contribution: the design and implementation of the *LoL@ Server Core application* which provides the business and presentation logic as outlined above. Note that these two chapters mention components that are not within the scope of the thesis but nonetheless necessary for understanding the system. These instances are indicated by footnotes.

1.2 Requirements

Application design for mobile devices is influenced by their characteristics. Design must be done in consideration of the following factors [FZ94, W3C01]:

Input possibilities. In constrast to desktop computing environments with comfortable input devices like mouses and keyboards, mobile phones offer only limited possibilities for text input.

Display size. Because mobile phones have to be small in size, they also have small screens. Next generation mobile devices will have color displays and a sufficiently fine resolution.

[1] `http://www.ftw.at`

Storage capacity, processor power, battery. Batteries are weighty, but mobile devices must be light in weight. Battery powered mobile computers will always face power constraints relative to their fixed counterparts. The more processor power and storage capacity a device has, the more power it consumes and the shorter the battery life is. From this follows that only little computation is possible on the phone itself.

Frequent disconnections. In comparison to fixed networks, mobile networks provide less connection stability. Mobile clients do not stay connected to the network continuously: The connect speed varies, high latencies can occur, and missing network coverage can lead to connection drops.

Data Rate. With the new air interfaces designed for UMTS, it is theoretically possible to transfer data at a data rate of up to 2 Mbit/s. This high data rate, however, is offered only in quite specific environments and at low travelling speeds (approx. 10 kph). In other areas and at higher moving speeds a data rate of 144 - 384 kbit/s is available to individual users. This corresponds to between two and six times the data rate of ISDN.

In any system designed to interact with people, the people are the most important consideration. Two very basic but important factors that lay the ground for success in application design are:

Response time. The application's response time must correlate with the response times and attention spans of human beings. Users are impatient and do not want to wait a long time for the information resp. service they have requested.

Usability. Human-computer interaction design must be done carefully to provide a good user experience (effectiveness, safety, and ease of use) of the service. The device characteristics influence how information must be presented. The use of in- and output media must be optimized. This topic is discussed in Section 3.2.

Due to the rapid growth of the Internet, the development life cycle of Web applications has been greatly compressed. Web application designers often do not dedicate enough time to the analysis and design phase in comparison to the time spent for the implementation phase. This can lead to great problems in the testing, delivery, or maintenance phase. The basic requirements for Web application design are:

Extensibility. The architecture needs to support easy insertion and deletion of data resources. Web applications are never static, but are constantly changed and extended. The design needs to be flexible so that data resources and information needs not currently envisioned can be integrated in the future and a consistent set of user interfaces to these new resources can be provided.

Integration of legacy data. It is neither possible nor reasonable to develop a new application from scratch when there is an existing one that works well. Web applications must provide well-defined interfaces to integrate legacy data and legacy applications.

Maintainability. Updating content data as well as changing the graphical layout of the service must be easily and independently possible. This means that these different domains of concern must be designed as independent entities. Changes relating to one of these entities must not require changes in other entities. Normally, Web application developers and maintainers are not the same group of persons. Developers have to consider the needs of graphic designers, service maintainers, and content managers by adding interfaces to the service that are suitable for these groups.

Device independence. As the W3C puts it, "Content authors can no longer afford to develop content that is targeted for use via a single access mechanism. The key challenge facing them is to enable their content or applications to be delivered through a variety of access mechanisms with a minimum of effort." [FMS01] The basic requirement to achieve device independence is to anticipate a separate entity for the presentation logic when designing a Web application.

1.3 Organization of the Thesis

Chapter 2 provides an overview of the LoL@ service. It describes the main concepts, the terminology, the architecture and the functionality of the LoL@ service.

Chapter 3 presents the – in contrast to a desktop computing environment – limited terminal capabilities. Human-computer interaction design issues for applications targeted at devices with small display sizes are discussed. The main user interaction patterns and the graphical design of LoL@ are presented.

Chapter 4 focuses on the design of the LoL@ Server Core application. A concept of the overall design according to the requirements stated in Section 1.2 is developed and presented in detail.

Chapter 5 is dedicated to the presentation of details about the Java-based implementation of the design. Moreover, this chapter gives an overview of the tools used for implementation and briefly presents alternative solutions.

Chapter 6 describes related work and compares it to the developed approach. Existing mobile tour guides as well as content publishing frameworks with a design approach similar to the proposed system are surveyed.

Chapter 7 gives an evaluation of the application with regards to the specified requirements. A section of this chapter is dedicated to an overview of future work.

2 The LoL@ Service

This chapter introduces LoL@[1], the Local Location Assistant. First, an overview of the application's functionality is given. After that, the architecture is presented. The last section of this chapter describes a typical LoL@ interaction.

2.1 Overview

The LoL@ service demonstrates a mass-market application for the Universal Mobile Telecommunications System (UMTS, [Ric00]). The project aims at demonstrating the technical possibilities of applications based on location information and provides a case-study on UMTS application development.

Location aware applications take advantage of knowledge of the user's physical position to improve the information presented to the user. LoL@ is a mobile location aware tourist guide designed for pedestrian tourists who walk along a pre-defined tour through the first district of Vienna. This tour consists of a number of tourist attractions connected by a defined (culturally interesting and visually pleasing) path. In the LoL@ terminology, tourist attractions are called "Points of Interest" (PoIs). The tourists receive location-based information about the Points of Interest. For each PoI, the user can request information (e.g., historical and architectural description, pictures, opening hours, entrance fees, audio, video, etc.).

There are two different kinds of screens: textual information screens and graphical map screens. *Information screens* can contain menus, descriptions of PoIs, or let the user interact with the system. Mainly, users will use information screens to view information about PoIs. *Map screens* come in two varieties: an overview map and detail maps. The overview map (shown in Figure 2.1) covers the greater tour area – i.e., the complete first district of Vienna. It is used to provide the user with an overview of the tour area and the route of the tour. Detail maps give a "zoomed in" view of an area. In a detail map all PoIs are shown.

If the user wishes so, LoL@ can locate the user (via GPS[2] or cell IDs [Syr01]) and show the user's approximate location on the map, depending on available locationing accuracy.

[1] LoL@ can be accessed at `http://lola.ftw.at`.
[2] Global Positioning System

Figure 2.1: LoL@ Map Overview Screen

Users can use *routing* if they are lost, want to find a certain point (like the start of the tour or a manually selected place) or want to be guided to the next PoI of the tour. LoL@ can generate street maps and textual descriptions of where the user should walk to reach the desired destination (see Section 3.4.4). Speech output of routing information is also available.

Users can enter their own data – they can mark PoIs to remember and keep notes in a *tour diary*. The diary can be downloaded to a PC after the tour is finished. For more information about the tour diary, see Section 3.4.5.

In addition, LoL@ accepts a limited set of *speech commands*. These are used as shortcuts to save keystrokes. E.g., using the speech command "diary" will open the diary screen from anywhere within LoL@ – the user does not have to click through a series of menus to get to the diary. The speech commands available in LoL@ are documented in Section 3.4.7.

If the user's connection drops for some reason or the user decides to exit LoL@ he/she has the possibility to *resume the tour* when logging in again – the state of the application from the user's point of view is stored at the server and the user can request a reload of his/her state (e.g., screen). For more information about the resume functionality, see Section 3.4.8.

2.2 Architecture Overview

This section provides an overview of the LoL@ architecture. The overall architecture is split into three domains: the user equipment domain, the mobile network domain and the service provider domain. The separation of the domains allows an efficient, because independent and simultaneous, implementation of the demonstrator. Figure 2.2 shows the LoL@ architecture and gives an overview of the interaction between the main components.

User Equipment Domain. The terminal is the physical device the end-user carries and interacts with. For user interface/presentation functions, a Web browser is used.

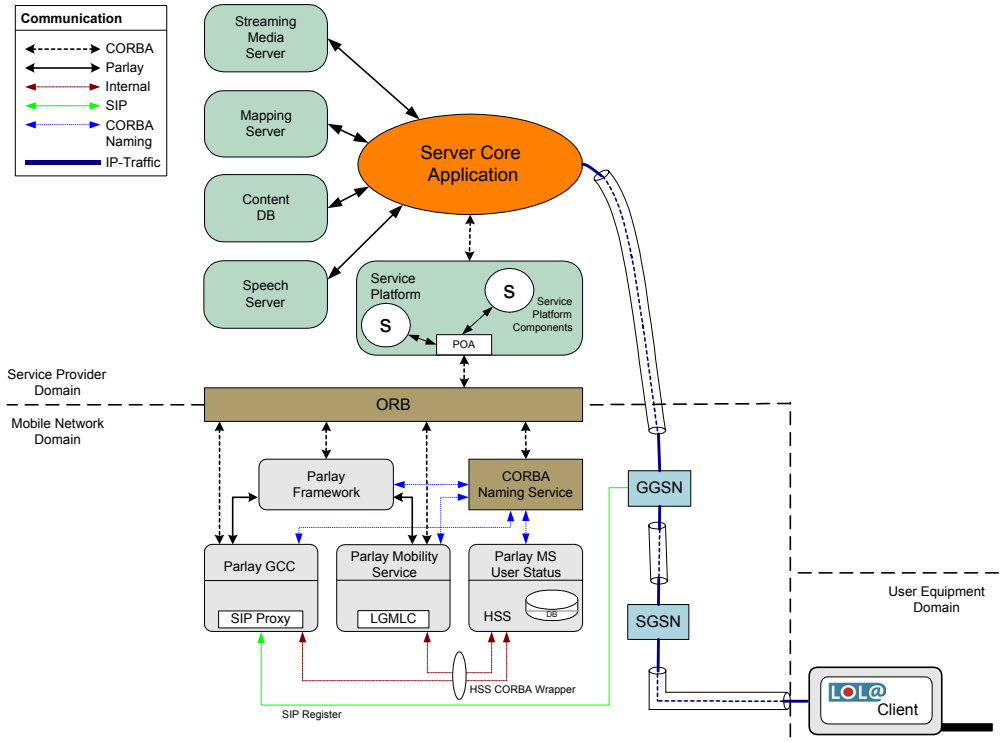

Figure 2.2: LoL@ Architecture

Other entities (not shown in Figure 2.2) are: Location Service (LCS) module [Ane01], SIP[3] User Agent [PGH01b], Speech module, multimedia presentation facilities, Userconnection module and Terminal Core component.

The LCS module is responsible for determining and transmitting the user's location. The SIP User Agent provides connection control functions. The Userconnection module listens on a well-defined, unprivileged port for messages and thus provides a simple server-push architecture. A part of application control flow is implemented at the client: The Terminal Core component controls the map viewer and the navigation buttons. For more information about the user equipment domain, see [Pos01] and [PHH01].

Mobile Network Domain. The mobile network domain includes the radio access network and the core network (GGSN[4] and SGSN[5]). The radio access network (not shown in Figure 2.2) is comprised of antennas and base stations. The core network routes IP data (calls and data connections) to external networks. The GGSN acts as the gateway to other networks such as the internet. Moreover, the GGSN also assigns

[3]Session Initiation Protocol
[4]Gateway GPRS Support Node
[5]Serving GPRS Support Node

IP adresses to terminals, similar to a DHCP server. In the implementation phase of LoL@ radio access network and core network are replaced by a Local Area Network.

All other entities shown in the mobile network domain of Figure 2.2 provide services beyond mobile voice telephony. The Parlay API [Par] specifies CORBA [Gro01] interfaces for telecommunication network functions to allow applications to access network functions (e.g., getting a location estimate for a given user) in a standardized way. A subset of the *Parlay API* is implemented in LoL@: User Status, Mobility Service and Call Control. The *Home Subscriber Service* (HSS) database [Wen01] implements User Status functionality and manages a user's subscription related information (user identification, user profile). Access to the HSS from within the mobile network domain is implemented using CORBA. The *LoL@ Gateway Mobility Location Center* (LGMLC) implements Mobility Service functions. It provides location estimates and control functions for the location subsystem (e.g., connection to the terminal and periodic request of location data). The Generic Call Control (GCC) implementation provides methods to set up and manage calls. This is done using the (underlying) *SIP Proxy* which can set up and handle connections to SIP User Agents. Services like "click2dial" use this to set up a call between two partys.

The *CORBA Naming Service* is a standard service for CORBA applications. It associates abstract names with CORBA objects and allows clients to find those objects by looking up the corresponding names.

When the GGSN assigns an IP address to a terminal, the SIP User Agent at the terminal side registers the IP address and the SIP address [HSSR99] of the terminal at the SIP Proxy. The SIP Proxy in turn manipulates the entry in the HSS database and stores the IP address for the specified SIP address there. SIP addresses uniquely identify users of the network.

For more information on the location subsystem, see [AK01]. For more information on the mobile network domain, refer to [EPC+01].

Service Provider Domain. The *LoL@ Server Core application* is the most important of all LoL@ entities in the service provider domain. It provides the business logic for the LoL@ service and is responsible for content collection, content preparation, and session management. It also implements the presentation logic for the LoL@ service.

The service provider domain contains entities which provide data and functionality used in the LoL@ service. These entities are: the streaming media server, the mapping server [BFGPU01], the content database [KPP+01], the speech server [PGH01a], and the Service Platform [AK01]. It is foreseen that other data- or functionality providing entities will be added. The LoL@ Server Core application must be designed flexibly to allow the integration of new entitites in a straightforward way.

This thesis focuses on the server domain. The remainder of this thesis will describe the parts of the LoL@ service shown in Figure 2.3. This Figure shows the LoL@ architecture from the view of the Server Core application. The radio access network

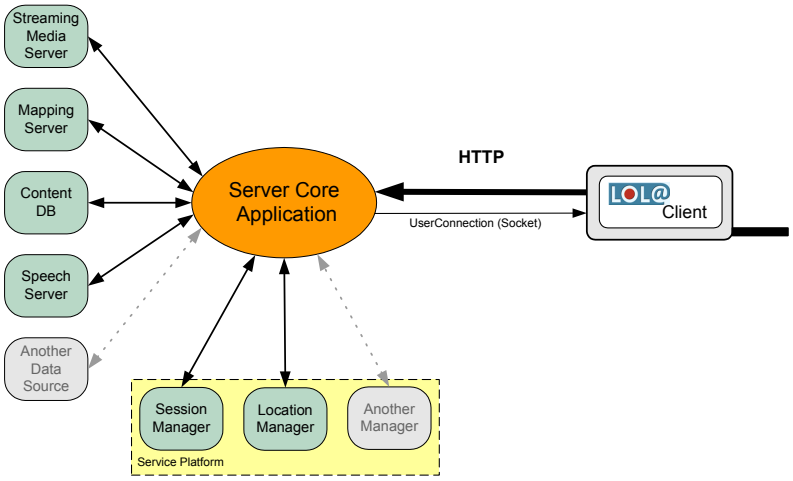

Figure 2.3: LoL@ Server Domain Architecture

and the core network are treated as a bit-pipe transporting HTTP [IGM+97] traffic. The services of the Service Platform provide an abstraction of the mobile network domain's services (e.g., getting the current location estimate or the SIP address for a user). Hence the Server Core application is basically the middleware of a Web application. Unlike a traditional Web application, it is enhanced by the features provided by the Service Platform and the possibility to send data asynchronously via the Userconnection module.

For more information about the LoL@ architecture in general, see [EPC+00].

2.3 A Typical LoL@ Interaction

This section presents a sample LoL@ interaction. To simplify the matter, entities in the system which are not involved in this interaction are not shown and error conditions are omitted. In this scenario, it is assumed that the user requests a LoL@ screen to get information about a PoI.

Figure 2.4 shows what happens when the user clicks on a link. In (1), an HTTP request containing parameters is sent to the server. The parameters of the request are sent to the Session Manager (2) for storage[6]. The core component requests the page from the cache (3). If the page is already stored in the cache, the procedure continues with step (9). If this is not the case, the page generation process starts (4).

In step (5), the appropriate XML template is fetched. The template includes instructions that tell the page generator which data sources to contact and which commands to execute

[6]This is needed for later retrieval in case the user requests to resume a tour.

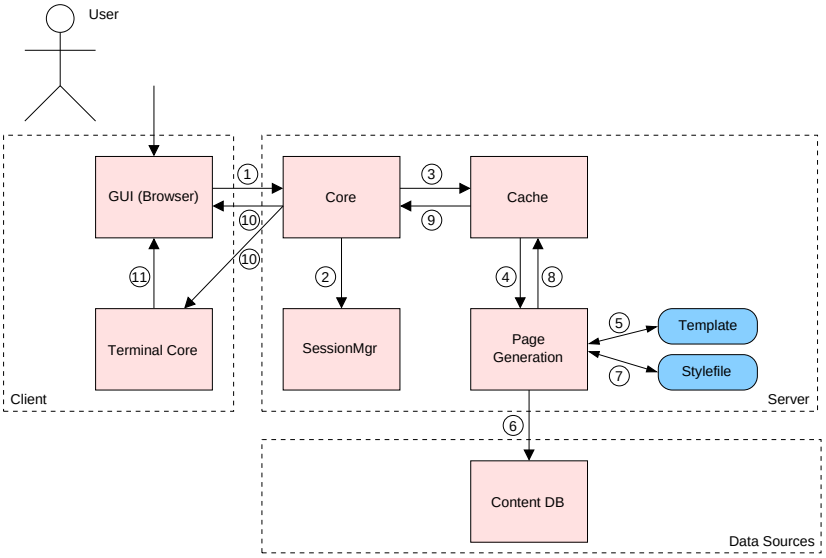

Figure 2.4: A Simple LoL@ Interaction

to get the required data. The commands are executed (6). In this sample scenario, the data is fetched from the content database. Based on the intermediate processing results, the data items in the template are dynamically changed, extended or removed. The instructions in the template are replaced with the result data of command execution. The stylefile containing layout information is loaded (7). The filled-out XML template and the stylefile are the input files for the XSL transformation step. An XSLT processor transforms the XML data according to the rendering rules defined in the XSL stylefile. In our case, the stylesheet defines HTML code. The result of the transformation is the required HTML file (8).

The caching system stores the file and sends it to the core component (9). The core component in turn sends the page to the client (10). The client's Web browser renders the HTML code and presents it to the user. The HTML file contains JavaScript instructions for the Terminal Core component (10). The Terminal Core component interprets the instructions and sets the navigation buttons according to the user's state (11).

3 User Interaction and Graphic Design

This section presents the main user interaction patterns and the graphical layout of the LoL@ service. First of all, the – in contrast to a desktop computing environment – limited terminal capabilities are presented. After that, human-computer interaction design issues for applications targeted at mobile devices are discussed. After the theoretical part, the building blocks of the LoL@ screen layout as well as LoL@'s user interface are presented.

3.1 Terminal Capabilities and Terminal Layout

UMTS devices will be available within the next year(s). The LoL@ terminal capability specification is based on assumptions on what such a device will look like. As the beginning of a convergence between personal digital assistants (PDAs) and cell phones can already be observed [For00], we assume that it will be a PDA-like phone. Since a device meeting our requirements is not yet available, we will use a notebook computer for the implementation phase.

Figure 3.1: Terminal Display Size

Portable computing devices must be small enough and smoothly enough shaped to fit in a pocket or bag. The maximum size of mobile phones is also limited by the distance between ear and mouth. The size of future mobile device's displays will be comparable with the display sizes of today's PDAs. We assume that the terminal has a display size of 320 pixels width and 120 pixels height (Figure 3.1). We further assume that some space on the device will be used to integrate physical cursor keys, therefore we do not reserve space on the

touchscreen for soft-keys[1]. A color LCD display with an integrated touchscreen is used.

Mobile phones are mainly used for voice calls, thus it can be safely assumed that the device has audio in- and output interface hardware. These interfaces shall complement the visual user interface. Systems which process combined input modes are called multimodal systems [Ovi99]. It will be possible to access the most important LoL@ screens with speech commands. Sound is of particular value where the eyes are engaged in some other task. The audio output interface will be used for speech synthesis of textual routing information – sentences like "Turn left and walk 33 meters along Naglergasse" – which are displayed at the screen and at the same time read to the user by a text-to-speech engine. Anyhow, it is possible to use LoL@ only through the graphical user interface (GUI). The other interfaces can be used optionally.

As outlined in Section 2.2, parts of the application logic will be implemented at the client. Hence, the terminal must be able to execute applications. We utilize a MExE-compliant mobile device. MExE (Mobile Station Application Execution Environment, [MEx01]) builds on incorporating a Java Virtual Machine into the mobile device and specifies a Java-based execution environment for mobile devices. MExE consists of three Classmarks. MExE Classmark 1 is based on WAP [WAP]. MExE Classmark 2 is based on PersonalJava [PJA00]. Finally, MExE Classmark 3 is based on the Java 2 Micro Edition CLDC[2] and MIDP[3] environment [J2M00].

LoL@ is designed for a MExE Classmark 2 phone. The MExE Classmark 2 standard is based on the PersonalJava application environment and the JavaPhone extension [JAP00]. PersonalJava is derived from JDK 1.1.8 but includes only a subset of the JDK 1.1.8 APIs. The PersonalJava Virtual Machine adheres to the same specification as the Enterprise Java Virtual Machine [J2E01] but its implementation is tuned for resource-constrained devices. The JavaPhone API is a vertical extension to PersonalJava and provides features like direct telephony control, user profile access and power control. For the part of LoL@ that will be executed at the client, we need `java.applet`, `java.awt`, `java.awt.event`, `java.awt.image`, `java.io`, `java.lang`, `java.lang.reflect`, `java.net`, `java.text` and `java.util`. All of these packages are included in the PersonalJava API.

3.2 Human-Computer Interaction with Mobile Devices

Human-computer interaction (HCI) is a discipline concerned with the design, evaluation, and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them [HBC+92]. Usability is defined as the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component [IEE90].

[1] To emulate the physical keys on the device, a keyboard region (on the larger laptop display) beneath the emulation of the display is used.

[2] Connected Limited Device Configuration

[3] Mobile Information Device Profile

Obviously, a big screen leads to better usability than a small screen [Nie99]. Therefore, especially for small screens and limited input methods, careful screen design and a well-designed site structure, including navigational aspects, is crucial. User interface design for applications targeted at mobile devices differs from user interface design for Web applications. The smaller displays of mobile devices affect the types of interaction possible and desirable. The device characteristics do not just alter how information should be presented but they also affect the style of user interaction [BFJ+01]. They affect the user experience and the commercial viability of the systems.

The experience with usability of applications for WAP devices reported in [Nie00] demonstrates the importance of user interface design for small devices. In comparison to WAP devices with monochrome and even smaller screens, no possibility to use multimedia and the restricted scripting language WML [WML], the capabilites of the (not yet available) device used for LoL@ are quite rich but still not plenty. The use of output media must be optimized and the new features of telecommunication networks, like location awareness, must be offered to the user to outweigth the limitations of small display sizes.

The following human-computer interaction design issues were deemed most important when designing LoL@'s user interface. [PUM02] and [UPNM02] discuss the design issues and the design process in more detail.

Clear and consistent screen design done with simplicity and flexibility in mind is the basic criteria for effective HCI design. Together with simple and consistent navigation techniques, this reduces cognitive load [Tuo97]. Each feature must look and work the same way as the other features. Consistent organization of various display features, such as screen titles, content sections, and control options (navigation items) is necessary.

Stability. Users expect the interface to remain the same unless they change it. Buttons must stay in the same place. When the user changes the state of the application, all user selections done before must remain valid. For example, if the user selects a PoI in a map screen and zooms the map in afterwards, this selection must still be active.

Scrolling must be avoided whenever possible. Horizontal scrolling is not acceptable. Vertical scrolling will only be used in screens which provide textual information (i.e., historical descriptions), but not in navigation menus.

Low number of clicks to perform a certain action. Because of the limited screen size, it is necessary to use several menu levels. This hierarchy of navigation and information items must be presented in a logical sequence. A tree-like structure is used. The navigation items are the nodes and the information items are the leafs of the tree. For example, if a user wants to view historical information about a PoI, the most logical sequence is to first select the PoI, then use a button to request information about it, and afterwards use a button to select historical information on that PoI. There is a trade-off between number of clicks and number of options in one level:

The more options to choose from in one level, the fewer menu levels are needed. The maximal number of options presented in a navigation item (i.e., menu entries) was restricted to four to improve clarity.

Appropriate colors. LoL@ will be used outdoors where the lighting conditions are potentially not very good. Sunlight could cause bad display contrast. Text must have a good brightness contrast with its background. The colors chosen for selected and un-selected items must be far enough apart.

Non-modal interactions. Application windows are either modeless or modal. A modal dialog will prevent the user from having any other interactions with the application until it has been attended to and dismissed. In contrast a non-modal dialog does not prevent the user from interacting with the application. LoL@ uses just one application window in which all screens are displayed. Hence, screens which have to be "completed" before any other actions can be taken must be avoided. The map screens and textual screens are two differing "views" but not two application modi. If a PoI has been selected in the textual screen, this selection is also relevant for the map screen, and vice versa. Once a PoI has been selected in one of the views, switching between them is always possible. Section 3.4.3 explains this feature in greater detail.

An important question is how to make use of the available location information. As relevance of information is closely related to location, it would be possible to use a push information model approach. Based on the user's current location and other context information, such as user preferences, the system would be able to come to a decision about what information is currently most important for the user according to his or her context. Based on the experiences reported in [CDMF00], we decided against this approach. LoL@ does not change or restrict the information presented to the users due to their location. Location awareness is used in the sense that by default, the user's current location is the selected PoI. Choosing another PoI and requesting information about it is always possible. The decision about what information to retrieve is taken by the user without intervention on part of the system. This is a pull information model approach. An advantage of this approach is that it makes it possible to use LoL@ (in a limited way) "offline" when not actually walking the tour.

Privacy of the location information is a very important question in combination with location awareness [Lev]. We do not use proximity awareness (automatic sensing of device-carrying persons who are nearby, [BDFR99]), and LoL@ will not send location information to other terminals. Positioning is not turned on by default: Users have to press the positioning button before their location is determined[4].

As [CMD01] states, "When designing any technology to support a tourist's exploration of a city, one needs to recognise that the primary aim of the technology should be to assist the

[4]Although LoL@ does not know the position of the user when positioning is turned off, the network operators always know at least the serving cell and thus the approximate location of the user as soon as the terminal is logged on to the network.

visitor in experiencing the real city." An initial evaluation of the GUIDE system [CDM+00] revealed that visitors generally enjoyed using the system to explore the city and did not focus too much of their attention on interacting with the system itself. This is especially true for leisure time applications, such as LoL@.

3.3 Screen Layout: The Building Blocks

In this section, the LoL@ screen layout is decomposed into functional items. There are three basic building blocks:

- *Buttons* that provide application control options.
- *Map screens* that show the geographical position of the tour and the PoIs belonging to it.
- *Textual screens* providing information to the user.

Figure 3.2 shows the structure of a screen. The service control buttons, embodied as squares with a side length of 30 pixels, are located on the left and on the right side of the display. Textual and map screens are interchangeably displayed in the center part of the display which has a width of 260 pixels.

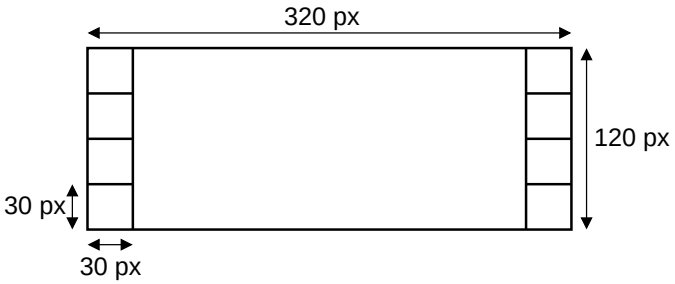

Figure 3.2: Screen Structure

3.3.1 Service Control Buttons (Soft Keys)

The buttons on the left side are the main navigation buttons and are always shown. Appearance of the buttons on the right side depends on the currently accessed screen (i.e., differs in textual and map view). If the functionality a button provides is not available at the moment, this button is "greyed-out" to indicate that this feature can not be used at the moment. Table 3.1 shows the control buttons and summarizes the initiated actions when they are clicked by the user.

Icon(s)	Button name	Action
 	Main Menu	Display LoL@ "Main Menu" screen resp. close application when already in main menu.
	Help	Show help screen for active screen.
 	Map/Information	Switch from text to map view resp. from map to text view.
	Back	Go back to the last screen.
 	Positioning	Turn positioning on resp. off.
 	Routing	Start resp. stop routing.
	Next	Request next routing information item.
	Found it!	Notify the system of arrival at a PoI.
 	Zoom	Zoom the map in resp. out.
	Diary	Open tour diary.
	Save	Save diary entry.

Table 3.1: Control Buttons

Realization

The buttons are part of the LoL@ Terminal Core component[5] and are implemented as Java applets. When LoL@ is started on a terminal, a JAR file containing these applets is downloaded to the terminal.

3.3.2 Map Screens

Two map scales are used. Figure 3.3 shows the overview map (A) used for general orientation, and an exemplary detail map (B). PoIs are grouped into regions of interest which are displayed in the overview map. Major sights, like Stephansplatz (St. Stephen's Cathedral), are displayed using dedicated symbols while minor PoIs are displayed with a category symbol only. In a detail map, all PoIs in the current region are shown. A PoI is selected by clicking on its icon in the map. Object-oriented user interaction is used. In the first step, a user selects a PoI. In the second step, an action may be invoked on the selected object.

Maps show the user's location (red crosshairs symbol in (B)) if positioning is turned on. In addition, the map is augmented with tool tips showing PoI names (see Figure 3.10, picture (1)) and routing information (see Figure 3.10, picture (6)).

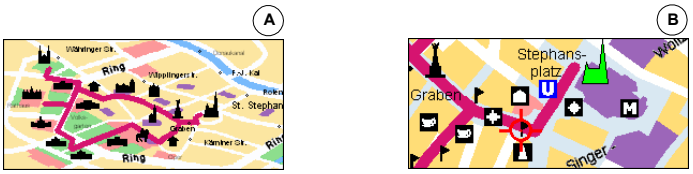

Figure 3.3: Overview Map Screen and Detail Map Screen

Realization

The map viewer is a 3rd party product. It is a Java applet and supports the display of ActiveCGM files [ACG]. It can be controlled by a set of public Java methods and by JavaScript commands. This is done by the LoL@ Terminal Core component. The map viewer classes as well as the Terminal Core component classes are included in the JAR file downloaded at startup.

3.3.3 Textual Screens

A screen contains two elements: a header element and a content element. The header element contains a title and optionally contains icons. Figure 3.4 shows the basic structure of a LoL@ textual screen.

[5]The Terminal Core component is not within the scope of this thesis.

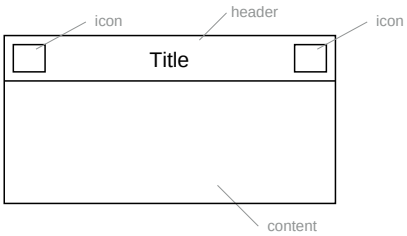

Figure 3.4: Structure of a LoL@ Textual Screen

Figure 3.5 shows the basic layout definitions for textual LoL@ screens. They are categorized into four basic layout definitions:

Screen with menu-buttons. This layout definition consists of a screen with buttons. This definition is used for screens that contain navigation buttons to other screens. The button in the bottom right corner is optional.

Text screen. This layout definition is used for all screens which present textual information to the user.

List screen. This layout definition is used for screens that contain lists. Each list item is a link to another screen.

Generic screen. In order to avoid restrictions, we also employ a "generic" screen design which can be used to describe screens that don't follow the definitions discussed above. It's a non-mandatory option for the generic screen design to have a header element. This layout definition will be used for pages which cannot be categorized in one of the 3 layouts above – e.g., screens that present images, videos, or audio files.

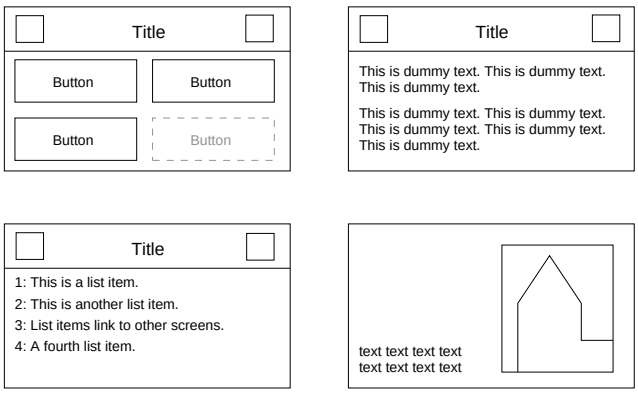

Figure 3.5: Basic Layout Definitions

Realization

The textual screens are implemented as HTML pages and generated by the LoL@ Server Core component. See Chapter 4 for a detailed description of the generation mechanism.

3.4 Program Flow and User Interaction

This section explains the main LoL@ usage interaction patterns by providing screen shots appropriate to show the most important use cases. Section 3.4.1 shows the "look and feel" of LoL@ after starting the application and documents the basic usage interaction pattern. Section 3.4.2 describes how to get information about PoIs. Section 3.4.3 explains how to switch back and forth between text and map view. The usage of the LoL@ routing mechanism is introduced in Section 3.4.4. Section 3.4.5 presents the tour diary. As Section 3.4.6 demonstrates, tour diaries can be downloaded to a PC after walking the tour. Section 3.4.7 describes the usage of speech commands. Finally, Section 3.4.8 shows how to use the resume tour functionality.

3.4.1 Main Screens

Figure 3.6 shows the LoL@ *Main Menu* screen (2) and the subsequent screens. When LoL@ is started, a welcome splash screen (1) is shown. After clicking on the image the *Main Menu* screen (2) is loaded. Users have the possiblity to either select a tour, view/change their preferences, or view/manipulate the tour diary.

Select Tour brings up the list of available tours (3). Only the first one, *Vienna's Glorious Past*, is implemented at the moment [FK00]. After selecting a tour, the list of PoIs (6) is shown. This list is most important when users are not actually walking along the tour, but planning it (i.e., in a hotel room). The list gives users the possibility to obtain more information about a certain PoI by clicking on its name. After clicking, the *PoI Information* screen (7) is shown. This screen is explained in Section 3.4.2. In screen (6), users can also choose to start the tour. In this case, the *Overview Map* screen (8) is shown. This function is most important when actually walking the tour. Usage of the *Overview Map* screen is described in Section 3.4.4.

Options opens the *Preferences* screen (5). Users can choose if they want to enable *Voice Routing*, and how long they want the map information textbox to appear before it vanishes. If *Voice Routing* is enabled, the textual routing information is not only presented on the screen, but additionally read aloud. The map information textbox is shown in picture (1) of Figure 3.10. The *Map Information Duration* time is measured in milliseconds. If set to `-1`, the textbox does not vanish. By default, *Voice Routing* is set to "yes" and *Map Information Duration* is set to `2500` ms.

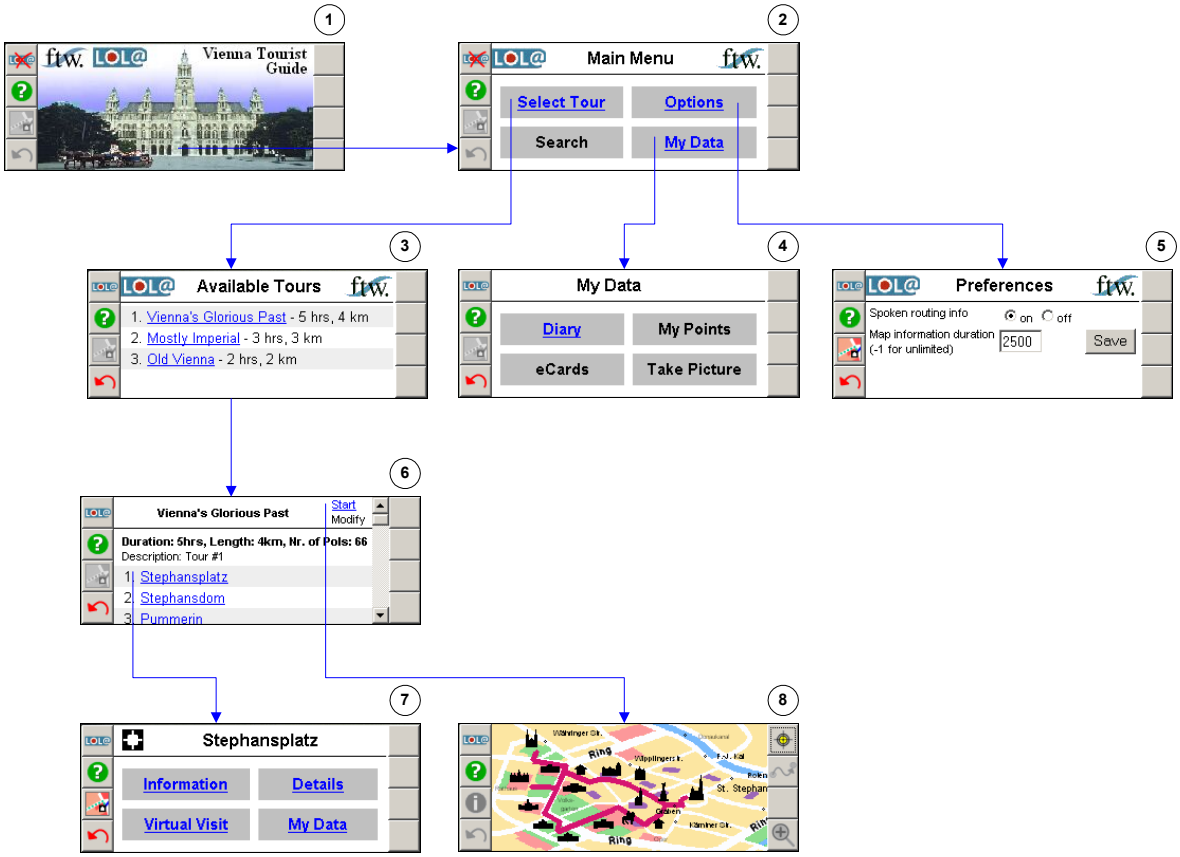

Figure 3.6: Storyboard – *Main Screens*

My Data Selecting *My Data* opens the *My Data* screen (4). This screen gives access to the tour diary which is explained in Section 3.4.5. The other options (*My Points*, *Take Picture*, and *eCards*) are not yet implemented.

Search is not yet implemented.

3.4.2 Information Screens

The screens shown in Figure 3.7 and 3.8 provide multimedia information about PoIs. The *PoI Information* screen (1) in Figure 3.7 is accessed from the list of PoIs (see Figure 3.6, picture (6)). It can also be accessed from the map screens (see Section 3.4.3 for details). The *PoI Information* screen (1) provides four options: *Information*, *Details*, *Virtual Visit* and *My Data*.

Information After selecting *Information* (2), users get general information about the PoI: a brief description, address, contact information (telephone number, fax number, email address), opening hours and entrance fees.

Details Selecting the *Details* button brings up the *Details Menu* (4). It gives access to detailed historical information (7), architectural description (see Figure 3.8, (2c)), list of events (see Figure 3.8, (2a)) and multimedia data (3). Historical (7) and architectural information of a PoI can contain links to related PoIs. By clicking on these *See Also*-links, users can quickly get related information.

My Data As outlined earlier, this button is used to open the tour diary. See Section 3.4.5 for details.

Virtual Visit The *Virtual Visit* button in the *PoI Information* screen (1) and the *Media* button in the *Details Menu* (4) provide access to multimedia information about the PoI. There are three kinds of multimedia information available: photos (5), audio files, and videos (6).

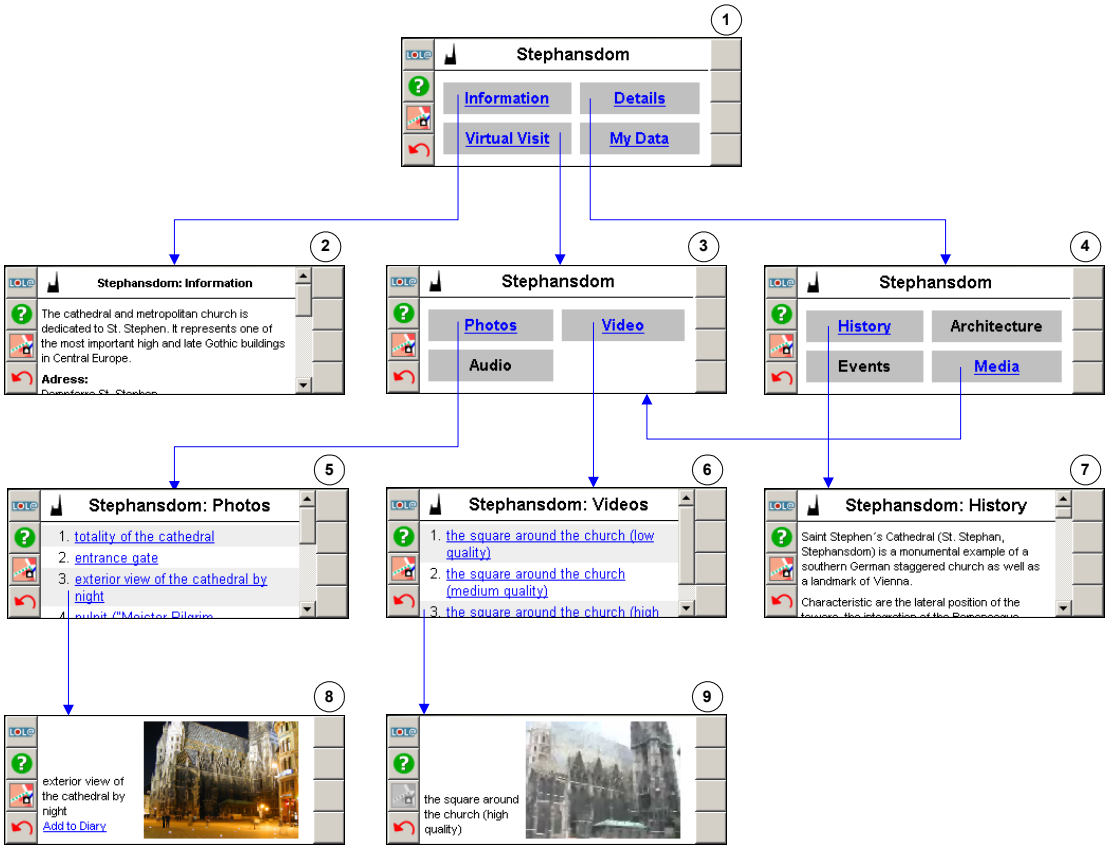

Figure 3.7: Storyboard – *Information Screens*

If the multimedia data (3) is accessed via the *Media* button in the *Details Menu* (4), all available files are displayed. When the screen is accessed via the *Virtual Visit* button in the *PoI Information* screen (1), the lists of photos, audio files, and videos are filtered and only the most interesting items are displayed. This is intended for

use by visitors who are not yet walking the tour but want to get a fast impression if they like a PoI in order to plan which PoIs to visit and which not.

An example of a photo is shown in (8). An example of a video is shown in (9). Access to audio files is shown in Figure 3.8, picture (3b).

As can be seen in (8), it is possible to add a photo to the tour diary. This functionality is explained in section 3.4.5.

If there is no data for an item, no link is presented to the user. However, screen layout stays constant to avoid many different screen designs. In the case of PoI *Stephansplatz* (St. Stephen's Cathedral, shown in Figure 3.7), there are no audio files, no architectural description and no events available for this PoI. In order to complete the summary of all LoL@ screens related to PoI information, these screens are depicted in Figure 3.8.

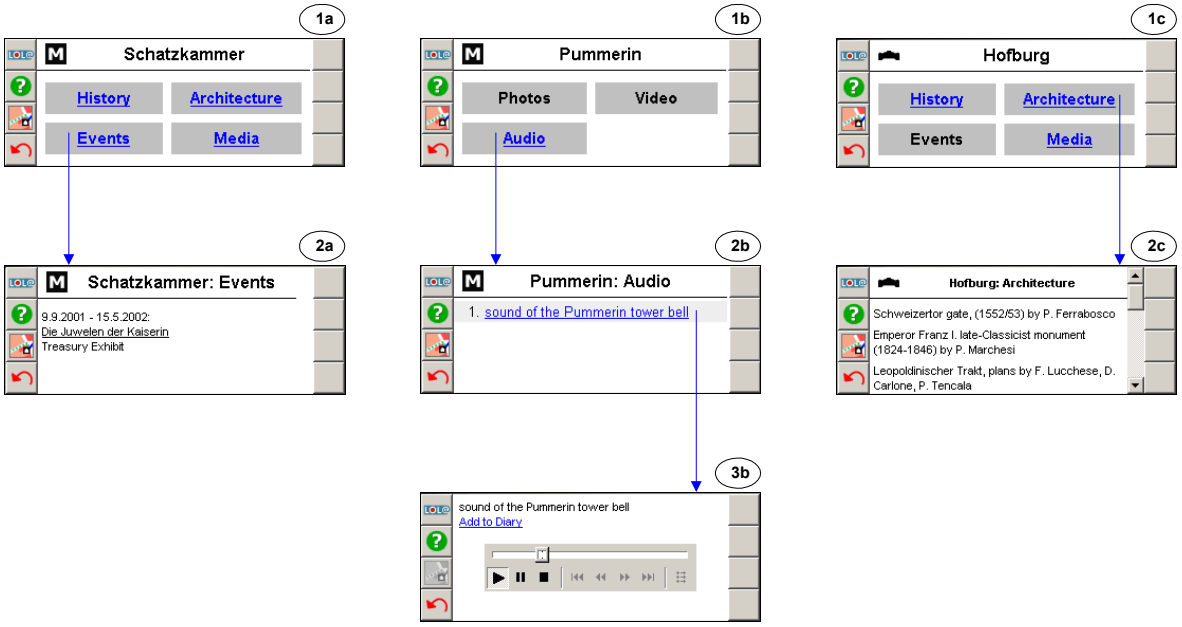

Figure 3.8: Storyboard – *More Information Screens*

3.4.3 Switching Between Textual Screens and Map Screen

Figure 3.9 shows how to switch between a textual screen and the map screen. The *Information* button is visible in map view and is symbolized by a green sign with the letter "i" printed on it. The *Map* button is visible in text view and is symbolized by a little map (in light red color). It is always possible to switch views when the *Map* button (resp. the *Information* button) is active. Switching from map to text view allows the user to get information about the PoI he or she is located at. It is also possible to select a PoI in

the map screen by clicking on it and to use the *Information* button afterwards to switch. Switching from text to map view permits the user to see the location of the current PoI in the detailed map.

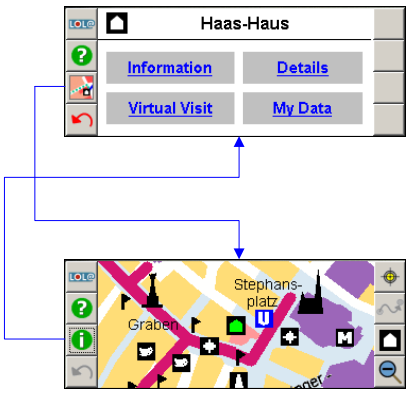

Figure 3.9: Switching Back and Forth between Textual and Map View

3.4.4 Routing

Figure 3.10 shows how LoL@ can be used to find the way to a PoI. In this scenario, it is assumed that the user's current position is not yet known by the system.

Select destination PoI The *Overview Map* screen (1) is displayed after selecting *Start Tour* in the list of PoIs (see Figure 3.6, picture (6)). In the *Overview Map* screen, the most important PoIs of the tour (displayed as little icons of buildings) can be selected by clicking on them. Selection is indicated by highlighting the affected PoI in green color and also by an appearing textbox which vanishes after some time[6]. PoIs are unselected by clicking on their symbol again or by selecting another PoI. In picture (1), the PoI *Stephansdom* (St. Stephen's Cathedral) is selected. After selecting a PoI, the map can be zoomed by using the *Zoom* button to see the detail map of the region around the selected PoI. The *Zoom* button is located at the right bottom of the screen and represented with a magnifying-glass icon. (2) shows the zoomed-in view.

Turn Positioning on In order to use the routing functionality, positioning must be turned on. This is done in picture (2) by clicking on the *Positioning* button (symbolized by a yellow crosshairs icon) at the upper left corner of the screen. When positioning is turned on, the position of the user is displayed in the map (big red crosshairs icon) and the *Positioning* button is displayed in a pushed-in manner (see picture (3)). Routing is possible now, therefore the *Routing* button (located below the *Positioning* button) becomes active.

[6]This time interval can be defined in the user's preferences, see Section 3.4.1.

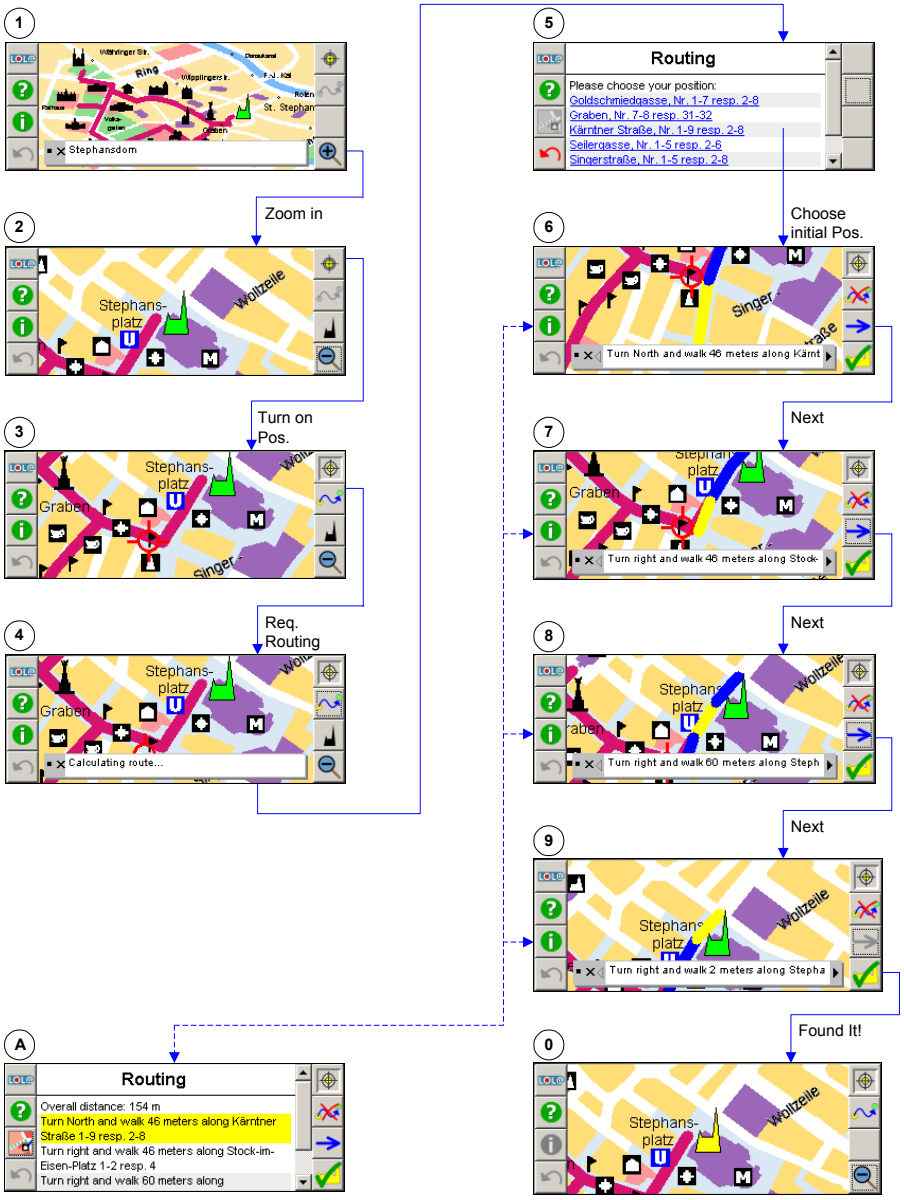

Figure 3.10: Storyboard – *Routing Screens*

Request Routing Routing is requested in picture (4) by clicking on the *Routing* button. This will cause the system to calculate the route to the destination PoI. The destination PoI is the currently selected PoI. If no PoI was selected, the destination PoI is either the next PoI of the tour or – if the tour was just started and hence the routing functionality was not used before – the first PoI of the tour. (The steps shown in (2), (3), and (4) can also be performed in the *Overview Map* screen. The application will zoom in automatically after the initial position is determined (5)).

Determination of user position Routing always starts with the determination of the accurate user position. LoL@ uses a hybrid routing concept, consisting of automatic user positioning (with GPS) and user interaction because of the limited accuracy of the current location methods. If a user has confirmed arrival at a PoI previously[7], it is assumed that he/she is still located at this PoI. If the location measurement data does not violate this assumption, the starting point of the new route is the current PoI. If the measurement data does not match the location of the PoI an interactive positioning procedure is performed. This procedure is also used if there is no current PoI because the user has just started the tour, as it is the case in Figure 3.10. In such situations, the location system calculates a location estimate and an accuracy value. A list of street names within the accuracy radius is presented to the user who selects his/her current location. This is shown in (5). After this procedure, the starting point of the route is known to the system.

Routing The route to the destination PoI is calculated and presented to the user ((6), (7), (8), and (9)). Now the user gets information about how to reach the next PoI. This information is segmented into chunks of street sections. The system will display the direction, the distance (in meters), and the street name for each street section the user has to pass. If the street numbers of the street section are known to the system, they will also be included. If Voice Routing is enabled in the user's preferences, the routing information will additionally be read aloud by a text-to-speech engine.

The user walks along the calculated route and confirms arrival at every route segment by clicking on the *Next* button (blue arrow symbol). The map shows the current route segment as well as the already passed segments and the remainder of the route in different colors.

Found It! If the user arrives at the destination PoI, he/she notifies the system by clicking on the *Found It!* button (green tickmark). This ends the routing mechanism and the PoI the user arrived at is highlighted in yellow color. The *Found It!* button can be clicked at any time when routing is active. This prevents the user from having to click the *Next* button very often although he/she has already found the PoI without assistance of the system.

Alternatively, the user can get a textual list (A) including overall distance (in meters), all street names, length of segments, and directions of the complete route by clicking on the

[7] by clicking on the *Found It!* button, see picture (9) of Figure 3.10

Information button. The currently active segment is highlighted in yellow color. Clicking on the *Map* button closes this list and brings the user back to the map.

3.4.5 Tour Diary

Users can keep notes in a tour diary. The notes are stored in a database [KPP+01] located at the LoL@ server and can be downloaded at a later time (see Section 3.4.6). Three types of notes are possible: User-defined notes, Information notes, and PoI notes.

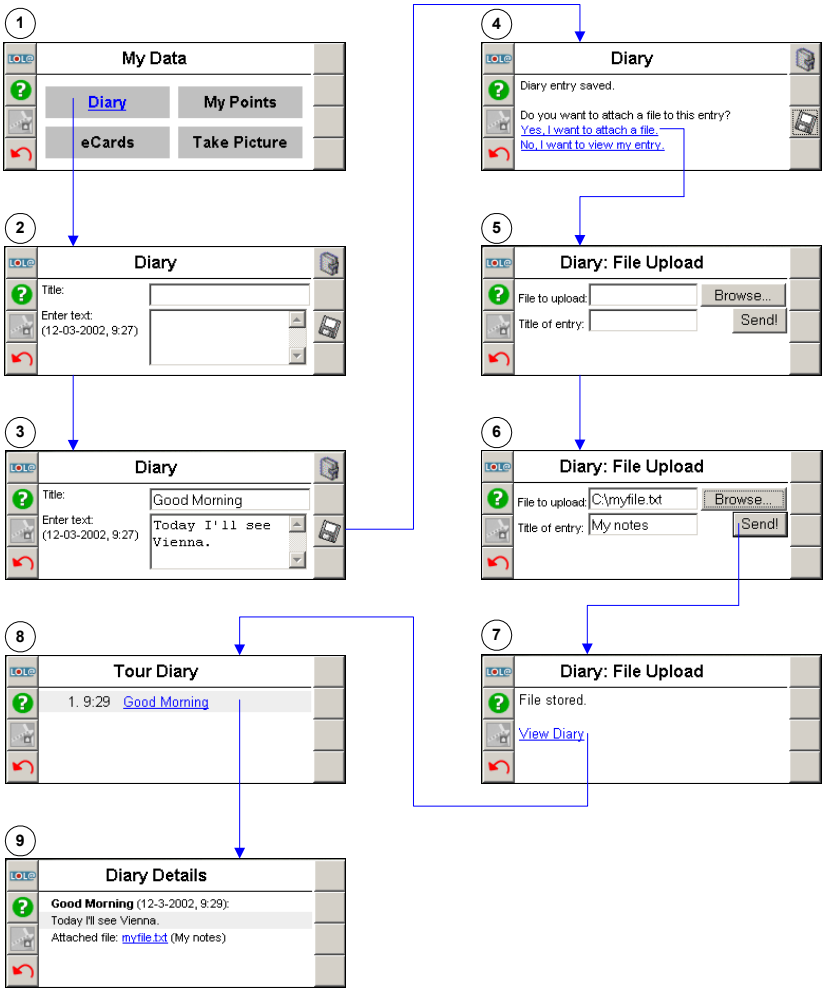

Figure 3.11: Storyboard – *Diary Screens*

User-defined notes consist of a title and a text. In addition, it is possible to upload files and "attach" them to a specific note[8]. If positioning is turned on, the current location

[8]For security reasons, each user's "quota" is 25 MB. A single file can be up to 5 MB big.

of the user is also stored. Figure 3.11 shows how to add a user-defined note to the tour diary.

The *My Data* screen (1) can be accessed from the *Main Menu* (see Figure 3.6, picture (2)) and from each *PoI Information* menu (see Figure 3.7, picture (1)). After clicking on the *Diary* button, an empty form is presented to the user (2). He/she fills out the title and the text of the note and clicks on the *Save* button (3). In picture (4), the user is asked if a file should be attached to the entry. If *Yes, I want to attach a file* is chosen, the *File Upload* menu (5) comes up. (Clicking *No, I want to view my entry* opens the tour diary for viewing (8).) Filename and title are filled in by the user (6). After clicking on *Send!* the file is stored at the server. The next screen (7) informs the user whether saving the file was successful or not. Clicking the *View Diary*-link in picture (7) as well as choosing the *Diary* button (on the top right corner of (2), (3), and (4)) opens the tour diary for viewing. A list of all entries (8) is presented to the user. Each entry can be selected for detail view (9).

Information notes Another feature is the possibility to add LoL@ multimedia files to the diary. Historical and architectural information, audio files, and photos can be collected in the tour diary by clicking on the respective *Add to Diary*-links. An example[9] of an *Add to Diary*-link is depicted in picture (1). Videos can not be added because we can not assume that the required plug-in is installed at the client later used for diary download (see Section 3.4.6). Figure 3.12 shows how to add a photo to the tour diary.

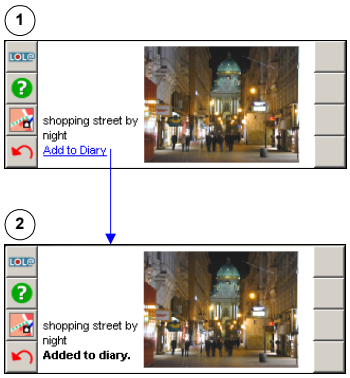

Figure 3.12: *Add to Diary* Screens

PoI notes Each time the user reaches a PoI (resp. clicks the *Found It!* button, see picture (9) of Figure 3.10), the name of the PoI and a timestamp is added to the tour diary. This way, a logbook is provided to the user which makes it possible to reconstruct the walked tour afterwards.

[9]Please note: It is planned to replace these links with a button in the right button bar in future versions.

Figure 3.13 shows the different types of tour diary notes. The first note is a user-defined note that was added in Figure 3.11. The second note is an information note that was added in Figure 3.12. The third entry is an automatically added PoI note.

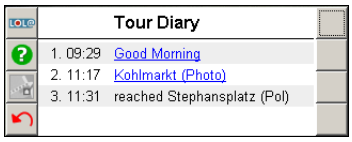

Figure 3.13: Types of Diary Notes

3.4.6 Tour Diary Download

After finishing the sightseeing tour, a user may wish to access the tour diary. It is possible to download[10] the tour diary to have a nice souvenir that brings back memories of the city and the sights that were visited, and can be shared with the family and friends who could not make the trip. The download of the tour diary will be done using a desktop computer. Hence the graphical representation of the diary is designed for a standard VGA monitor's display size and it is possible to put all diary notes on one screen rather than using a list with links. For privacy reasons, accessing the tour diary download page requires user authentication. Figure 3.14 shows an exemplary tour diary download page.

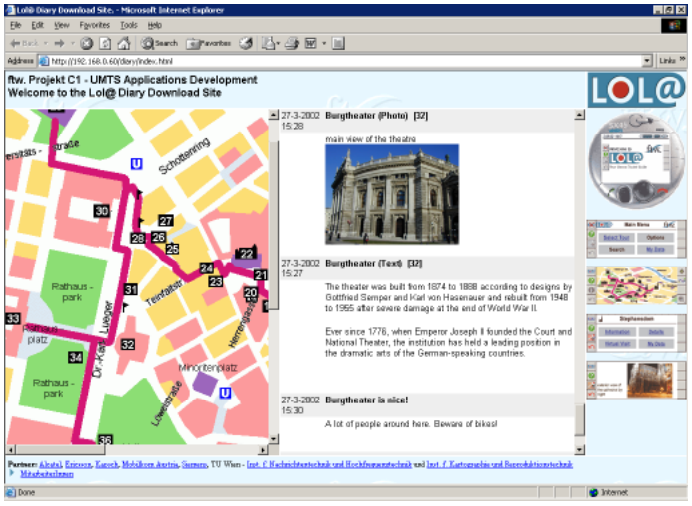

Figure 3.14: Diary Download Page

The diary notes are displayed at the right side of the screen. All linked objects that were added to the tour diary – like photos and textual descriptions – are embedded directly into

[10]The download URL is `http://lola.ftw.at/diary/index.html`.

the page. At the left side of the screen, a map showing all PoIs of the tour is displayed. The PoIs in the map are numbered rather than named. For information and PoI notes, the corresponding numbers are printed next to the diary notes. User-defined notes do not belong to a PoI.

3.4.7 Using Speech Commands

LoL@ accepts a set of speech commands. These are used as shortcuts to save keystrokes. The user has to press a button and then speak the command[11]. The commands can be spoken in english, german, or french. Table 3.2 summarizes the commands and lists the buttons they correspond to. The column "Context" defines whether this speech command can be used application-wide or restricted.

Icon	Button name	English	German	French	Context
	Help	help	Hilfe / helfen	secours aide	app.
	Map	map	Karte / Stadtplan	plan de route / de ville	app.
	Information	information	Info / Information	information / renseignements	app.
	Routing	route / routing	Wegweiser / Wegfinder / Routenfinder	route / routage	app.
	Next	next	nächster / weiter	(le) prochain / (la) prochaine	routing active
	Found it!	found (it)	gefunden / angekommen	trouve / arrive	routing active
	Diary	diary	Tagebuch	journal	app.

Table 3.2: Speech Commands

[11] We assume that this button is a physical button integrated on the LoL@ terminal.

3.4.8 Resume Tour

As [CDM+00] says, visitors should be able to interrupt a tour in order to take a (coffee) break whenever they desire. In case the user decides to exit LoL@ or the user's connection drops for some reason, it is possible to resume the application's state (as perceived by the user) at a later time.

Figure 3.15: *Resume Tour* Screen

Figure 3.15 shows that the *List of Tours* menu provides an additional *Resume Tour* item when the user starts LoL@ again after an interruption. After clicking the *Resume Tour*-link, the last screen shown to the user before interruption is loaded again.

Major information about the user's state (like the current screen) will be preserved, minor information (like the back button) will be lost. Some screens can not be resumed: i.e., the routing information screen (see picture (6), (7), (8), (9) of Figure 3.10). In this case, no *Resume Tour*-link is presented to the user and he/she has to get back to the previous state manually.

4 Design

This chapter describes the design of the LoL@ Server Core application. The requirements were presented in Section 1.2. As outlined in Section 2.2 and schematically depicted by Figure 2.3, the LoL@ Server Core application acts as a two-way facilitator:

- It integrates and manages existing data sources and services, which are distributed over the network.
- It prepares content according to the terminal output capabilities. The LoL@ terminal equipment (see Section 3.1) includes a Web browser that can render HTML code.

The LoL@ Server Core application produces all textual LoL@ screens described in Section 3.4. Moreover, it dispatches requests from the users for certain actions to the component responsible for processing it (e.g., turn on positioning). Additionally, the Server Core application provides a communication interface for the Terminal Core application and sends data asynchrounosly to the Terminal Core. As already mentioned in Section 2.2, the mobile network that provides location data and user profile data is abstracted by the Service Platform and can be treated in almost the same manner as other data sources. These facts led to the conclusion that Web application middleware is needed.

Section 4.1 gives a brief overview of the evolution of Web development tools and discusses design issues for Web applications. Section 4.2 presents the design of the LoL@ Server Core application. Section 4.3 presents the data sources and their interfaces as well as the design of the access wrapper components created. Section 4.4 details the concept of using so-called handlers for the business logic. Section 4.5 explains the design of the caching system.

4.1 Developing for the Web: A Very Brief History

In the early days of the Web, static HTML pages were used. The idea was to simply take the contents of an HTML file and transmit it over a TCP connection using HTTP communication. HTML is designed as a markup language. It allows to structure text (into headings, paragraphs, lists, hypertext links, ...). Web servers are responsible for taking a request from a client and sending the requested file as a reply. Web clients are responsible for parsing the HTML file and rendering it to the client's display. The decision about *how*

to render the marked-up text is up to the client's browser. The majority of the first Web pages provided (mostly scientific) information.

The inherent limitation discovered early on was that the Web did not allow two-way communication, and that dynamic content could not be delivered. A set of tags was added to HTML to direct a Web browser to display a form to be filled out by a user and then forward the collected data to an HTTP server specified in the form. This user input constrains or parameterizes the retrieval of the information. Additionally, the Common Gateway Interface [CGI] which was originally created as an interface to other applications can be exploited to allow dynamic content delivery. Using the CGI protocol, a Web server can pass requests to external programs. The programs are given the form data values from the Web server. The program executes and generates response data in HTML format which is then passed back to the client browser as if it were the contents of a static file. CGI quickly became a de facto standard. With HTML forms and the CGI interface, the Web became an interactive and dynamic medium. The CGI interface together with scripting languages to write CGI programs made it possible to create not just static Web pages, but dynamic Web applications. Today, dynamic elements are included in the majority of the existing Web sites.

The World Wide Web grew enormously. The open and easily unterstandable standards allowed individuals to publish information for an incredibly big audience in a cheap and easy way, without the need for a lot of technical expertise. As the total amount of Web users grew, the appearance of Web pages became more and more important. In contrast to scientists who wanted to publish their information but did not care about layout, the new Web users wanted their pages not only to inform, but also to look appealing. So HTML was not – as originally intended – used to mark up content in order to structure it, but rather to layout it in order to define what it should look like when presented on screen.

In addition, Web browser manufacturers misused their influence (e.g., market share) to "define" their own extensions of HTML – without official backing from the W3C that is in charge of defining new versions of the HTML standard. On the one hand, this led to curiosities like the <`blink`> and <`marquee`> tags, which allow text to blink and dance across the screen. On the other hand, interoperability between different Web browsers was no longer given. As a consequence, the HTML Editorial Review Board, which consists of W3C members and representatives from major browser vendors, was formed to collaborate and agree upon a common standard for HTML.

Together with HTML 4.0 [RHJ99], Cascading Style Sheets (CSS) [CSS02] were introduced to encourage the use of style sheets instead of HTML presentation elements to allow better distinction between document structure and presentation and thus solve the problem of intermixing content and layout information in a single file. Although CSS makes layout definition and adaption easier, it did not reach its goal. In addition, a part of the CSS features can not be used, because different Web browsers do not interpret CSS stylesheets the same way. Building a non-trivial site layout using HTML and CSS that is rendered the same way by every available browser is a complicated task.

Mixing content and layout in a single file makes it hard to maintain. To keep information up-to-date for large Web sites, there are tools that tackle this problem by investigating a separate entity of the system for data storage: The data is not stored directly in HTML pages, but in a data repository (e.g., a database management system) and fetched from there when a request for a certain page occurs. This makes it possible to exploit the features of relational database management systems to manage a Web site's data. Figure 4.1 shows the basic principle. There are a lot of Web application development tools (like PHP, JSP, ASP, ...) that implement this approach. All of them use a certain syntax to allow application code calls from within HTML pages. Each page of a Web application is a single file that mixes HTML code with application code calls, separated only by specified escape sequences. The application code calls do database lookups (or other actions) and insert the results of the call as HTML code into the HTML page.

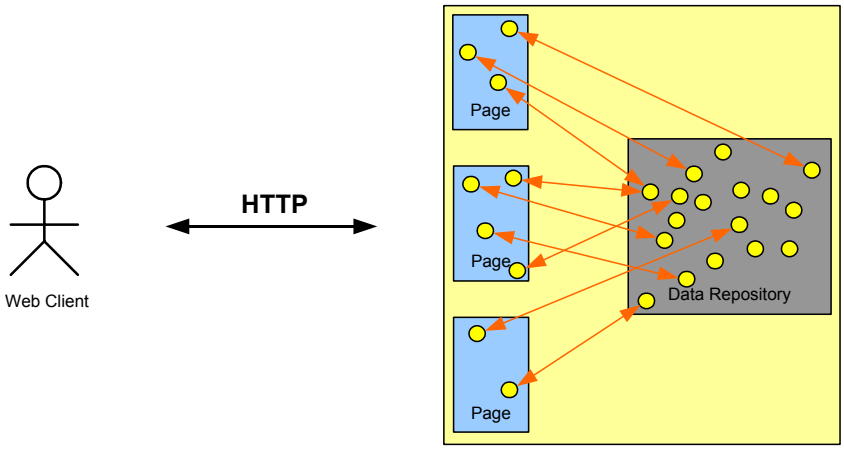

Figure 4.1: First Generation Web Application Development Tools

Obviously, these Web application development tools employ a page-oriented approach. The graphical output of the system can be seen as the central core of the whole system. The big advantage of this approach is that it is easy to learn and unterstand, but there are also many disadvantages of this approach. Since the definition of the common layout has to be replicated in every page, it is hard to maintain. In addition, changing the layout of the complete Web site is not easily possible. The most serious problem is that the Web application's logic, which is responsible for providing the site's dynamic services, is embedded into several (possibly hundreds) HTML pages.

Web applications existing today have nothing to do with the basic hypermedia systems of the early nineties. The Web has gone far beyond presenting information. Companies bring their whole business process to the Web. Web applications can potentially be – or grow to – very complex distributed systems. Scalability, both in terms of service extensions and increasing number of users, is an important factor. The complexity of Web applications will increase even more, and the page-centric model is not adequate anymore. The pages

a Web application is comprised of must be seen as what they are: an interface provided to the users for interacting with the application that is created by a complex process at the server, but not the core of the system. Web applications have to perform different tasks:

Content. Web applications have to integrate and manage several (heterogeneous) data sources, as well as legacy data and legacy applications.

Layout. The interfaces that are presented to the user must be adaptable to the user's viewing device. Moreover, an administration interface for content management and for administration of the service is required.

Logic. The distributed program flow between client and server that is necessary for dynamic services must be controlled.

The basic idea when designing a maintainable and flexible Web application is to cleary separate these different domains of concern. In literature, this principle has two different names. Cocoon (see Section 6.2.1) and MyXML (see Section 6.2.2) call it *separation of content, logic, and layout* but it is also referenced as *Model View Controller* [KP88] design pattern.

Just like using a separate component for data storage as outlined above, the system also needs a separate entity responsible for the styling of the pages. As shown in Figure 4.2, the content and the layout of a single page are stored in different places of the system and combined upon request of a client. When the layout of a site is decoupled from its content and its business logic, it is easy to define more than just one layout. This makes it possible to support different output devices (i.e., screen sizes) and different output formats (i.e., HTML, WML, PDF, ...). Obviously, more than one presentation of a certain format is possible as well: e.g., a "printer-friendly" version, or different versions that are adapted to a certain browser's features and oddities.

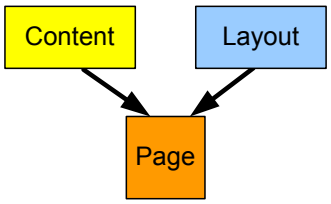

Figure 4.2: Content and Layout as Separate Entities

Beside the benefit of device independence, the advantage is that the different components can be maintained separately and that changing one of them is possible without having to change the other entity. Web applications are designed and implemented by a group of different domain experts. User interface designers and graphic designers develop the layout, application developers implement the business logic, and content managers are responsible for the application's content. Separation of domains also helps them to work simultaneously and collaboratively.

4.2 Server Core Application

In LoL@, the separation between content and layout is realized by using XML[1] [BPSMM00] for content and XSL[2] [Cla99] stylesheets for layout. The result pages are produced by XSL transformations (see Figure 4.3). XSL files are static files. The XML content is dynamically generated according to the data items necessary to produce a certain LoL@ screen.

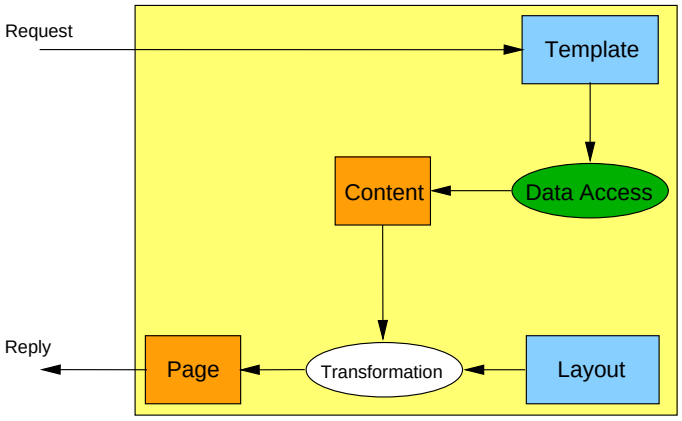

Figure 4.3: Templates and Data Access

The LoL@ application's content is fetched from various data sources (see Section 4.3) which are mostly proprietary (e.g., do not use well-known interfaces resp. conform to standards like JDBC). This is the main reason for the fact that an evaluation of state-of-the-art Web publishing tools (see Section 6.2) revealed that none of the existing tools which provide a separate entity for layout as outlined above would be easily adaptable to integrate the LoL@ data sources existing at that time. An initial estimation of the learning time necessary to know an existing tool's mode of operation from back to front to be able to *adapt* it for our needs in LoL@ showed that this would take the same amount of time as designing and implementing one from scratch. Thus it was decided not to use an existing tool, but to design and implement a tool tailored to LoL@'s needs.

It is not only necessary to connect to these data sources and retrieve data from them, but also to define which content should be included into which page. Templates are used for this definition. For each LoL@ screen, there is a definition of the content of the page that has to be produced to satisfy the user's request. Templates, which are static files, define how the process of dynamic creation must be executed. Figure 4.3 shows the templates and the data gathering step.

Figure 4.4 provides a schematic overview of the proposed system's overall architecture, including the components necessary for data gathering. The different entities in the system and their responsibilities are:

[1] Extensible Markup Language
[2] Extensible Stylesheet Language

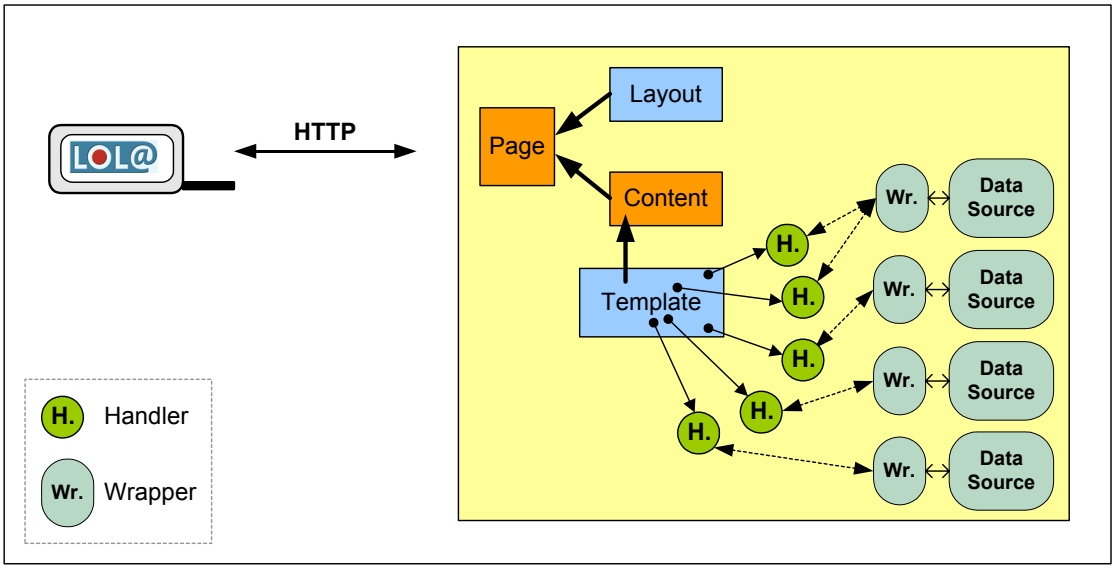

Figure 4.4: Schematic View of the LoL@ Server Core Application

Data sources. The data sources provide all data that is used in the application. The LoL@ data sources (content database, user database, mapping server, Session Manager, Location Manager, and the local filesystem) and their interfaces are described in Section 4.3.

Wrappers. Wrappers prepare this data to provide unified data format, data encoding, and access methods. Wrappers are described in Section 4.3. In Figure 4.4, the Wrappers are depicted as blue ovals with the abbreviation "Wr.".

Templates. Templates define the human-computer interface. Templates do not define what a page looks like, but what content it has and which basic layout definition (see Section 3.3.3) to use. An initial decomposition of the LoL@ screens and the LoL@ screen structure revealed that it is not necessary to define each and every LoL@ screen. Instead, it is possible to create templates that define a category of screens. I.e., the screen with the title "History" shown in Figure 3.7 is available for almost all PoIs, but presents different textual descriptions for each PoI. Nevertheless, there is just one template that defines the contents of all history screens. Templates are described in Section 4.2.1.

Handlers. Handlers act as adaptors between wrappers and templates: They react upon the template's structure and send appropriate calls to wrapper methods. Handlers are described in Section 4.4. In Figure 4.4, the Handlers are depicted as green circles with the abbreviation "H.".

Page content. The page content is dynamically created by the collaborative work of templates, handlers, wrappers, and data sources. The page content is included in a filled-

```
<template sid="301" resumeable="yes" cacheable="yes">
  <header>
    <SQL>
      <text>select title, icon from PoI where PoIID = </text>
      <para>pid</para>
    </SQL>
    <titlestring>: Information</titlestring>
  </header>
  <content>
    <text>
       <SQL>
         <text>select description from PoI where PoIID = </text>
         <para>pid</para>
       </SQL>
       <SQL>
         <text>select * from GeneralInformation where PoIID = </text>
         <para>pid</para>
       </SQL>
    </text>
  </content>
</template>
```

Figure 4.5: Sample Template

out template. How page content is produced by processing templates is described in Section 4.2.3.

Stylesheets. The stylesheets store all layout information. For the LoL@ demonstrator, the rules defined in the stylesheets instruct the XSLT processor to produce HTML code. Other output formats are possible. Stylesheets are described in Section 5.3.

Pages. Pages are generated by applying the layout rules defined in the stylesheets to the page content. Pages are the result files produced by the system. They are sent to the mobile device for presentation at the screen.

4.2.1 Templates

Templates express the human-computer interaction flow – defined in [PFL+01] and presented in Section 3.4 – in machine-readable form. The XML [BPSMM00] format is used for this purpose. Figure 4.5 shows an exemplary template. A template consists of its name, basic layout definitions, parameters, instructions, links, and data.

Name. For naming of the templates, the screen numbers defined in [PFL+01] were used. These are triple- resp. four-digit integer numbers that uniquely identify a screen. The

basic structure of a template looks like this:

```
<template sid="XXX" resumeable="yes|no" cacheable="yes|no">
    ...
</template>
```

As can be seen in this example, each template defines if its result page is cacheable (see Section 4.5) and/or resumeable (see Section 3.4.8).

Basic layout definitions. The template defines which one of the four basic screen layouts (see Figure 3.5) is used. The corresponding XML elements are: <`table /`>, <`list /`>, <`textual /`>, and <`generic /`>. Furthermore, the template reflects the screen structure (Figure 3.4): The <`header /`> element and its children define the header part of the screen, whereas the <`content /`> element and its children define the content part.

Parameters. The HTTP parameter name/value pairs of the request must be accessible inside the template because they are needed for instructions and links. These parameters are not known before runtime. Hence, placeholders in the template are needed.

Instructions. The instructions define which data items are included in a LoL@ screen, and how these data items can be fetched. From this follows that the data source the necessary data items are stored in has to be defined by the instruction. A unique XML tag was defined for each data source. Additionally, the instruction has to include the command that must be used to communicate with the data source. The command depends on the type of the data source. Some commands are quite easy to express (e.g., positioning on/off), others need a lot of parameters (e.g., an SQL query that depends on certain HTTP parameters). According to the complexity of the commands, they are realized either as XML element attributes or as child elements of the instruction element. Figure 4.5 shows an example of an instruction: the <`SQL /`> elements define that the data source to connect to is the content database. The SQL statement itself is composed of <`text /`> elements that store the static part of the SQL query, and of parameter placeholders.

Links. The LoL@ screens have links to each other. The template has to define which links a screen contains. A link is defined by its link text and all HTTP parameter name/value pairs needed to request the page. Some link parameters depend on the parameters of the request. The following example shows a link that is defined using two parameters and produces the "History"-link shown in picture (4) of Figure 3.7. The parameter `sid` (Screen ID) is a constant value, whereas the parameter `pid` (PoI ID) is included in the client's request and therefore has to be inserted at runtime.

```
<link check="yes">
  <linktext>History</linktext>
  <linkpara><name>sid</name><var>310</var></linkpara>
  <linkpara><name>pid</name><para>pid</para></linkpara>
</link>
```

Data. The screen layout definitions include some data which is not stored in any data source (e.g., screen titles or button text). This data will be stored directly in the template.

All templates used for LoL@ are combined in one static XML document. The XML Schema [Fal01, TBMM01, BM01] defining this XML document is given in Appendix A.

4.2.2 Page Generation

The Server Core application accepts HTTP requests from the terminals, processes them and sends a reply.

Figure 4.6 shows the building blocks of the Server Core application's business logic and how these blocks relate to each other. Figure 4.7 shows the interactions between the entities when producing a result page upon request from a terminal. To simplify the matter, the cache's detailed mode of operation is omitted.

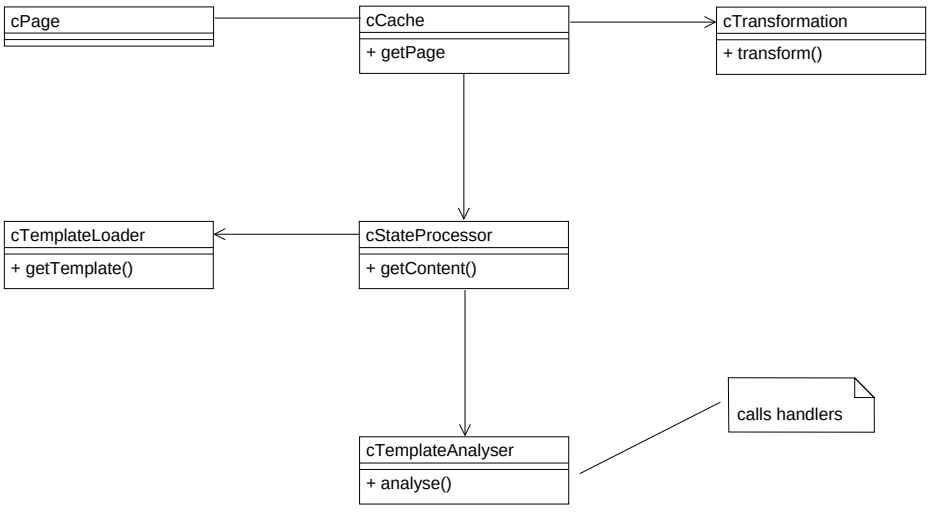

Figure 4.6: UML Class Diagram of Server Core

Users request *Page* objects. These objects are retrieved from the *Cache* which initiates the page creation step by invoking *StateProcessor* if a cache-miss occurs.

StateProcessor evaluates the input parameters (HTTP parameter name/value pairs) and controls *TemplateLoader* and *TemplateAnalyser*. First, the system has to decide based

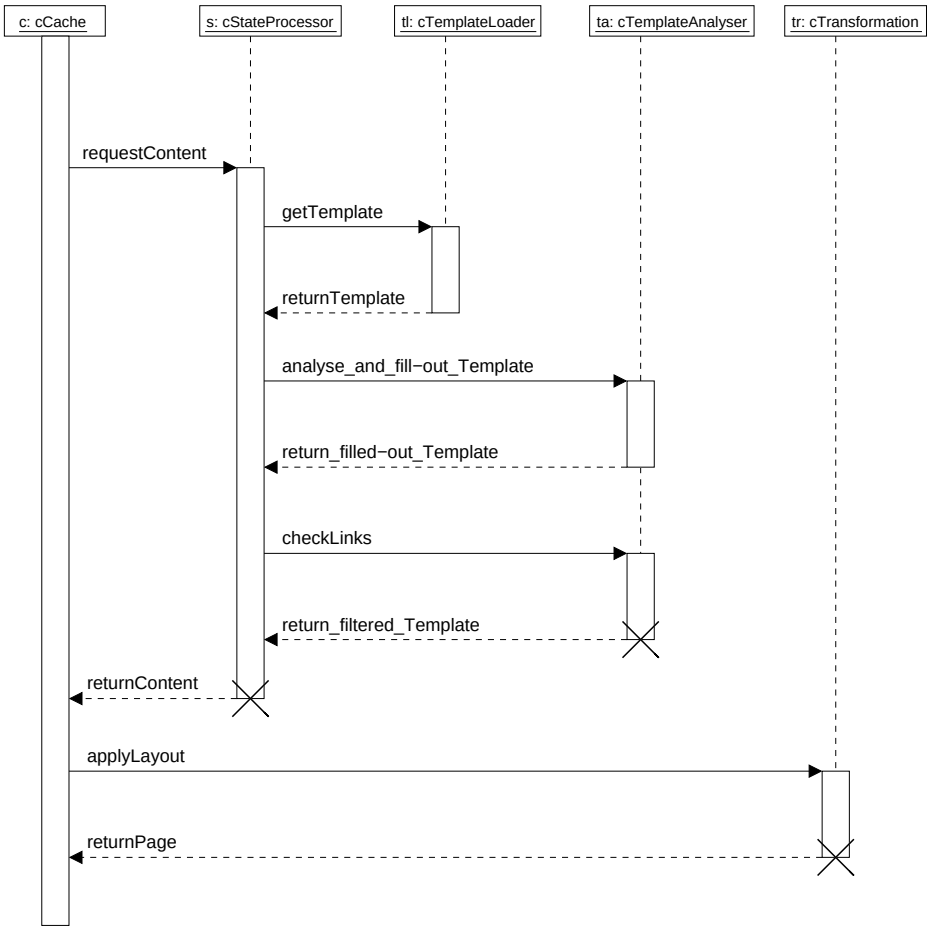

Figure 4.7: UML Sequence Diagram of Server Core

on the HTTP request which template to load. Hence, every HTTP request must contain the screen name as a parameter, because the screen name is used for this decision. *TemplateLoader* loads the appropriate template into memory. Now the template must be filled out.

TemplateAnalyser controls the necessary data gathering process. It analyses the templates. For each instruction it encounters, it calls the corresponding handler. The handler in turn executes the instruction and dynamically changes, removes, or extends the template's data as well as the template's data structure. The details of this step are described in Section 4.2.3. The result is a template document containing all data items required for the screen in marked-up format.

Subsequently, the filled-out XML template document acts as input source for the XSL transformation. The second input source for the XSL transformation is a stylesheet defining the rules for displaying the data items (e.g., the title is printed bold and with a font size of 10pt). With the help of an XSL(T) processor, *Transformation* applies these stylesheet rules

to the result of *TemplateAnalyser*'s work. For the LoL@ terminal, the output data defined in the stylesheets is HTML code suitable for the assumed display size (see Section 3.1). The result of the transformation step is an HTML page which is returned to the client as HTTP response.

4.2.3 Processing Templates

As outlined in the last section, processing templates means that the instructions included in the templates must be executed. This is done collaboratively by *TemplateAnalyser*, which controls the processing, and by *Handlers* which correspond to a certain instruction and know how to execute the latter. Figure 4.7 depicts that template processing consists of analysing and filling out the template, as well as of checking the links included in the template (see Section 4.2.1) for validity. This section explains these steps in detail. Figure 4.8 shows the entities involved. Figure 4.9 shows the interactions between *TemplateAnalyser* and the handlers.

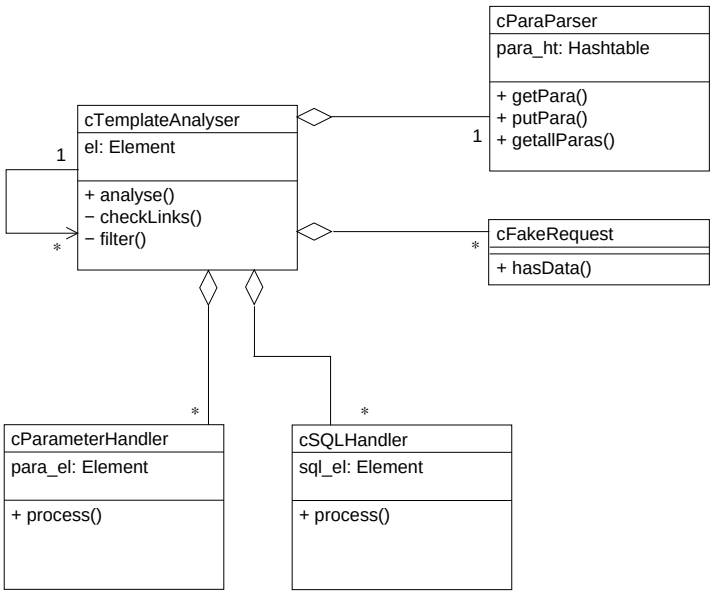

Figure 4.8: UML Class Diagram of Template Processing

Please note: To simplify the matter, Figure 4.8 and Figure 4.9 show merely the handlers that are necessary to process the sample template shown in Figure 4.5. However, this procedure is the same for other templates except for the fact that – depending on the included instructions – other handlers are called.

The processing of the template consists of 3 steps. In the first step, the parameter placeholders are replaced with their actual values. After that, the instructions are executed. Finally, the links are checked.

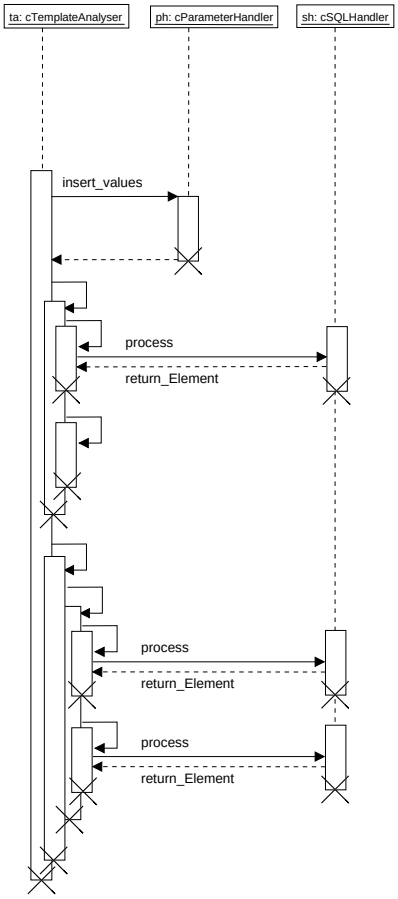

Figure 4.9: UML Sequence Diagram of Template Processing

Replace parameter placeholders. The parameter placeholders are replaced with the actual values of the parameters. This must be the first step of template processing, because other instructions depend on the actual values of parameters. *Parameter-Handler* is called before all other handlers and is responsible for replacing placeholders with actual values. If a parameter is defined in the template, but not supplied, an error is raised.

Execute instructions ... The instructions contained in the templates are parsed and executed. For each type of instruction, there is a corresponding *Handler* that knows how to deal with the processing of this instruction. *Handlers* are very small objects which are responsible for bridging the gap between an instruction and its execution. *Handlers* contact a data source (resp. its wrapper component, see Section 4.3) and call the correct methods to fetch the data. A listing of all handlers and their actions can be found in Section 4.4.

cTemplateAnalyser is responsible for invoking handlers. *cTemplateAnalyser* starts

with the root element of the template and calls itself recursively for every child element. If it encounters an element that is an instruction, it breaks the recursion and calls the handler for this instruction. The handler in turn executes its method. Figure 4.9 shows this procedure for the template depicted in Figure 4.5.

... and replace them with their result data. The XML elements that hold the instructions act as placeholders for the data inserted by the handlers. The data is provided with tags that mark it up as result data. *Handlers* do not return values. They work directly on the in-memory representation of the template. Figure 4.10 shows the sample template (Figure 4.5) with all result data filled in.

```
<header>
  <result>
    <result_tupel>
      <title>Palais Harrach</title>
      <icon>m_museum.gif</icon>
    </result_tupel>
  </result>
  <titlestring>: Information</titlestring>
</header>
<content>
  <textscreen>
    <result>
      <result_tupel>
        <description>Harrach, an Austro-Bohemian family of the higher
                     nobility ordered to build Harrach Palace.
        </description>
      </result_tupel>
    </result>
    <result />
  </textscreen>
</content>
```

Figure 4.10: Filled-out Template

Link checking. Finally, it must be checked whether pages the links defined in the template point to have data in order to decide if the link should be shown in the result page. Link checking must be done after the execution of all instructions, because link data could potentially be the result of a handler's work. If there is no data, the link should be shown as normal text. In this case, the XML tag marking up the link is removed, and the link is marked up as text.

FakeRequest performs these checks. A "faked" request is constructed by concatenating the HTTP parameter name/value pairs that define the link. *ParaParser* takes

this concatenated string and constructs an instance of itself that provides access to the "faked" HTTP parameters just as to other HTTP request's parameters. By calling *StateProcessor*'s method `getContent()` – exactly as it is done for HTTP requests from clients – and evaluating the result data, *FakeRequest* can determine whether the page has data or not.

The result data generated is additionally turned to account by the caching system used in LoL@. For more information, refer to Section 4.5.

Obviously, the procedure described above works recursively. The execution of `getContent()` will cause the page generation process to start (see Figure 4.7), which will lead to another link checking procedure. As there are no circular links in the LoL@ hypertext structure, this is no problem. Additionally, by defining the attribute `check="no"` for a certain element `<link />`, it is possible to omit checks for this link. This is used for links that do not need to be checked because it is clear before run-time that there will always be a result page for this link. Please note that this method is not generally applicable. It will cause performance problems when used for a Web site that has a lot of pages and many internal links. For the LoL@ site structure, which is structured in a tree-like way, using this method is possible. For other site structures the link checking mechanism must be limited regarding recursion depth.

4.3 Data Sources

This section describes the heterogeneous LoL@ data sources (Figure 4.11) that contain both static and real-time tourist-related information as well as data created by the mobile network's components. For each data source, the connectivity to it and the data format the result data is provided in is described.

If necessary, *wrappers* act as mediators between the characteristics of the data source and the needs of the LoL@ Server Core application. The wrapper components are responsible for accessing the data source and for preparing the result data. They must ensure:

- standardized data encoding. In LoL@, UTF8 [AAB+00] is used.
- standardized result data format. As LoL@ uses an XML-centric approach, DOM data structures [HHW+00] are most suitable for further processing. The Document Object Model (DOM) represents XML documents in memory in a tree-like data structure and provides a standardized way for manipulation of XML data with programming languages. Since an in-memory representation of the XML data in order to manipulate its data elements as well as its data structure was needed, DOM was preferred over SAX [SAX02].

Section 4.3.1 describes the functionality of and the interface to the mapping server. Session Manager and Location Manager are described in Section 4.3.2 resp. Section 4.3.3.

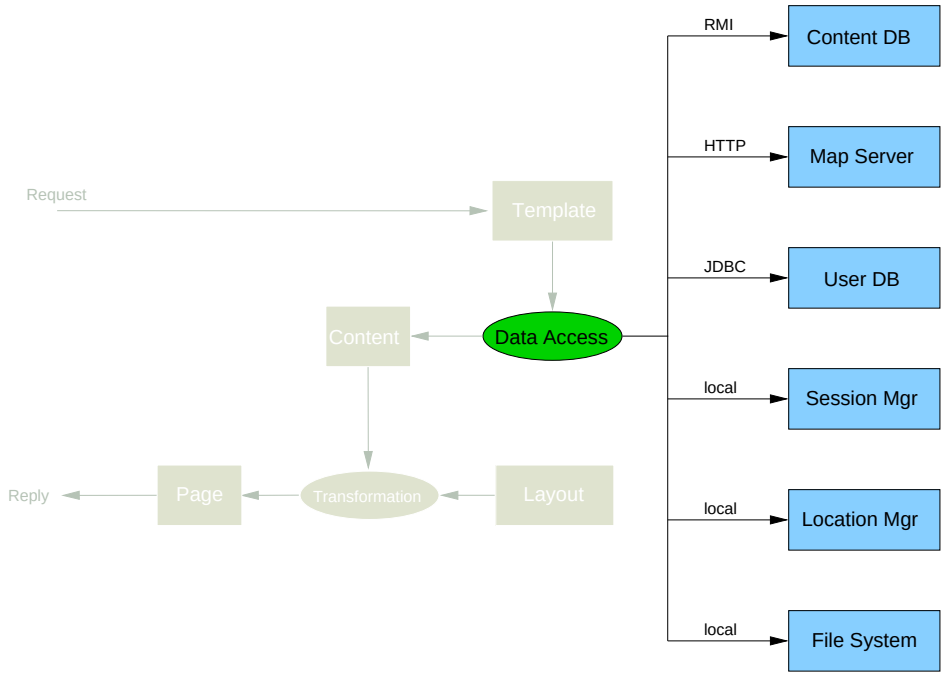

Figure 4.11: Data Components

The database connections used and the database layout are described in Section 4.3.4. Section 4.3.4.1 describes which kind of files are stored in the local filesystem and how they are organized.

4.3.1 Mapping Server

The mapping server provides textual routing information. Basically, the routing information is gathered by sending HTTP requests to the mapping server which will send HTTP replies containing routing information items formatted as simple HTML code in an agreed format. Detailed information about the routing concept and the mapping server as well as the specification of the input parameters of HTTP requests and the reply syntax of the HTTP replies can be found in [BFGPU01][3].

The mapping server accepts Gauss-Krüger coordinates[4] [HGM02] as input parameters and calculates the following kinds of routing information:

[3]The routing concept and the implementation of the mapping server are not within the scope of this thesis.

[4]To represent geodetic coordinates (latitude and longitude) in a map, they are mathematically projected onto a surface that can be layed flat. In case of Gauss-Krüger, the surface is a concentric cylinder which is tangent to the equator and makes contact along one meridian. For Austria, this is meridian 34. The Gauss-Krüger system is also called Transverse Mercator system.

Shortest route between two arbitrary points. If the user is away from the tour, he or she can request information about how to reach the tour. Routing to the tour selects the shortest way to the given destination.

Route between two PoIs. The main application of routing in LoL@ is to guide the user along a pre-defined tour from one PoI to the next PoI. When using routing along the tour, only street sections which are part of the tour are taken into account for the calculation of the route. This means that potentially not the shortest way is selected.

All street sections within a circle with a certain center point and radius. When users start routing the first time, their geographical position is determined by the location subsystem (see Section 2.2). When the conditions are bad (e.g., too few satellites available to determine the position with the GPS receiver), the location accuracy can be too low to be usable (i.e., it is not possible to uniquely determine the street the user is located at but rather a tupel of streets the user could be located at). To overcome this problem, an additional method to determine the user's position is used unter this conditions.

Figure 4.12: Initial Positioning

The (not very accurate) estimation of the user's location is taken as an input parameter and all streetnames which are within a circle with a defined radius (see Figure 4.12) are presented to the user. After choosing the street the user is currently located at, the location of the user is known. The user interface of the initial positioning process can be seen in picture (5) of Figure 3.10.

Midpoint of a street section: After the user has selected a position in the interactive initial positioning process, the only identifier known about the street section chosen is an (internal) ID of it[5]. As street sections are comprised of a set of coordinates but – for further calculations – a single coordinate is necessary, the midpoint of the street section is used. The mapping server provides a script that accepts street section IDs as input parameters, calculates the Gauss-Krüger coordinates of the midpoint of this street section, and returns them. These coordinates are usable input parameters for other mapping server scripts.

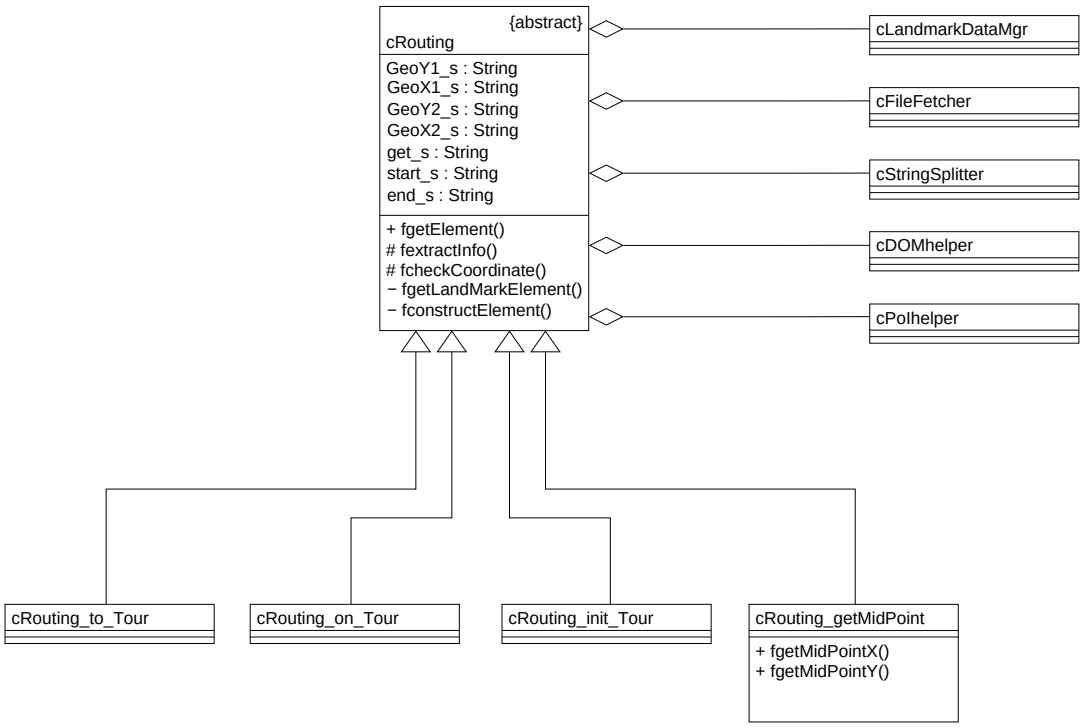

Figure 4.13: UML Class Diagram of Mapping Wrapper

The Mapping Wrapper shown in Figure 4.13 provides access classes for all the functions of the mapping server described above. *Routing* is an abstract class that provides all necessary functions to communicate via HTTP with the mapping server. After retrieving the data from the mapping server, the Mapping Wrapper extracts the relevant parts of the acquired data and prepares them for later use. *FileFetcher* communicates with the server. *StringSplitter* parses the result data and creates a DOM data structure that contains the routing information in marked-up format that can be used for further processing. *PoIhelper* is a helper class that looks up Gauss-Krüger coordinates of PoIs in the content database.

Figure 4.13 furthermore shows *Routing_to_tour*, *Routing_on_tour*, *Routing_init_tour*, and *Routing_getMidPoint*, which provide access to the functions described above. They set the class variables of *Routing* according to their task and use the *Routing* methods to get the result DOM element. Whenever possible, PoI IDs rather than Gauss-Krüger coordinates can be used as input parameters.

If there are wrong or missing input parameters, or any problems in data accessing, a *RoutingException* is thrown. The following error conditions are possible:

- PoI ID does not exist.
- No difference between source and destination coordinates.

[5]These IDs are used by the map viewer (Section 3.3.2).

- Mapping server can not be reached over the network.
- Mapping server is up and running, but sends no files.
- Parsing the file is not possible due to wrong syntax.

4.3.1.1 Landmarks

There is additional information that is not fetched from the mapping server, but influences the routing mechanism results. User orientation and guiding is improved by the inclusion of so-called "landmarks" [BFGPU01]. These are significant buildings visible from the user's current position. Landmarks act as an orientation support for the user on his/her way to the next PoI.

Figure 4.14 shows the access component for landmark data. Landmark data is static data: textual information and images. It is stored in the content database and in the local filesystem. For every routing information item, *cRouting* (see Figure 4.13) has to check whether a landmark exists. If so, its textual information and the URI of the image is appended to the routing information item's DOM data structure. To prevent lots of database queries which are not performant (see Section 4.3.4), *LandmarkDataMgr* loads all data into memory at startup. To ensure that this is done only one time, *LandmarkDataMgr* is designed as a singleton [GHJV95]. As landmark data does not change, consistency between the in-memory data and the data stored in the database is always ensured. *LandmarkDataMgr* is not a wrapper class. It is used by the mapping wrapper, but not by handlers.

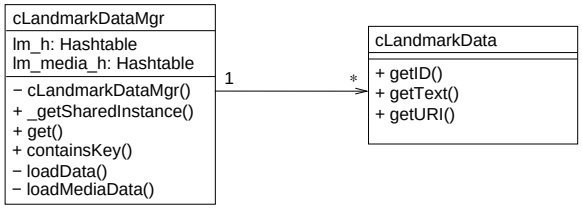

Figure 4.14: UML Class Diagram of Access to Landmark Data

4.3.2 Session Manager

As described in Section 2.2, the Service Platform accesses the mobile network domain using the Parlay interfaces and provides the mobile network's functions to the LoL@ Server Core application. The Session Manager[6] is a part of the Service Platform. It provides access to a user's mobile network subscription related information by querying the *Home Subscriber*

[6]The Session Manager was done in cooperation with Peter Wenzl, Ericsson Austria.

Service (HSS)[7]. It is necessary to build an infrastructure for user data administration to perform this task.

This infrastructure is reapplied for a task that has similar requirements: For the resume function (see Section 3.4.8) and for management of users' preferences, it is necessary to manage user-specific data related to the user's current LoL@ session (i.e., the user's application state), and to store this data in a persistent way.

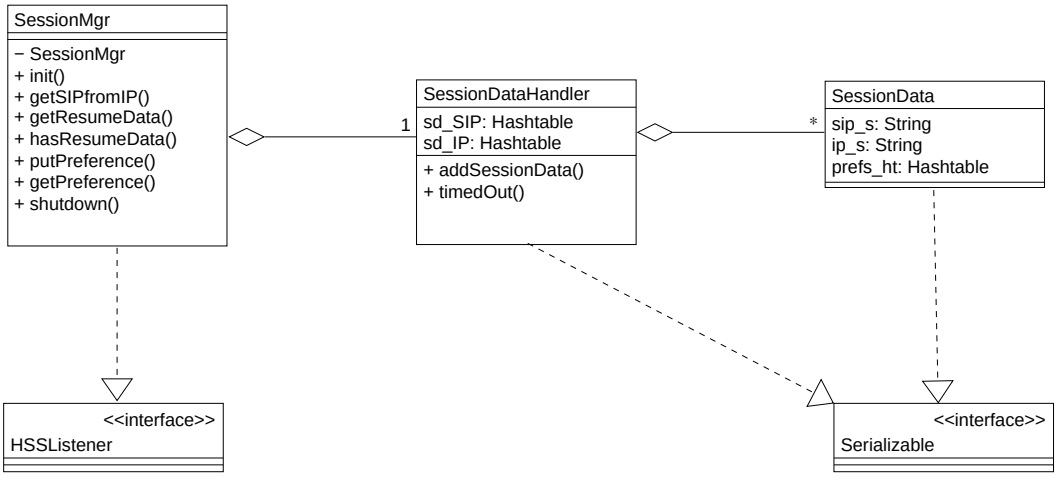

Figure 4.15: UML Class Diagram of LoL@ Session Manager

In detail, the Session Manager is responsible for:

Mapping from SIP address to IP address and vice versa. The mobile network provides a unique ID of every terminal: Every end-user has a unique SIP address[8] [HSSR99]. As HTTP is a stateless protocol, session tracking is necessary to maintain a relationship between two successive HTTP requests. LoL@ uses SIP addresses for session tracking. For every occuring HTTP request, the SIP address corresponding to the client's IP address is looked up and the request can thus be matched to a certain user. In comparison to traditional Web applications, where either hidden HTML form fields [FIE99], URL rewriting [Hal00], or cookies [COO99] are used for session tracking, this is a superior way. Each of the three methods mentioned above uses a certain method to transfer data about the client's state between client and server that can either be turned off or be manipulated by the client. Since SIP addresses are provided by the mobile network, manipulation by the client is not possible.

Persistent storage of the users' status. Some information about the client's state has to be stored in a persistent way. *SessionData* models the data to be stored. Persistent

[7]The HSS is realized as an LDAP server.

[8]SIP (Session Initiation Protocol) addresses have a syntax similar to email addresses: `sip:user@host`.

storage is achieved by serializing the data-containing parts of the *SessionMgr* components and a subsequent write to a file. During initialization the Session Manager reconstructs its previous state if such a file exists. This mechanism is required for reconstruction of the session information for logged-in users in case of an unintended shutdown of the LoL@ Core. The information stored about a user expires if he or she does not use LoL@ for a defined time period.

Figure 4.15 shows the Session Manager's main components. *SessionMgr* is implemented as a singleton [GHJV95]. This allows easy referencing by other components of the LoL@ Core architecture and ensures consistency of user data. The Server Core is responsible for initializing resp. shutting down the only existing instance. This is done in the main servlet's `init()` and `destroy()` methods.

4.3.3 Location Manager

The Location Manager[9] [AK01] is part of the Service Platform and provides an abstraction of and an interface to the mobile network's positioning functionality towards the Server Core application. *LocationMgr* (Figure 4.16) is implemented as a singleton and responsible for:

Sending position information to the terminals. Upon request of the Server Core, the *LocationMgr* starts resp. stops positioning for a certain user in the operator's positioning system. *AppPeriodicLocation* receives all positioning information and forwards it via the *LocationListener* interface to the *LocationMgr*. The *LocationMgr* then forwards position information and error messages to the *UserConnection* corresponding to the respective LoL@ client.

Coordinate checking The method `PosNear()` is used to check if user input on his/her current position complies with the location estimate determined by the LCS service (see Section 2.2.)

Providing an asynchronous communication channel between server and terminal. Each *UserConnection* is associated with a specific user terminal (see Figure 4.16.) *UserConnection* handles location information and errors on a per-user basis. It also handles the socket connection to the terminal and communicates with *at.ftw.C1.terminal.CoreMsg*. *LocationMgr* provides a public method `getUserConnection()` to allow other components to use its asynchronous communication channel between server and terminal.

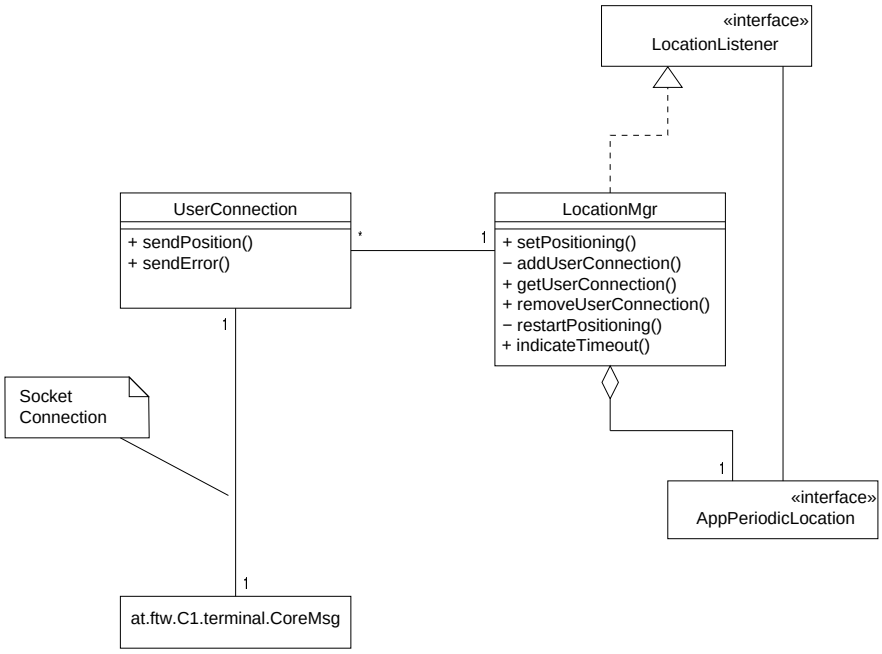

Figure 4.16: UML Class Diagram of LoL@ Location Manager

4.3.4 Database Connectivity Components

The applications' content data resides in a database on a remote server. The main function of the content database is the storage of numerical and textual values.

[KPP+01] specifies the database connection: SQL queries are sent to the remote server using the RMI[10] protocol [RMI02]. The remote server executes the query using a JDBC driver. Each result tupel of the query is converted into a `java.util.Vector` and these are collected in another `java.util.Vector`. Finally this Vector array is returned, again using RMI calls.

This kind of database connection[11] turned out to be inefficient and was the main reason for integrating a caching system in the LoL@ design. Caching reduces the performance penalty created by the RMI connection for read-only data. Since the major part of the content database's tables hold static data that is never changed by the application, a simple design of the caching system – without using a replacement strategy – is possible.

On the other hand, there exist tables which are updated frequently: The tour diary notes (see Section 3.4.5) are also stored in the database. Since designing a caching system that can handle read-write transactions is a non-trivial task, migrating the read-write part of the database – the user database – to another database management system that can

[9]The Location Manager is not within the scope of this thesis.
[10]Remote Method Invocation
[11]This design decision was done outside the scope of this thesis.

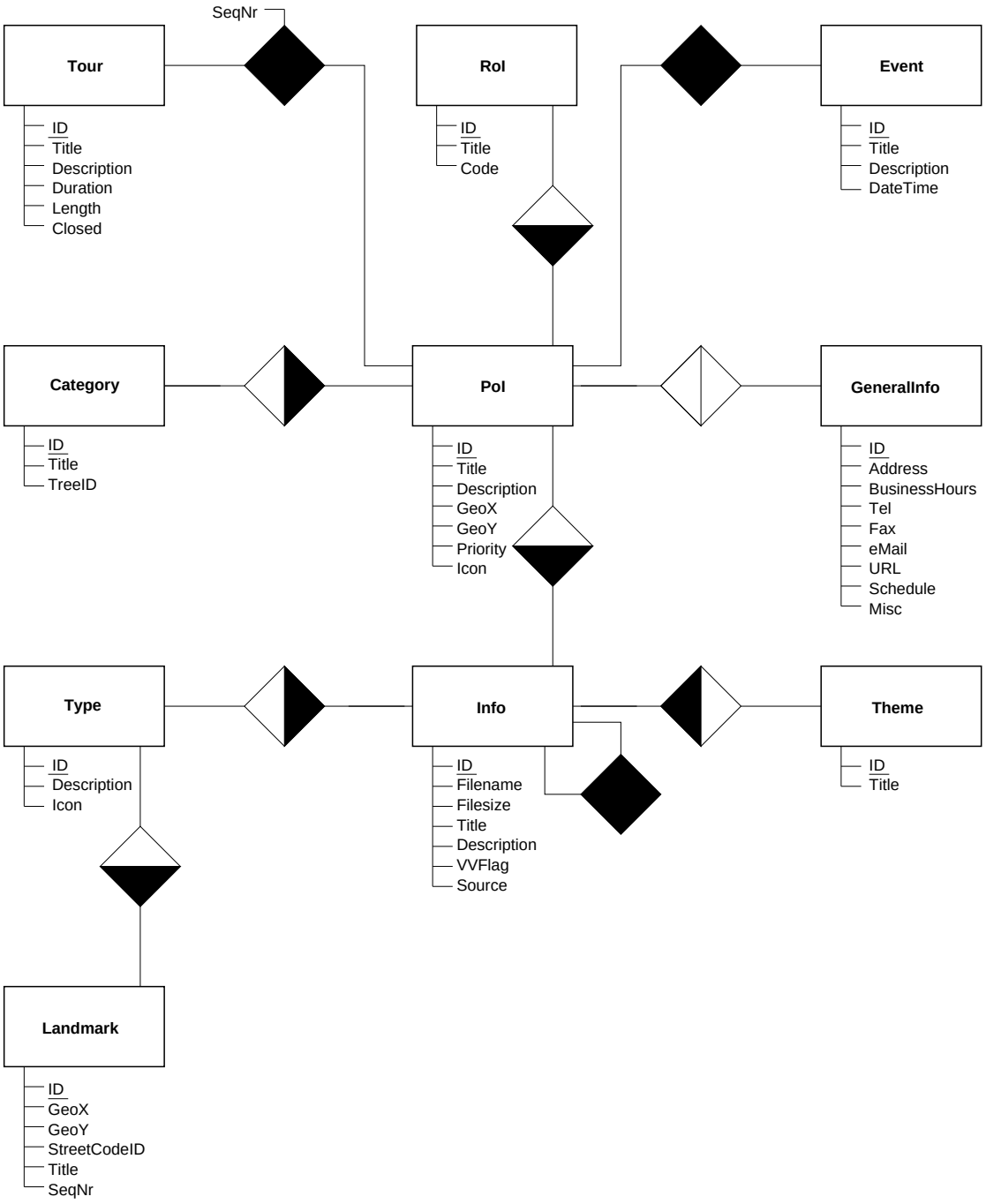

Figure 4.17: EER of Content Database

be accessed using a standard JDBC connection [JDB02] was an equitable design decision and solved the problem of poor read-write operations' performance. The decision was also influenced by the fact that tour diary data does not yet exist but will be created by users, simply because migrating an empty database is a straightforward task.

The design of the database(s) access component is shown in Figure 4.18. *RMIDBConnection* is used for accessing the content database. *MySQLDBConnection* is used for accessing the user database.

The Extended Entity Relationship diagram of the content database is shown in Figure 4.17. Most of the queries which fetch data from the content database are defined by instructions in the templates (see Section 4.2.3). For information stored in the content database that is required for LoL@'s business logic, access classes were created: *cPoIHelper* provides methods to lookup information about PoIs, like their name and their Gauss-Krüger coordinates. *cInformationHelper* is described in Section 4.3.4.1.

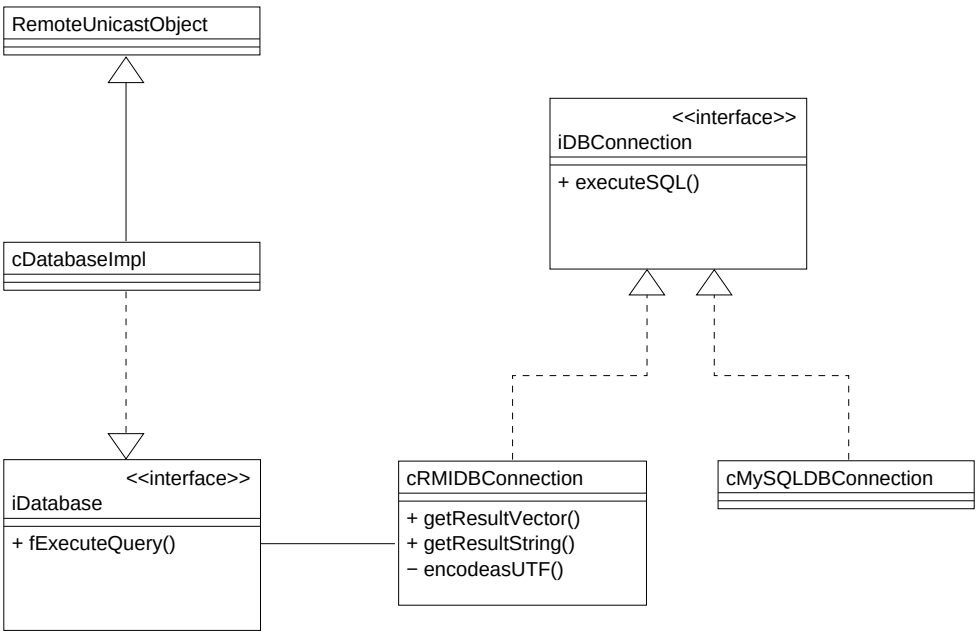

Figure 4.18: UML Class Diagramm of Database Connection

4.3.4.1 Local Filesystem

For performance reasons, the multimedia content (text, audio, video, images) is not stored directly in the database. Multimedia files are stored in the local filesystem and only their path information (local URI) is stored in the content database.

Delivery of static multimedia files to clients is done by the Web server. To make the files accessible, they have to be stored under the document root of the Web server. Video

files are delivered by a streaming media server. As video embedding is done using the HTML <embed> tag, no integration of the streaming media server within the Server Core application is necessary.

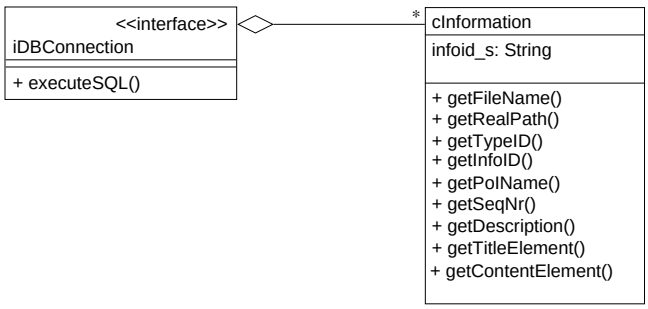

Figure 4.19: UML Class Diagramm of *Information* Access

Path information about files is stored in the content database's table `Info`. The *Information* access class (Figure 4.19) is used for convenient access. This class provides translation between the URIs of information tupels stored in the database and their real location which do not always match.

4.3.4.2 Diary Notes Access Wrapper

The different kinds of diary notes were already presented in Section 3.4.5. A wrapper class[12] abstracting the database queries necessary to store and retrieve the various kinds of diary notes was created [HMP+01]. Figure 4.20 shows the Diary Wrapper's design.

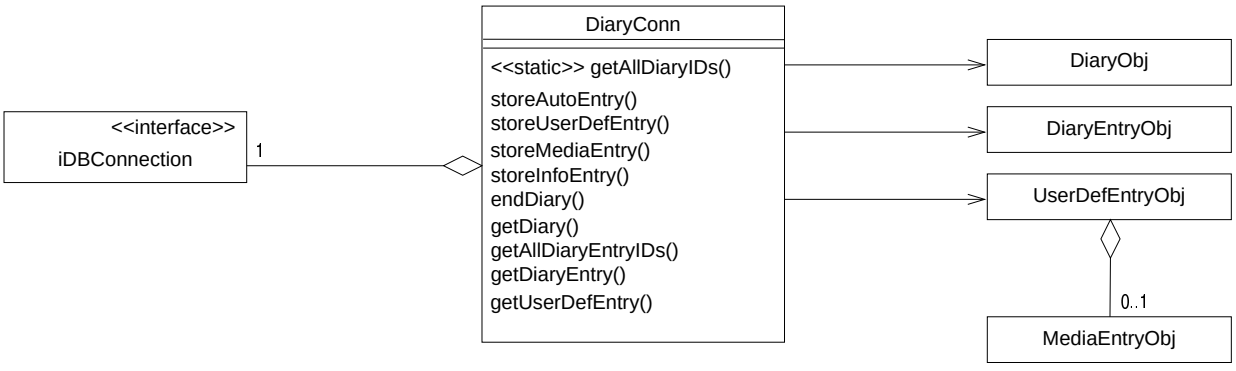

Figure 4.20: UML Class Diagram of Diary Database Access

[12]These classes are not within the scope of this thesis.

4.4 Handlers

As already mentioned in Section 4.2.3, handlers implement the business logic. Each handler corresponds to a certain instruction. All handlers implement the *iHandler* interface which stipulates that they must provide a method `process()`. This method invokes the actions necessary to execute their instruction. *cHandler* is an abstract class which is extended by *cSessionHandler* and *cLocationHandler*. These provide access to an instance of *SessionMgr* resp. *LocationMgr*. Handlers which are in need of one of these further extend these classes. Handler which do not need that simply extend *cHandler*.

Instructions are defined using XML elements. These elements can have attributes and child elements. The XML Schema that defines the attributes and child elements can be found in Appendix A. All handlers get an XML data structure as input and change, extend, and/or remove certain parts of the data structure. For accessing and manipulating XML data, JDOM [JDO02] is used.

Figure 4.21 shows all handlers. The remainder of this section explains each handler in detail.

4.4.1 ParameterHandler

This handler is responsible for looking up the values of HTTP parameters by name. As shown in Figure 4.21, access to HTTP parameters of requests is provided by the *ParaParser* object. Instances of this object have a one-to-one relationship to an HTTP request. *ParameterHandler* gets the name of the HTTP parameter as input. It uses *ParaParser*'s method `getPara()` to lookup the corresponding value. Finally, the name of the parameter is replaced with its actual value.

4.4.2 SQLHandler

SQLHandler is responsible for the execution of SQL commands. Using *cRMIDBConnection*, it sends the input SQL query included in the DOM data structure to the database management system for execution. The result is prepared as follows: Each result tupel is represented by an XML element `result_tupel` and its child elements. The tables' column names are used as the element names of the single elements of the tupel. Figure 4.10 shows what the exemplary template shown in Figure 4.5 looks like after *SQLHandler* did its work.

4.4.3 TimeHandler

This handler is responsible for inserting the current date and the current time in the specified format. *TimeHandler* simply relies on `java.util.Date`.

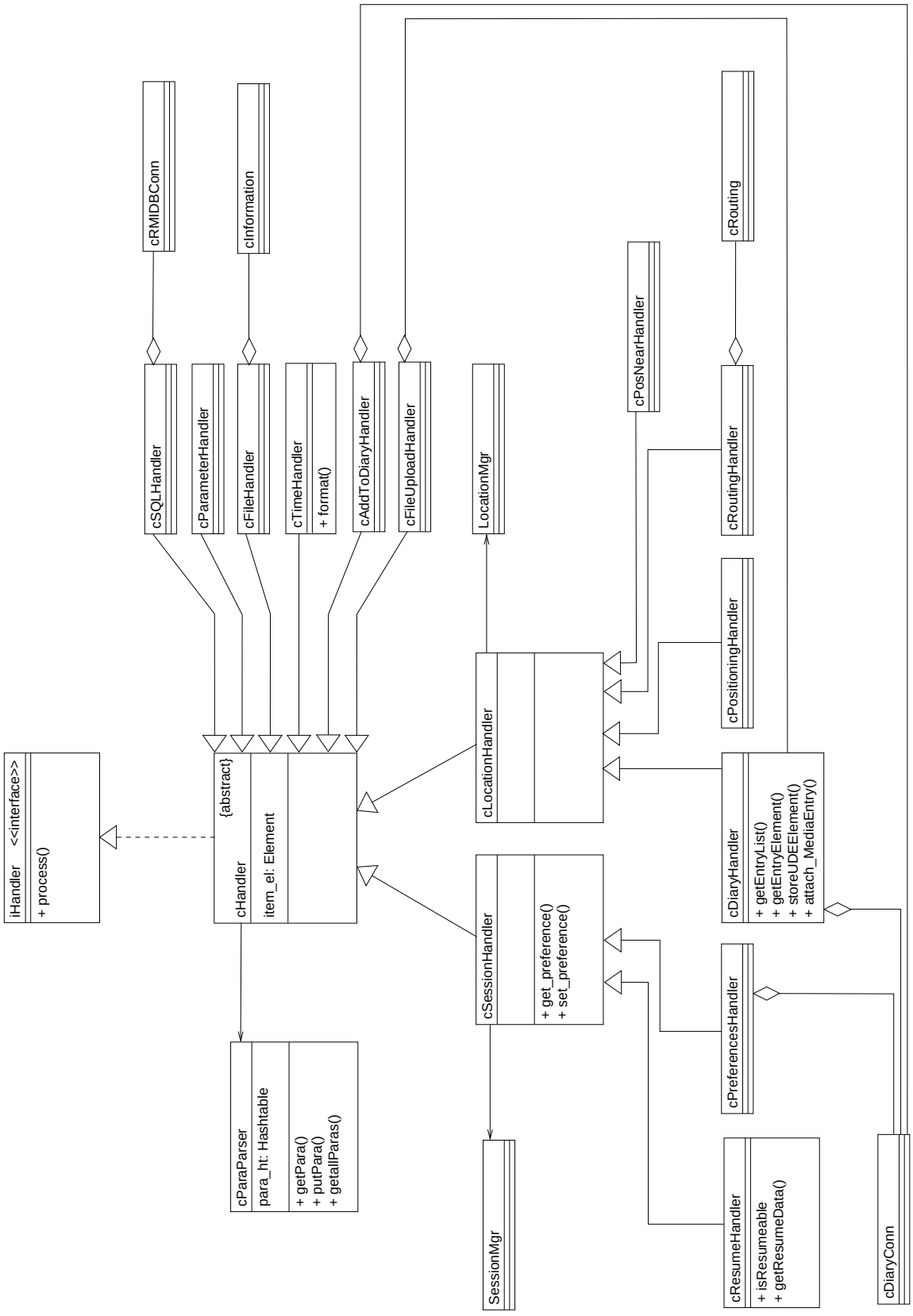

Figure 4.21: UML Class Diagram of Handlers

4.4.4 FileHandler

FileHandler relies on the *Information* wrapper class and is responsible for retrieving files resp. information about files from the local filesystem. These files have different files types, but – pertaining to the actions necessary to prepare their retrieval – they can be classified into two categories.

Text files. Some textual information about the PoIs is stored in text files. The contents of these text files are loaded into memory, and a DOM data structure is created out of them.

Images, audio and video files. These files are integrated by including HTML `<embed />` tags resp. the HTML `<img />` tag in the result page. *FileHandler* has to insert the file's URL (and optionally the file's properties i.e., height and width) into the data structure. The client's browser then makes subsequent HTTP requests to fetch these files.

4.4.5 PreferencesHandler

PreferencesHandler has to store and retrieve the user's preferences and the Terminal Core's status variables. Both of these are boolean resp. integer values. They are sent as HTTP parameter name/value pairs. *PreferencesHandler* forwards these name/value pairs to and retrieves values by name from the Session Manager.

Name	Value	Description
pos	boolean	Sent when the user clicks on the *Positioning* button. See Figure 3.10.
voice	boolean	Sent when the user changes the value. See Figure 3.6.
mapdur	integer	Sent when the user changes the value. See Figure 3.6.
map	integer (0/1/2)	Sent whenever the user changes views (0 = textual screen, 1 = detail map, 2 = overview map). See Section 3.4.3.
foundit	integer	Sent when the user clicks the *Found It!* button. See Figure 3.10.
back	1	Sent when the user clicks the *Back* button. See Table 3.1.

Table 4.1: User's Preferences and State Variables

On every invocation of *PreferencesHandler*, a DOM data structure containing all variables and their actual values is returned, including the newly set values of the variables. If a value is not yet set, a default value is returned.

4.4.6 ResumeHandler

This handler is responsible for resuming the tour in case of unintended shutdown/interruption of the user's LoL@ session (see Section 3.4.8). There are two different kinds of data relevant for determining if and what to resume:

- The user's preferences described in Section 4.4.5.
- As described in Section 2.3, each HTTP request (including its parameters) from the client is forwarded to the Session Manager, if the corresponding template defines it as resumeable (see Section 4.2.1).

Upon invocation, *ResumeHandler* retrieves all data available for a certain user from *SessionMgr*. If there is no data available for this user because he or she has not used LoL@ before resp. too long ago, *SessionMgr* will not return values. From this follows that the previous tour can not be resumed.

If there is data available, *ResumeHandler* has to evaluate and prepare the data. The state variable `map` described in Section 4.4.5 determines whether a textual or a map screen has to be resumed. Resuming a textual screen is a straightforward task: The stored HTTP request including its parameters provides this data. Resuming a map screen is done by the Terminal Core. *ResumeHandler* has to add all name/value pairs described in Section 4.4.5 to the DOM data structure. Section 5.3 describes how these values are supplied to the Terminal Core which is responsible for resuming map screens.

4.4.7 DiaryHandler

This handler manages the retrieval of diary notes and the storage of user defined notes. To achieve this, *DiaryHandler* relies on the Diary Wrapper classes. The Diary Wrapper provides functionality to store and retrieve diary notes from the user database.

If positioning is turned on in the user's preferences, the current position of the user must be saved as part of the diary note. In this case, *DiaryHandler* needs an instance of *LocationMgr* to get access to the current position of the user.

4.4.8 FileUploadHandler

This handler has to evaluate whether saving the file was successful. File storage is done with the help of `com.oreilly.servlet.MultiPartRequest` [COS01]. If saving the file fails, *FileUploadHandler* generates an error message.

If saving was successful, it is neccessary to add a reference to the URI of the file stored at the server to the user database to enable later downloading of the file. To do so, *FileUploadHandler* relies on *DiaryHandler*.

4.4.9 AddToDiaryHandler

AddToDiaryHandler relies on the Diary Wrapper classes and is responsible for adding Information notes to the tour diary. This is a straightforward task.

4.4.10 PositioningHandler

This handler is responsible for stopping and starting positioning for a certain user. *PositioningHandler* instructs the Location Manager to act accordingly.

4.4.11 RoutingHandler

This handler relies on the Location Manager and the Routing wrapper classes. According to its input parameters, *RoutingHandler* has to decide which routing methods to use. Table 4.2 shows how this decision is done.

	Source PoI available	**Source PoI NOT available**
Destination PoI available	Use *Routing between two PoIs*.	Current user position is determined using *Initial Positioning*. Use *Routing between two arbitrary points*. Call *PosNearHandler* to check if user input on his/her current position and estimated location of user (from LCS) correspond.
Destination PoI NOT available	Destination PoI is next PoI on tour according to the given Source PoI. Use *Routing between two PoIs*. Call *PosNearHandler*.	Source PoI is determined by using *Initial Positioning*. Destination PoI is the first PoI of the tour. Use *Routing between two arbitrary points*.

Table 4.2: Routing

The Routing wrapper classes need source and destination coordinates to calculate routes. If these input parameters are not supplied by the terminal's request, they have to be determined. The appropriate method is called and the result data is inserted into the DOM data structure.

4.4.12 PosNearHandler

PosNearHandler's input parameters are the SIP address of a certain user and user input on his/her current position. *PosNearHandler* relies on the Location Manager's method `PosNear()` to compare the user input on his/her current position encoded in the terminal's HTTP requests and the location estimate determined by the LCS service. *PosNearHandler* returns a boolean value.

At the moment, this handler is used only in conjunction with *RoutingHandler*. As it is already foreseen that user position checks will be done independently from the routing mechanism, *PosNearHandler* is designed as self-contained handler rather than as part of *RoutingHandler*.

4.5 Cache

For performance reasons, a caching system is used. The cache is designed in a transparent way which means that a client of the cache will not gain insight whether a request is fulfilled with cached or generated data[13]. Hence, the cache controls the page generation mechanism.

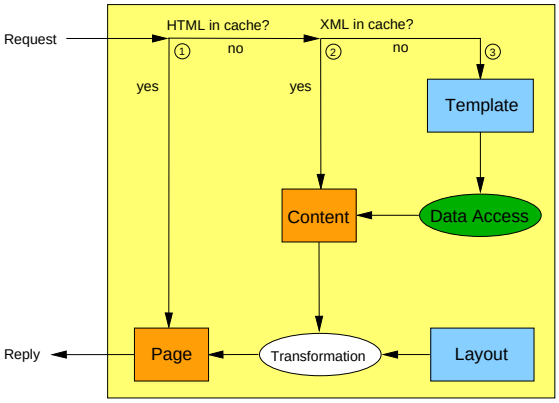

Figure 4.22: Two-tiered Cache

[13] A smart client that measures the time it takes until the request is finished will know whether the page was generated or cached, because requests fulfilled with cached data are processed substantially faster.

In the beginning of this chapter, Figure 4.3 showed an overview of the different entities the LoL@ Server Core application is comprised of, as well as their interactions. In Figure 4.22, the caching system's mode of operation is added and it is shown upon which principles the cache chooses which cached items can be used, if cached items can be used at all.

The caching system manages two data repositories: one that contains HTML data, and another one containing XML data. Figure 4.22 shows that for each occuring request a cache lookup for HTML code appropriate to satisfy the request is done. If HTML data is available (1), the cached HTML page is sent back to the terminal.

If this is not the case, the second-stage cache comes into play. As already explained in Section 4.2.3, links in templates are checked by fake requests. This produces XML data. This data is not thrown away, but cached for later use in order to improve performance. These cached data items will very likely be used shortly afterwards because the user is likely to click on the links presented to him/her. In (2), a lookup for XML data is done.

If there is neither HTML nor XML data available (3), the complete data generation process has to be executed to produce the requested HTML page. After sending the HTML result back to the terminal, it is also written to the cache. The intermediate XML data that is produced by the page generation process is not written to the cache, because it is – obviously – never used. If there are further requests for the same LoL@ screen, the HTML data stored in the cache will be used to fulfill it.

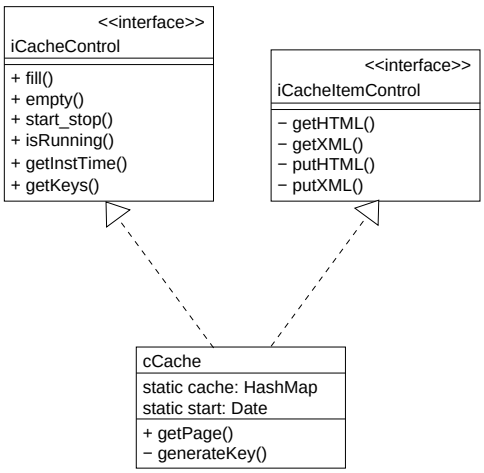

Figure 4.23: UML Class Diagram of Cache

The design of the cache is simple: No replacement strategy is used. If one or more of the cached pages are outdated, it is necessary to empty the contents of the cache. Only pages that are comprised of data that is not changed in the data source can be cached. Each template defines if it is cacheable (see Figure 4.5). As already mentioned in Section 4.3.4, the caching system's main function is to remove the shortcomings of the connection to the content database which stores data that is never changed by the application.

A simple Web-based interface – the Cache Manager[14] – is available for manipulating the contents of the cache. An exemplary screenshot of the Cache Manager is shown in Figure 4.24. *iCacheControl* (see Figure 4.23) defines the actions that are possible.

Starting and stopping the caching system. For testing purposes, it is possible to stop and start the cache. If the cache is stopped, each page will be generated from scratch. Additionally, controlling whether the cache should be used can be done at a finer grain: Appending the name/value pair `cache=0` to the parameters of a certain HTTP request will instruct the LoL@ Server Core application not to use cached items for this request, but to execute the page generation process. This feature was very helpful during the implementation phase.

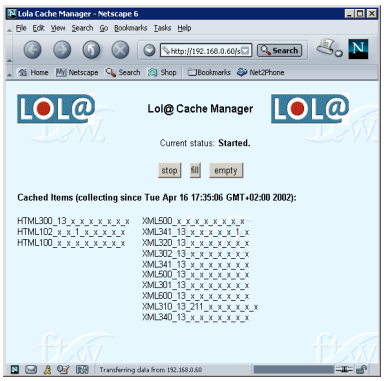

Figure 4.24: Cache Manager

Filling the cache. Normally, the cache would fill itself with each request done, but LoL@ is a demonstrator, so in addition the possibility to fill the cache at startup (or at an arbitrary time) is provided. This is done by executing `wget` [GNU02] – a command-line tool used for retrieving files via the HTTP protocol – with appropriate parameters.

Emptying the cache. Removing all cached items from the cache is a straightforward task: Calling its method `clear()` removes all entries from the `java.util.HashMap` that stores the cached items.

[14] http://lola.ftw.at/servlet/CacheMgr (This URL is secured with a password.)

5 Implementation

This chapter describes the Java-based implementation of the design that was described in the previous chapter. Section 5.1 introduces the tools used for implementation. Section 5.2 presents the Java packages. Section 5.3 discusses special problems of producing HTML code for small screens with XSL transformations. Finally, Section 5.4 explains how to extend the LoL@ Server Core application for the integration of new data sources.

5.1 Implementation Tools

This section introduces Java servlets and relates the features of the latter to the features of other technologies available for Web application development. After that, tools used for XML manipulation with the Java programming language are presented.

5.1.1 Java Servlet Technology

The Java servlet technology [SER00] provides the possibility to develop Web applications in Java. A servlet is a generic server extension – a Java class that can be loaded dynamically to expand the functionality of a Web server. The servlet API gives access to HTTP request parameters (`javax.servlet.http.HttpServletRequest`) and specifies how to send the HTTP reply (`javax.servlet.http.HttpServletResponse`). Java servlets execute in a Java Virtual Machine (JVM) loaded by a servlet container. A servlet container can be compared to a Web server: Instead of serving static files, it executes the requested Java class files and returns the results (usually HTML code) produced by them. Widely used servlet containers are Apache Tomcat [TOM02] and Apache JServ [JSE02]. For LoL@, Apache Tomcat 3.2 is used.

In contrast to the CGI model described in Section 4.1, which spawns a new system process for every request, servlets are loaded into memory only once and run from memory thereafter until they are explicitly unloaded. Instead of spawning a new process, each servlet call spawns a new thread within the Web server process. This gives servlets performance benefits over CGI programs. In this respect, Java servlet technology is comparable to Apache mod_perl [ASF02]. mod_perl is a server-side scripting module for the Apache Web server. Just like servlet technology, mod_perl provides code caching. Modules and scripts

are loaded and compiled only once, and then served from the in-memory cache until they are explicitly unloaded or the Web server is shut down.

When code is cached and does not exit, it will not clean up memory as it would when using CGI. This can have unexpected effects (e.g., memory leaks potentially caused by the script resp. servlet are not cleaned up). In addition, both Perl with mod_perl and servlets do not initialize global resp. instance variables for each request. Each of the (potentially concurrently occuring) client threads can manipulate the values of these variables. Application developers have to implement their code in a thread-safe way to avoid inconsistencies of shared data caused by concurrency.

It is important to state that whatever Web application development tools and technologies used, there is a limitation that can not be removed, because it is completely independent from the server-side tools and technologies: HTTP is a stateless protocol. It provides no built-in way for a server to recognize that a sequence of requests all originated from the same user. To achieve the latter nevertheless, a unique ID must be allocated to every client in order to track his or her session. As already outlined in Section 4.3.2, the available methods for session tracking (hidden HTML form fields [FIE99], URL rewriting [Hal00], and cookies [COO99]) all transfer additional data about the client's state between client and server as a part of the HTTP requests. Web application development tools can only provide a convenient way to handle these rather limited methods. Just like there is `CGI.pm` for Perl and built-in support via the `session_*` directives for PHP, Java servlets hide details of session management resp. cookie handling by providing the Session Tracking API (`javax.servlet.http.HttpSession` and `javax.servlet.http.Cookie`).

In conclusion, Java servlet technology and Apache mod_perl provide equal functionality and features and face quite the same problems that stem from the shortcomings of using HTTP for stateful communication and the problems of cached code. It is best to choose the technology that fits best in the Web application's environment. For this reason, the LoL@ Server Core application is implemented using Java servlet technology.

5.1.2 Java and XML

There are many tools to access and manipulate XML documents within the Java programming language. The *Java API for XML Processing (JAXP)* [JAX02] enables applications to parse and transform XML documents using an API that is independent of a particular XML tool's implementation.

As shown in Figure 5.1, JAXP adds an additional layer between the calls for and the methods of particular XML tools. The benefits of this layer are that it provides a standardized API. Hence, JAXP makes it possible to switch between particular XML processor implementations without making application code changes (e.g., switch between Apache Xerces and Apache Crimson as XML parser, as depicted in Figure 5.1). JAXP supports processing of XML documents using DOM [HHW+00], SAX [SAX02], and XSLT [Cla99].

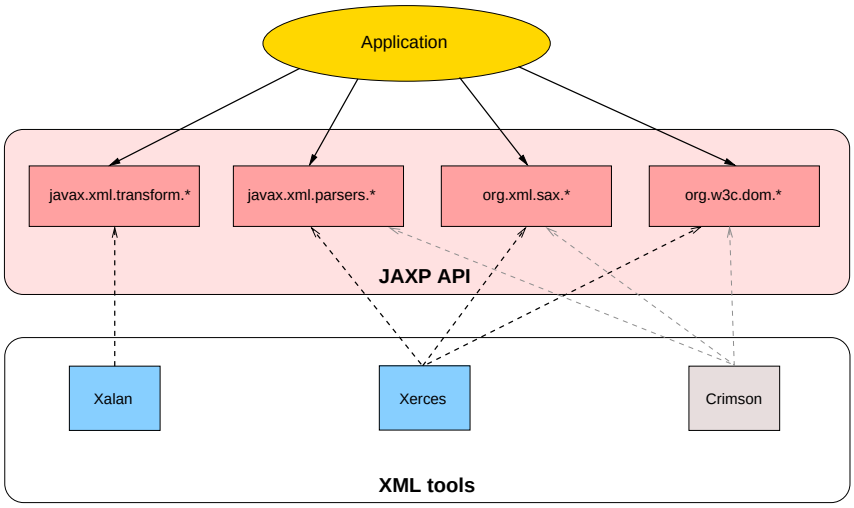

Figure 5.1: Java API for XML Processing (JAXP)

The XML tools used in LoL@ are Apache Xerces and Apache Xalan. The *Apache Xerces* Java XML Parser [XER02] supports the XML 1.0 W3C recommendation, the XML Schema W3C recommendation (version 1.0), DOM level 1 and 2, and the SAX version 1 and 2 APIs. *Apache Xalan* [XAL02] is an XSLT processor. It implements the W3C recommendations for XSL Transformations and the XML Path Language (XPath).

5.2 Java Packages

As shown in Figure 5.2, the design of the LoL@ Server Core application has been mapped onto six Java packages.

at.ftw.C1.server.core.* This package contains 14 classes (est. 3500 LOC) that implement the handling of requests and the page generation process described in Section 4.2.

at.ftw.C1.server.core.handlers.* The classes in this package (14 classes, est. 2500 LOC) implement the handlers. Handlers were described in Section 4.4.

at.ftw.C1.server.datasources.* This package contains 22 classes (est. 4000 LOC) that implement the wrappers that are responsible for the connection to the different data sources. Wrappers were described in Section 4.3.

at.ftw.C1.server.cache.* This package's classes (4 classes, 1000 LOC) implement the caching system and the cache manager described in Section 4.5.

at.ftw.C1.server.helpers.* This package contains helper classes (9 classes, 1500 LOC). `cInformation` was described in Section 4.3.4.1. `cPoIhelper`, `cFileFetcher`, and

`cStringSplitter` were described in Section 4.3.1. `cJDOMhelper` provides convenience methods for XML manipulation. `cJpgDimension` provides a way to determine the dimensions of files in JPEG format. The other classes in this package were used for debugging during the implemenation phase.

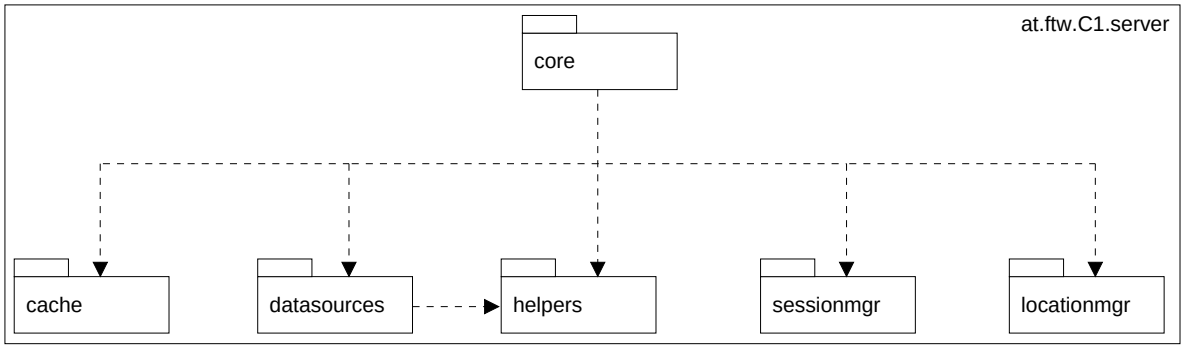

Figure 5.2: Package Diagram of the Server Core Application

5.3 XSL Stylefiles

This section discusses special problems of producing HTML code for small screens with XSL transformations. After that, the communication from the Server Core to the Terminal Core application – which is implemented using JavaScript instructions encoded in the HTML pages and therefore handled in the stylesheets – is presented.

In order not to waste space on the small display, the HTML pages need to be designed carefully. Obviously, horizontal scrollbars shall never appear. Most of the HTML pages must fit in the available space: Vertical scrollbars are acceptable (resp. not avoidable) for screens that provide textual information about PoIs, but not for navigation screens.

As discussed in Section 4.1, HTML was designed to markup text in order to structure it, but not for pixel-accurate positioning of text items on a screen. The latter is necessary for small screen sizes. When implementing the layout of applications which are displayed on a standard desktop display, a few pixels are often an infinitesimal amount of space. When implementing the layout for an application displayed on a device with 260x120 pixels as in LoL@, a few pixels difference make for perceptible deviations. A simple linebreak can cause problems.

To build the site layout of LoL@, HTML tables which position the text items on the screen must be used. These tables must be nested and carefully arranged. The resulting HTML code can be rather complex.

With XSL it is possible to determine the number of characters of a text in advance and to do conditional processing based on that result. This provides a convenient way to define

the rendering rules for a certain text item or HTML table at runtime depending on a text's number of characters. This is used for:

- *Vertical space:* Some elements of a screen *must* fit in a single line. This is ensured by determining the element's number of characters and using a different CSS class attribute to render a title in a smaller font size if it otherwise would not fit into a single line.

- *Scrollbars:* Whether a vertical scrollbar will or will not appear on the screen has to be determined in advance before the width of the outermost of the nested HTML tables can be determined. If there is a scrollbar, the HTML table's width must be smaller.

The Terminal Core application needs certain parameters in order to set the LoL@ buttons (see Section 3.3.1) according to the user's application state. These parameters are delivered to the Terminal Core application by calling a JavaScript method in the `onLoad` attribute of the HTML <`body`> tag. XSL's features were exploited to determine which information to send to the Terminal Core application depending on the HTTP parameters. The HTTP parameters are added to the XML content data in the page generation step. The parameter `sid` must always be sent. If `pid` (PoI ID) or `tid` (Tour ID) were included in the client's request, these values are also necessary. Moreover, the Terminal Core application must be notified of error pages and similar events. XSL provides <`xsl:choose`>, the equivalent to case statements in other programming languages. Several <`xsl:choose`> are necessary to test all the conditions mentioned above and concatenate a string that contains all necessary parameters according to the conditions. This string is delivered to the Terminal Core application by calling a JavaScript method (with the string as parameter) in the `onLoad` attribute of the HTML <`body`> tag.

Calculations like these can cause complex XSL files. The stylefiles that define the layout rules for producing the HTML code used in the LoL@ demonstrator are rather voluminous (est. 1500 LOC).

Even using the same delivery language (HTML), documents need to be formatted in different ways for different screen sizes. Hence, effort was made to make the stylefiles reuseable to support devices that use HTML as output format, but another screen size. This was achieved by avoiding hard-coded values of the demonstrator device's display size in the stylesheets but rather using variables for the width and heigth of the display. Moreover, since HTML can not position text items absolutely it was also necessary to define rather informal characteristics like "How many lines fit on the screen if the font size is 10". To adapt the stylefiles to another screen size, it is possible to determine these characteristics for the new screen size and change the values of the XSL variables.

In addition, XSL stylesheets are used to present the user's preferences differently in different cases:

```
voice_set=1;mapdur_set=2500;
```

Figure 5.3: User Preferences (style = system)

- The *LoL@ users* need an easy to use HTML interface for accessing and manipulating their preferences (see picture (5) of Figure 3.6).
- The *Terminal Core application* needs access to the user's preferences for the resume functionality (see Section 3.4.8). The values are parsed for later processing. Here, the stylesheet is used to produce plain text output (see Figure 5.3), since the Terminal Core application is not based on XML.

The layout of both the Diary Download Page (see Section 3.4.6) and the LoL@ Cache Manager (see Section 4.5) is also implemented using XSL stylesheets. For an exemplary screenshot of a diary download page, see Figure 3.14. For an exemplary screenshot of the Cache Manager, see Figure 4.24.

5.4 How to Add New Data Sources

This section explains how to extend the LoL@ Server Core application to integrate new data sources. Three steps are necessary to integrate a new data source (Figure 5.4).

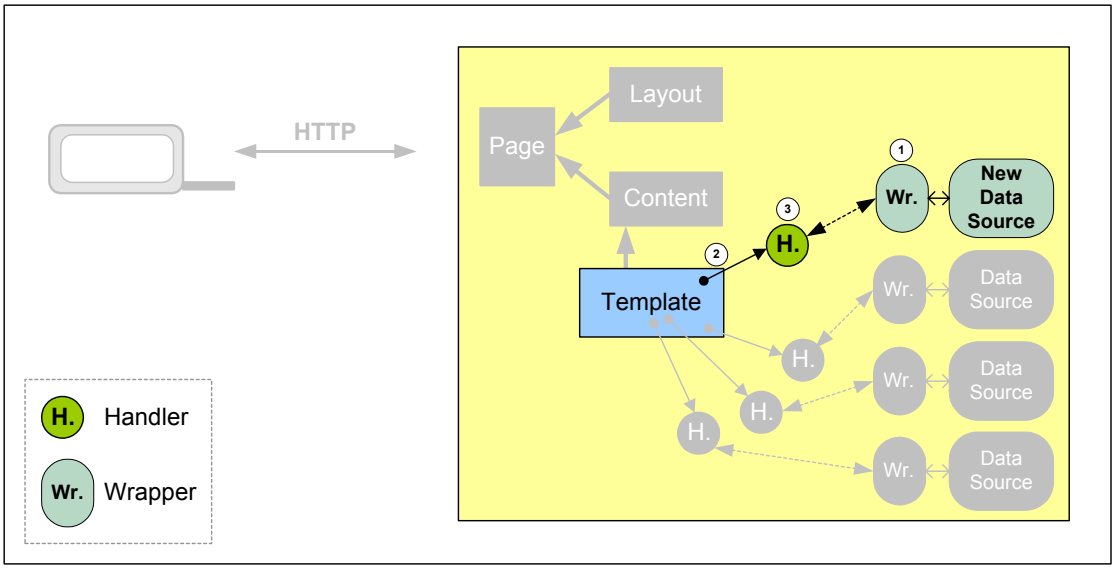

Figure 5.4: Steps Necessary to Add a New Data Source

These three steps are:

1. create a wrapper component (see Section 4.3) for the new data source.
2. create a new template instruction (see Section 4.2.1).
3. create a handler (see Section 4.4).

The *wrapper component* is responsible for taking certain input parameters, sending requests to the new data source, and converting the result data to XML format. If the data source already provides XML format (e.g., an XML database), the wrapper component will be very simple. For data sources that provide data in a format that is very different from XML, creating the wrapper will be more complicated. Hence, the complexity of the wrapper component depends on the original data format of the new data source. For example, if the new data source is an LDAP server, the wrapper must convert the results of LDAP queries (name/value pairs) to XML elements. For flexibility, it was decided not to force wrappers to provide certain methods by obliging them to implement a certain interface. The shortcomings of this decision are that the interface between wrapper and handler is not standardized, but in practice most of the existing wrappers provide a method `getElement()`.

In the next step, a new *instruction* must be created. This means that it is necessary to define a new XML tag that has a unique name. According to the possible input parameters that can be sent to the data source to define or constrain retrieval, the XML tag will have attributes resp. child elements or not. In the LDAP server example, the tag could be named <`ldap`> and have an attribute `query` that defines the LDAP querystring. The new instruction must then be inserted into one (or more) template(s) where appropriate according to the desired human-computer interaction flow.

Finally, the *handler* must be created. The handler has to implement the interface `iHandler` and therefore must provide a method `process()`. In this method, merely the call for the method provided by the wrapper component is inclosed. The parameters for the call are contained in the instruction. Hence, creating the handler is a straightforward task.

After creating these three entities, the new data source is integrated into the system. Obviously, it might also be necessary to extend the stylefiles with layout rules for the new data items.

6 Related Work

This chapter presents related work. First, an overview of research concerning mobile tour guides is given. Next, Web publishing frameworks based on XML/XSL technology are presented.

6.1 Mobile Tour Guides

The most important examples of mobile tour guides are Cyberguide and GUIDE. Both were developed in a scientific environment.

6.1.1 Cyberguide

The Cyberguide Project [AAH+97, CYB96] was started in 1996. Its aim was to build prototypes of handheld tour guides that provide information based on knowledge of a tourist's position and orientation. The architecture of Cyberguide consists of four components:

- *The map:* The user's position is updated automatically in the map and the map is scrolled to ensure that the user's current position remains on the visible portion of the map. In contrast to LoL@, Cyberguide does not support routing.
- *The information base*, which is stored at the client. It contains information about the part of the college campus[1] the system can be used in.
- *The positioning system* is based on infrared technology and therefore works only indoors. It uses TV remote control beacons which broadcast location IDs. The client's positioning subsystem is a custom infrared transceiver unit that translates the IDs into map locations and orientations.
- *The communications system*, which is used merely for sending feedback about the application to the developers via email.

[1] Georgia Institute of Technology

Since (mobile) computing technology has improved a lot since 1996, the architecture of Cyberguide, as well as the hardware, can not be compared to LoL@'s architecture and hardware. Nevertheless, there are two interesting findings concerning unsolved problems that we also encountered when designing and implementing LoL@:

First, as [LAAA96] states, "absolute positioning information throughout an entire space is not so important. It is far more useful to know what someone is looking at than to know someone's exact physical position and orientation." Second, [LKAA96] suggests to use an electronic compass or an inertial navigation system to find user orientation. This would in part solve the problem of not knowing what the user is currently looking at.

6.1.2 GUIDE

GUIDE [CDM+00, DMCB98] is a mobile tourist guide for tourists visiting the city of Lancaster. Visitors of the city can create and follow a tailored tour of the city. GUIDE is based on the usage of portable PCs[2] as terminals and WaveLAN as data delivery method. Determination of the user's location is done on the cell level.

GUIDE assists users in finding the way to certain tourist attractions (routing). Unlike in LoL@, the textual routing information provided is not comprised of information about lengths, directions, and names of street sections, but of more general descriptions that help the user to orientate him- oder herself using objects in the environment (e.g., significant buildings).

Information is broadcast by the cell base stations to the terminals either as part of a regular scheme or in response to user requests. Broadcasting brings problems with it: If users move while reading, it can happen that they read information pertaining to a cell in which they no longer reside. A possible solution would be to broadcast the information pertaining to the new cell when the user enters this zone, but this would overwrite the information the user is currently reading.

The broadcasting approach – which was the exclusive mode of information retrieval in the first version of GUIDE – entails that in contrast to LoL@, the information presented to the GUIDE users is restricted due to their context. An expert walkthrough of the system revealed that constraining the user's access to information based on their location can be frustrating for visitors if the information they require can not be accessed because it is not deemed to be of sufficient relevance to the area concerned. In a later release of GUIDE, this problem was tackled by including search facilities that allow searches in the whole information base, irrespective of the user's current location.

[2]The end-user terminal used for GUIDE is a pen-based tablet PC with a resolution of 800 by 600 pixels.

6.2 XML/XSL-based Web Publishing Tools

There is an uncountable number of tools for Web publishing available. In this section, three Web publishing frameworks that are based on XML/XSL technology – namely Apache Cocoon, MyXML, and Apache AxKit – are discussed. All of them are Open Source software.

6.2.1 Apache Cocoon

Cocoon [Coc02] is an Open Source Web publishing framework developed as a part of the Apache Software Foundation's XML project. The Cocoon Project was started in 1998. Initially, its goal was to implement a content management system for the homepages of all subprojects of the Apache Project. Cocoon 1 was based on the DOM Level 1 API. This approach severely limited scalability.

Consequently, Cocoon 2 is a complete rewrite of the original Cocoon application and is now based on the concept of component pipelines that pass SAX events to describe the process of publishing content to the Web. Each processing step has well-defined behavior coupled with fixed inputs and outputs.

The three types of pipeline components are generators, transformers, and serializers. A Cocoon pipeline is composed of one generator, zero or more transformers, and one serializer.

Generators (and readers). *Generators* read an XML data source and produce a series of SAX events which are then passed into the pipeline. Cocoon interacts with many data sources (filesystems, relational database management systems, native XML databases, ...). *Readers* access external resources as well, but copy them directly to the HTTP response instead of producing SAX events. Readers are used for serving static files (e.g., images and CSS files).

Transformers (and actions). *Transformers* consume and produce SAX events and execute the main processing steps in a Cocoon pipeline. They accept SAX events as input, perform some processing based on the input, and then pass the results to the next component of the pipeline as SAX events. The most important transformer is the XSLT transformer that feeds the SAX events to an XSLT processor to perform an XSLT transformation. *Actions* allow to integrate additional, often custom-built, dynamic behavior into a pipeline and are used for carrying out tasks like form validation, sending mail, etc.

Serializers. *Serializers* consume SAX events and produce a response suitable for the Web client. There are serializers for many different output formats like HTML, WML, PDF, SVG (Scalable Vector Graphics), RTF, and more.

Conditional processing inside the pipelines is possible with matchers and selectors. *Matchers* are equivalent to `if` statements and can use wildcards to define their conditions. They

are used to test whether a particular pipeline should be entered. *Selectors* are similar to `if-then-else` statements and are used to create conditional sections within a pipeline.

The processing control flow within a Cocoon Web application – its business logic – is defined in so-called *sitemaps*, which are configuration files in XML format. In these sitemap files, components – generators, readers, and transformers – are declared before being used in pipelines. Subsequently, pipelines are defined using these declared components.

Another important feature created by the Apache Cocoon project is XSP (eXtensible Server Pages) – a technology that enables the generation of dynamic XML content. XSP provides a way for showing a certain (legacy) data source through an XML interface. XSPs are files that contain programming language code (enclosed in XML tags within a special namespace) used to implement data retrieval actions, and XML tags that are placed around the code to mark up the retrieved data with its meaning. At runtime, XSPs are compiled[3] into Cocoon Generators that generate SAX events. Mixing programming language code and XML markup in a single file has its shortcomings. First, since XSP pages are XML files, programmers must avoid using XML reserved characters like < and & in the code, and must ensure well-formedness of the code. Second, and more serious: once code starts to mix with XML markup, code as well as markup gets hard to maintain. This problem was tackled by introducing the concept of so-called *logicsheets*. A logicsheet is a library of custom elements that can be added to XSP pages. Since logicsheets can be called from multiple XSP pages, this enables code reuse. Logicsheets are implemented as XSLT stylesheets that include Java code. Cocoon includes some built-in logicsheets (e.g., for database access).

To improve performance, the Cocoon architecture provides a caching system. Site resources, even those which are dynamically generated (and implemented to be cache-aware), can be cached.

For connectivity to other applications, Cocoon 2 ships with servlet and command line connectors. Using the servlet connector, Cocoon can be called from a servlet engine or an application server. The command line interface can be used to generate static Web sites, or parts of Web sites that are static, as a batch process.

At the time the decision about which tool the LoL@ Server Core application should be built on was made, Cocoon 2 was not available in a stable release. Furthermore, Cocoon was deemed to be too heavy-weight for our needs.

6.2.2 MyXML

The MyXML Web Development Kit [MyX02, KJKS01] was created by the Distributed Systems Group of the Information Systems Institute at the Technical University of Vienna. The major contribution of the MyXML WDK is the clear separation of content, layout, and business logic of a Web site.

[3] using XSL transformations

The content is defined in so-called MyXML documents which are files in XML format. The MyXML namespace provides template elements which support the dynamic generation of content at runtime. Using these template elements, it is possible to define database queries and access CGI parameters within MyXML documents. In addition, the notion of variables and loops is supported. The layout of MyXML documents is defined in XSL stylesheets. All stylesheets must import the MyXML stylesheet which handles the elements of the MyXML namespace.

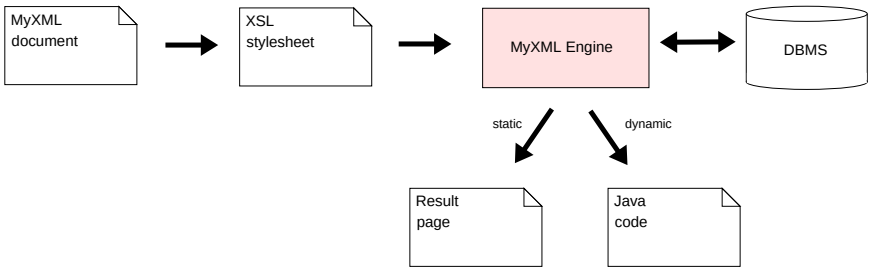

Figure 6.1: MyXML Template Engine

The MyXML engine is responsible for the generation of result files (Figure 6.1) from the input files described above. First, the well-formedness of the MyXML document is checked and the XSL stylesheet is applied to the document. This step adds layout information. In the next stage, the MyXML engine resolves the tags from the MyXML namespace. If it encounters a tag that has to be resolved dynamically (e.g., a CGI parameter), it creates Java code appropriate for producing the result page when given the necessary input parameters at runtime. If the engine can resolve all tags at compile time, a static result page is created.

MyXML was not used for the LoL@ Server Core application, because although it has support for database connections via JDBC, it has no support for the datasources used in LoL@.

6.2.3 AxKit

AxKit [AxK02] has recently become an official Apache project under the XML projects umbrella. It provides on-the-fly conversion from XML to any format. Basically, AxKit provides much of the same functionality as Cocoon (see Section 6.2.1), but unlike Cocoon which is Java-based, AxKit is built in Perl. Using mod_perl (see Section 5.1.1 or [ASF02]), it integrates tightly with the Apache Web server. This has performance benefits, especially for AxKit's caching facilities: When delivering cached results AxKit runs at about 80% of the speed of the Apache Web server. In order to deliver cached results, AxKit just tells the Apache Web server where to find the cached file.

The pipelining technique that AxKit uses allows content to be converted to a presentable format in stages. The basic concept of using a pipeline for the process of publishing content

to the Web was already introduced in Section 6.2.1 and will therefore be omitted here. Obviously, the major difference is that dynamic Web components are built using Perl instead of using Java.

One feature of AxKit that is not offered by Cocoon is *XPathScript*. This technology provides a way to embed Perl code and XPath expressions within HTML files. Although this approach mixes content and layout, it is worth noting. There is an API for accessing and transforming XML sources. Results of XPath queries can be either output directly or assigned to Perl variables. Using the special data structure `$->t`[4], additional parameters for the transformation (e.g., which output to produce before and after the transformed element) can be specified.

Another notable AxKit feature is *gzip compression*. If this feature is turned on and a client's request includes the HTTP header `Accept-Encoding` with appropriate values, AxKit will compress the result pages using Compress::Zlib before sending them to the client.

Furthermore, AxKit includes an *XSP* engine. As already mentioned in Section 6.2.1, XSP was originally invented by the Apache Cocoon team. AxKit provides the same technology, except for the fact that logicsheets are implemented using Perl. There are several logicsheets for AxKit's XSP engine available on CPAN[5] (e.g., AxKit::XSP:Param which allows to read form and querystring parameters within an XSP page).

Since a Java-based approach had to be used for the LoL@ Server Core application, using AxKit was not possible.

[4]The "t" stands for transformation.

[5]CPAN, the Comprehensive Perl Archive Network, provides lots of Perl modules and can be accessed at `http://cpan.perl.org`.

7 Evaluation and Future Work

7.1 Evaluation

This thesis presented the content delivery system for heterogeneous data sources via XML and HTTP that is used in the LoL@ UMTS application. This section evaluates the application with regards to the requirements specified in Section 1.2: usability, response time, maintainability, device independence, extensibility, and integration of legacy data.

Device independence. The LoL@ Server Core application supports the presentation of LoL@'s content data in various output formats. To achieve this, the application's content data must be strictly separated from any layout information. Result pages suitable for displaying on the client's viewing device are generated by applying layout information to content data. This is the responsibility of the presentation logic component, which is the last of several entities involved in the server's page generation process. Content negotiation mechanisms as described in Section 7.2 are not yet supported. The stylefiles that define the layout rules for producing the HTML code used in the LoL@ demonstrator are rather voluminous (est. 1500 LOC) due to two reasons: First, since HTML was not designed for pixel-accurate positioning but the latter is necessary for small screen sizes, the HTML code is rather complex. Second, the stylesheet contains logic of its own (see Section 5.3). Hence, effort was made to make the stylefiles reuseable[1] by avoiding hard-coded values of the demonstrator device's display size in the stylesheets but rather using variables.

Maintainability. Modifications and updates of the LoL@ application's look and feel are possible easily and independently of each other. The human-computer interaction flow ("the feel") can be changed by adapting the XML templates that define it: the instructions included in the templates can be (re)moved and extended. In addition, the instructions' parameters that control content retrieval at a fine-grained level can be adapted. This can be done by every person who has a basic knowledge of XML and a documentation of the LoL@ Server Core application. The layout of LoL@ ("the look") can be altered by modifying the stylesheets that define it. To accomplish this task, a person with knowledge of XSL is needed.

[1] to support devices that use HTML as output format, but another screen size

Extensibility./Integration of legacy data. The selected approach supports the integration of every kind of data source. To integrate a data source, a wrapper component for the new data source must be created. The complexity of the wrapper component depends on the original data format of the new data source. As outlined above, changing the human-computer interaction flow to integrate the new data source's data items into the application is possible with little effort, because it is defined in just one single part of the system. A detailed description about the steps necessary to integrate a new data source into the system can be found in Section 5.4.

Usability. To evaluate whether the user interface design is successful with respect to the human-computer interaction design issues specified in Section 3.2, field trials will be conducted. The field trials will be carried out by ten groups that consist of two persons each: a computer layperson and a computer expert. There will be two series of tests. The first test series will take place in a lab setting. The test setup is as follows: The layperson interacts with LoL@ and tries to accomplish a pre-defined task using a "thinking aloud" protocol. Meanwhile, the expert videotapes the layperson. A mirror is used to be able to tape the LoL@ screen and the layperson's facial expressions at the same time. After finishing the test, the expert uses qualitative methods (semistructured interview) to interview the layperson based on the taped video and asking the layperson to comment on his/her feelings about the application in certain situations. The second test series will take place outdoors in the first district of Vienna. The test setup will be the same except that no mirror will be used and that the expert will not film the the LoL@ screen, but layperson's body language and facial expressions while interacting with LoL@.

In addition, application performance tests to evaluate the response time of the system under production environment conditions will be done. Detailed information about the performance test specification can be found in [Ane02]. During the test phase conducted so far, the **response time** of the system was satisfactory.

7.2 Future Work

During the design and implementation of the LoL@ Server Core application, two major areas for improvement and further research were identified. On the one hand, the content delivery system could be extended: with content negotiation mechanisms for mobile devices, and with administration interfaces. On the other hand, the LoL@ mobile tourist guide could offer additional features to the users. Consequentially, adding these features would entail extensions of the content delivery system.

7.2.1 Content Delivery System

As mentioned throughout the thesis, different Web-enabled devices have different input, output, hardware, software, network and browser capabilities. In order for a Web-based application to provide optimized content to different clients it requires a device profile – which means that it needs a description of the capabilities of the client. Such a description is also known as the *delivery context*. Two compatible standards have been created for describing delivery context:

- The *Composite Capabilities / Preferences Profile (CC/PP)* [CCP02] was created by the W3C.
- The *User Agent Profile (UAProf)* [UAP01] was created by the WAP Forum.

Both standards are RDF vocabularies. The Resource Description Framework (RDF, [RDF02]) was developed by the W3C[2] to promote metadata within Web resources. It is a framework for describing metadata in a machine-processable form (XML format) and for interchanging these metadata. RDF specifies the metadata's syntax, but not the semantics. The latter is defined in so-called vocabularies that can be defined individually.

There is an Open Source implementation of CC/PP und UAProf available: DELI [DEL02] – the delivery context library – provides an API that can be used by Java-based Web applications to determine the delivery context of a client device using CC/PP or UAProf. Integrating DELI into LoL@ to determine the capabilities of a LoL@ terminal and preparing the content according to these capabilities would be an interesting task.

Another possible extension of the LoL@ Server Core application would be to add a Web-based administration interface for content management of the application's data. This task would merely be implementation work analogous to the tour diary implementation. Basically, new templates for the content management interface and new stylesheets that provide HTML forms to enter data would be necessary. Furthermore, an administration interface that allows to manipulate the LoL@ templates would ease content managers' work concerning modifications of the human-computer interaction flow. Of course, access control mechanisms for both these interfaces must be installed.

7.2.2 LoL@ Application

[EPS+01] describes a system called "GeoNotes" where users can put their own "labels" everywhere and share them with others. Much like attaching post-it notes on physical places, the system would allow users to create their own annotations and share these annotations with other users by virtually attaching them to physically existing entities like buildings, street signs, or trees. [EPS+01] also discusses the social impacts of the system.

[2] RDF is a W3C recommendation since February 1999.

Especially LoL@ users – tourists, which have leisure time and are strangers in the city – will have time for and interest in communicating with other tourists. This system would provide room to communicate about positive and negative experiences. If accepted by users, this kind of *"virtual word of mouth"* would help tourists to find the city's hidden nice places and to avoid tourist traps.

Another enhancement would be to allow users to create their own tour through the city. Using LoL@'s information screens (see Section 3.4.2), users could get an overview of the tourist attractions of Vienna's first district and subsequently create a *user-defined tour* according to their personal likings. In addition, another useful feature would be to take the following factors into account:

- Opening and closing times of attractions, as well as the best time to visit an attraction.
- Distance between attractions and the most aesthetic route between them. This factor must only be considered for user-defined tours.

LoL@ could then adapt pre-defined as well as user-defined tours according to these factors. This would prevent tourists from facing closed museum doors and from taking a hike instead of a walk through the city.

A XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://lola.ftw.at/ns/templates"
            xmlns="http://lola.ftw.at/ns/templates"
            elementFormDefault="qualified"
>

  <xsd:element name="data">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="template" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="header" type="MyComplexHeader" />
              <xsd:element name="content" type="MyComplexContent" />
            </xsd:sequence>
            <xsd:attribute name="sid" use="required">
              <xsd:simpleType>
                <xsd:restriction base="xsd:integer">
                  <xsd:minInclusive value="100" />
                  <xsd:maxInclusive value="9999" />
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="resumeable" type="MyCheckType" use="required" />
            <xsd:attribute name="cacheable" type="MyCheckType" use="required" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="MyComplexHeader">
    <xsd:sequence>
      <xsd:choice>
        <xsd:sequence>
          <xsd:element name="title" type="xsd:string" />
          <xsd:element name="icon" type="MyIconType" minOccurs="0" />
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element ref="SQL" />
        </xsd:sequence>
      </xsd:choice>
      <xsd:element name="posttitle" type="xsd:string" minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="MyComplexContent">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element ref="table" />
        <xsd:element ref="list" />
        <xsd:element ref="textual" />
        <xsd:element ref="generic" />
      </xsd:choice>
      <xsd:sequence minOccurs="0">
        <xsd:element name="resumeable" type="xsd:string" />
      </xsd:sequence>
      <xsd:sequence minOccurs="0">
        <xsd:element name="resume" type="xsd:string" />
        <xsd:element ref="options" />
      </xsd:sequence>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="table">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="button" minOccurs="3" maxOccurs="4" />
```

```
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="list">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="SQL" minOccurs="0" maxOccurs="unbound" />
      <xsd:element ref="linktext" />
      <xsd:element ref="linkpara" maxOccurs="unbound" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="textual">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="line" type="xsd:string" minOccurs="0"
                                                 maxOccurs="unbound" />
      <xsd:element ref="SQL" minOccurs="0" maxOccurs="unbound" />
      <xsd:element ref="file" minOccurs="0" />
      <xsd:element name="seealso" minOccurs="0">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="SQL" maxOccurs="unbound" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="generic">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="mediascreen">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="file" />
            <xsd:element ref="SQL" minOccurs="0" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="fileupload" type="xsd:string" />
      <xsd:element name="routing">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="para" maxOccurs="unbound" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="diary">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="item" maxOccurs="unbound" />
            <xsd:element name="currenttime" type="xsd:string" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<xsd:element name="SQL">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="text" />
      <xsd:element ref="para" minOccurs="0" />
      <xsd:element ref="text" minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="button">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="link" />
    </xsd:sequence>
    <xsd:attribute name="seqnr" use="required" />
      <xsd:simpleType>
        <xsd:restriction base="xsd:integer">
          <xsd:minInclusive value="1" />
          <xsd:maxInclusive value="4" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  <xsd:complexType>
</xsd:element>
```

```
<xsd:element name="link">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="linktext" />
      <xsd:element ref="linkpara" maxOccurs="unbound" />
    </xsd:sequence>
    <xsd:attribute name="check" type="MyCheckType" use="required" />
  </xsd:complexType>
</xsd:element>

<xsd:element name="linktext">
  <xsd:complexType>
    <xsd:choice>
      <xsd:sequence>
        <xsd:element ref="name" />
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element ref="SQL" />
      </xsd:sequence>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<xsd:element name="linkpara">
  <xsd:complexType>
    <xsd:choice>
      <xsd:sequence>
        <xsd:element ref="name" />
        <xsd:choice>
          <xsd:sequence>
            <xsd:element name="var" type="xsd:int" />
          </xsd:sequence>
          <xsd:sequence>
            <xsd:element ref="para" />
          </xsd:sequence>
        </xsd:choice>
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element ref="fillinID" />
      </xsd:sequence>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<xsd:element name="fillinID">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="name" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="name" type="MyIDType" />
<xsd:element name="text" type="xsd:string" />
<xsd:element name="para" type="MyIDType" />

<xsd:element name="file">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="SQL" />
    </xsd:sequence>
    <xsd:attribute name="type" type="MyFileType" use="required" />
  </xsd:complexType>
</xsd:element>

<xsd:element name="options">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="item" maxOccurs="unbound" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="item">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="caption" type="xsd:string" />
      <xsd:element name="name">
        <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="xsd:string">
              <xsd:attribute name="datatype" type="MyDataType" />
            </xsd:extension>
          <xsd:simpleContent>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="value" type="xsd:string">
```

```
        <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="xsd:string">
              <xsd:attribute name="type" type="xsd:string"
                                         fixed="default" />
            </xsd:extension>
          <xsd:simpleContent>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="visible" type="MyCheckType" use="required" />
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="MyIDType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="sid" />
    <xsd:enumeration value="tid" />
    <xsd:enumeration value="pid" />
    <xsd:enumeration value="iid" />
    <xsd:enumeration value="did" />
    <xsd:enumeration value="rid" />
    <xsd:enumeration value="GeoX" />
    <xsd:enumeration value="GeoY" />
    <xsd:enumeration value="strid" />
    <xsd:enumeration value="radius" />
    <xsd:enumeration value="vv" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="MyFileType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="text" />
    <xsd:enumeration value="audio" />
    <xsd:enumeration value="video" />
    <xsd:enumeration value="image" />
    <xsd:enumeration value="landmark" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="MyDataType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="boolean" />
    <xsd:enumeration value="integer" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="MyCheckType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="yes" />
    <xsd:enumeration value="no" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="MyIconType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="m_cafe.gif" />
    <xsd:enumeration value="m_denkmal.gif" />
    <xsd:enumeration value="m_gasse.gif" />
    <xsd:enumeration value="m_gebaeude.gif" />
    <xsd:enumeration value="m_haus.gif" />
    <xsd:enumeration value="m_hist_museen.gif" />
    <xsd:enumeration value="m_kirche.gif" />
    <xsd:enumeration value="m_museum.gif" />
    <xsd:enumeration value="m_park.gif" />
    <xsd:enumeration value="m_platz.gif" />
    <xsd:enumeration value="o_amhof.gif" />
    <xsd:enumeration value="o_burgtheater.gif" />
    <xsd:enumeration value="o_freyung.gif" />
    <xsd:enumeration value="o_hofburg.gif" />
    <xsd:enumeration value="o_josefsplatz.gif" />
    <xsd:enumeration value="o_michaelerkirche.gif" />
    <xsd:enumeration value="o_parlament.gif" />
    <xsd:enumeration value="o_pestsaeule.gif" />
    <xsd:enumeration value="o_rathaus.gif" />
    <xsd:enumeration value="o_stephansdom.gif" />
    <xsd:enumeration value="o_universitaet.gif" />
    <xsd:enumeration value="o_votivkirche.gif" />
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

Bibliography

[AAB+00] Joan Aliprand, Julie Allen, Joe Becker, Mark Davis, Michael Everson, Asmus Freytag, John Jenkins, Mike Ksar, Rick McGowan, Lisa Moore, Michel Suignard, and Ken Whistler. *The Unicode Standard, Version 3.0*. Addison-Wesley, January 2000. `http://www.unicode.org/unicode/uni2book/u2.html`.

[AAH+97] Gregory D. Abowd, Christopher G. Atkeson, Jason Hong, Sue Long, Rob Kooper, and Mike Pinkerton. Cyberguide: A Mobile Context-Aware Tour Guide. In *ACM Wireless Networks*, pages 3:421–433, 1997.

[ACG] Micrografx. *ActiveCGM Format*. `http://www.micrografx.com/icap/activecgm.asp`.

[AK01] Hermann Anegg and Harald Kunczier. Report 5.3.d Platform LCS Service Specification. Technical report, FTW Project C1, 2001.

[Ane01] Hermann Anegg. Report 6.4.a Terminal LCS Module Specification. Technical report, FTW Project C1, 2001.

[Ane02] Hermann Anegg. Report 8.1.b Field Trial Specification. Technical report, FTW Project C1, 2002.

[ASF02] The Apache Software Foundation. *mod_perl: The Apache/Perl Integration Project*, 2002. `http://perl.apache.org/`.

[AxK02] Apache XML Project. *AxKit XML Application Server*, 2002. `http://www.axkit.org/`.

[BDFR99] Jens Bergqvist, Per Dahlberg, Henrik Fagrell, and Johan Redström. Exploring Proximity Awareness, 1999. `http://citeseer.nj.nec.com/288241.html`.

[BFGPU01] Beatrix Brunner-Friedrich, Georg Gartner, Andreas Pammer, and Susanne Uhlirz. Report 4.2.a Routing Concept. Technical report, FTW Project C1, 2001.

[BFJ+01] George Buchanan, Sarah Farrant, Matt Jones, Harold Thimbleby, Gary Marsden, and Michael Pazzani. Improving Mobile Internet Usability. *Proc. of The 10th International WWWeb Conference (WWW10)*, 2001.

[BM01] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes*. World Wide Web Consortium, May 2001. `http://www.w3.org/TR/xmlschema-2/`.

[BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. World Wide Web Consortium, October 2000. `http://www.w3.org/TR/2000/REC-xml-20001006`.

[CCP02] World Wide Web Consortium. *Composite Capabilities/Preferences Profile (CC/PP) Working Group Public Home Page*, April 2002. `http://www.w3.org/Mobile/CCPP/`.

[CDM+00] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrain Friday, and Christos Efstratiou. Developing a Context-aware Electronic Tourist Guide: Some Issues and Experiences. In *Proc. of CHI'00*, pages 17–24, 2000.

[CDMF00] Keith Cheverst, Nigel Davies, Keith Mitchell, and Adrian Friday. Experiences of developing and deploying a context-aware tourist guide: the GUIDE project. In *Mobile Computing and Networking*, pages 20–31, 2000. `http://citeseer.nj.nec.com/cheverst00experiences.html`.

[CGI] NCSA HTTPd Home Page. *The Common Gateway Interface*. `http://hoohoo.ncsa.uiuc.edu/cgi/overview.html`.

[Cla99] James Clark. *XSL Transformations (XSLT) Version 1.0*. World Wide Web Consortium, November 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[CMD01] Keith Cheverst, Keith Mitchell, and Nigel Davies. Investigating Context-aware Information Push vs. Information Pull to Tourists. In *Proc. of Mobile HCI 2001*, 2001.

[Coc02] The Apache Software Foundation. *Cocoon Homepage*, 2002. `http://xml.apache.org/cocoon/index.html`.

[COO99] Netscape Communications Corp. *Persistent Client State HTTP Cookies*, 1999. `http://www.netscape.com/newsref/std/cookie_spec.html`.

[COS01] Jason Hunter. *com.oreilly.servlet.**, 2001. `http://www.servlets.com/cos/index.html`.

[CSS02] World Wide Web Consortium. *Cascading Style Sheets (CSS) Homepage*, May 2002. `http://www.w3.org/Style/CSS/`.

[CYB96] College of Computing of the Georgia Institute of Technology. *Cyberguide Project Page*, 1996. `http://www.cc.gatech.edu/fce/cyberguide/`.

[DEL02] Mark H. Butler. *DELI: Delivery Context Library*, 2002. `http://sourceforge.net/projects/delicon/`.

[DMCB98] Nigel Davies, Keith Mitchell, Keith Cheverst, and Gordon Blair. Developing a Context Sensitive Tourist Guide, 1998. `http://citeseer.nj.nec.com/davies98developing.html`.

[EPC+00] Peter Eichinger, Denes Paal, Laura Cottatellucci, Harald Kunczier, Gregor Erbach, Erwin Postmann, Qi Guan, Peter Wenzl, Albert Schauer, Andreas Krenn, Gerhard Schimon, and Günther Pospischil. Report 2.1.a Demonstrator Architecture. Technical report, FTW Project C1, 2000.

[EPC+01] Peter Eichinger, Denes Paal, Laura Cottatellucci, Harald Kunczier, Erwin Postmann, Peter Wenzl, Albert Schauer, Andreas Krenn, and Günther Pospischil. Report 2.3.a Mobile Network Domain. Technical report, FTW Project C1, 2001.

[EPS+01] Fredrik Espinoza, Per Persson, Anna Sandin, Hanna Nyström, Elenor Cacciatore, and Markus Bylund. Geonotes: Social and Navigational Aspects of Location-Based Information Systems. In *Proc. of Ubicomp 2001*, pages 2–17, 2001.

[Fal01] David C. Fallside. *XML Schema Part 0: Primer*. World Wide Web Consortium, May 2001. `http://www.w3.org/TR/xmlschema-0/`.

[FIE99] World Wide Web Consortium. *HTML 2.0 Materials: Forms*, September 1999. `http://www.w3.org/MarkUp/html-spec/html-spec_8.html`.

[FK00] Beatrix Friedrich and Roman Kopetzky. Report 4.3.a Tour Specification. Technical report, FTW Project C1, 2000.

[FMS01] Shlomit Ritz Finkelstein, Stephane Maes, and Lalitha Suryanarayana. *Device Independence Principles*. World Wide Web Consortium, September 2001. `http://www.w3.org/TR/2001/WD-di-princ-20010918/`.

[For00] David Forman. Mobile convergence - ultraportables come together. *Laptop*, pages 52–60, August 2000.

[FZ94] George Forman and John Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, April 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GNU02] The GNU Project. *GNU wget*, March 2002. `http://www.gnu.org/software/wget/wget.html`.

[Gro01] Object Management Group. Common Object Request Broker Architecture (CORBA/IIOP) 2.6 specification, December 2001. `http://www.omg.org/cgi-bin/doc?formal/01-12-01`.

[Hal00] Marty Hall. *Core Servlets and JavaServer Pages*. Sun Microsystems Press, 2000. `http://pdf.coreservlets.com/`.

[HBC+92] Hewett, Baecker, Card, Carey, Gasen, Mantei, Perlman, Strong, and Verplank. Definition of the term "human-computer interaction". *ACM SIGCHI Curricula for Human-Computer Interaction*, page 5, 1992. `http://sigchi.org/sigchi/cdg/cdg2.html`.

[HGM02] Günter Hake, Dietmar Grünreich, and Liquiu Meng. *Kartographie*. de Gruyter, 2002.

[HHW+00] Arnaud Le Hors, Philippe Le Hegaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. *Document Object Model (DOM) Level 2 Core Specification*. World Wide Web Consortium, November, 2000. `http://www.w3.org/TR/DOM-Level-2-Core/`.

[HMP+01] Manuel Horvath, Elke Michlmayr, Günther Pospischil, Martina Umlauft, and Peter Wenzl. Report 5.5.b LoL@ Core Design. Technical report, FTW Project C1, 2001.

[HSSR99] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. *SIP: Session Initiation Protocol*. Internet Engineering Task Force, March 1999. `ftp://ftp.isi.edu/in-notes/rfc2543.txt`.

[IEE90] IEEE, editor. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, 1990.

[IGM+97] U. Irvine, J. Gettys, J. Mogul, H. Frystyk, and Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force, January 1997. `http://www.ietf.org/rfc/rfc2068.txt`.

[J2E01] Sun Microsystems, Inc. *Java 2 Platform, Enterprise Edition*, August 2001. `http://java.sun.com/j2ee/`.

[J2M00] Sun Microsystems, Inc. *Java 2 Platform, Micro Edition*, September 2000. `http://java.sun.com/j2me/`.

[JAP00] Sun Microsystems, Inc. *JavaPhone API Version 1.0 Specification*, April 2000. `http://java.sun.com/products/javaphone/`.

[JAX02] Sun Microsystems, Inc. *Java API for XML Processing (JAXP)*, 2002. `http://java.sun.com/xml/jaxp/index.html`.

[JDB02] Sun Microsystems, Inc. *Java Database Connectivity (JDBC) 3.0*, February 2002. `http://java.sun.com/products/jdbc/`.

[JDO02] Jason Hunter and Brett McLaughlin. *JDOM: Java Document Object Model*, 2002. `http://www.jdom.org`.

[JSE02] The Apache Software Foundation. *The Apache JServ Project*, 2002. `http://java.apache.org/jserv/`.

[KJKS01] Engin Kirda, Mehdi Jazayeri, Clemens Kerer, and Markus Schranz. Experiences in engineering flexible web services. *IEEE Multimedia (Special Issue on Web Engineering)*, Jan-Mar 2001.

[KP88] Glenn Krasner and Stephen Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming (JOOP)*, August/September 1988.

[KPP+01] Roman Kopetzky, Christian Ploninger, Günther Pospischil, Stefan Tampe, and Martina Umlauft. Report 5.6.a Content Database and User Data Specification. Technical report, FTW Project C1, 2001.

[LAAA96] Sue Long, Dietmar Aust, Gregory D. Abowd, and Christopher G. Atkeson. Cyberguide: Prototyping Context-Aware Mobile Applications. In *Proc. of CHI'96*, 1996. `http://www.cc.gatech.edu/fce/cyberguide/pubs/chi96-cyberguide.html`.

[Les99] Lawrence Lessig. *Code and Other Laws of Cyberspace*. Basic Books, 1999.

[Lev] Sami Levijoki. Privacy vs Location Awareness. `http://www.hut.fi/~slevijok/privacy_vs_locationawareness.htm`.

[LKAA96] Sue Long, Rob Kooper, Gregory D. Abowd, and Christopher G. Atkeson. Rapid Prototyping of Mobile Context-Aware Applications: The Cyberguide Case Study. In *Proc. of 2nd ACM International Conference on Mobile Computing and Networking (MobiCom '96)*, 1996. `http://www.csd.uch.gr/~markatos/papers/mobicom96-cyberguide.ps`.

[MEx01] 3rd Generation Partnership Project. *3GPP TS 23.057 V4.4.0 (2001-12)*, December 2001. `http://www.3gpp.org/ftp/Specs/2001-12/Rel-4/23_series/23057-440.zip`.

[MyX02] Information Systems Institute of the Technical University of Vienna, Distributed Systems Group. *MyXML Homepage*, 2002. `http://www.infosys.tuwien.ac.at/myxml/`.

[Nie99] J. Nielsen. Alertbox 31/10/1999: Graceful Degradation of Scalable Internet Services, 1999. `http://www.useit.com/alertbox/991031.html`.

[Nie00] Nielsen Norman Group. *WAP Usability Report: Field Study Fall 2000*, December 2000. `http://nngroup.com/reports/wap/`.

[Ovi99] Sharon Oviatt. Ten myths of multimodal interaction. *Communications of the ACM*, 42(11):74–81, 1999. `http://citeseer.nj.nec.com/oviatt99ten.html`.

[Par] Parlay Group. *Parlay Specification*. `http://www.parlay.org/`.

[PFL+01] Günther Pospischil, Beatrix Friedrich, Mirjanka Lechthaler, Georg Gartner, Martina Umlauft, Andreas Pammer, and Roman Kopetzky. Report 4.4.b Interactive Human Interface. Technical report, FTW Project C1, 2001.

[PGH01a] Günther Pospischil, Qi Guan, and Jan Hosp. Report 5.4.b Speech Server Connectivity. Technical report, FTW Project C1, 2001.

[PGH01b] Günther Pospischil, Qi Guan, and Jan Hosp. Report 6.2.a SIP User Agent Functional and Test Report. Technical report, FTW Project C1, 2001.

[PHH01] Günther Pospischil, Manuel Horvath, and Jan Hosp. Report 6.5.a Terminal Core External Connections (WinSIP, Map Viewer). Technical report, FTW Project C1, 2001.

[PJA00] Sun Microsystems, Inc. *PersonalJava Application Environment Specification Version 1.2a (Final)*, November 2000. `http://java.sun.com/products/personaljava/`.

[Pos01] Günther Pospischil. Report 2.2.a Specification of User Equipment Domain. Technical report, FTW Project C1, 2001.

[PUM02] Günther Pospischil, Martina Umlauft, and Elke Michlmayr. Designing LoL@, a Mobile Tourist Guide for UMTS. to be presented at MobileHCI 2002, September 2002.

[RDF02] World Wide Web Consortium. *Resource Description Framework (RDF)*, March 2002. `http://www.w3.org/RDF/`.

[RHJ99] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.01 Specification*. World Wide Web Consortium, December 1999. `http://www.w3.org/TR/html4/`.

[Ric00] K. W. Richardson. UMTS overview. *IEEE Electronics & Communication Engineering Journal*, June 2000.

[RMI02] Sun Microsystems, Inc. *Remote Method Invocation*, 2002. `http://java.sun.com/products/jdk/rmi/`.

[SAX02] David Brownell. *Simple API for XML (SAX)*, 2002. `http://www.saxproject.org/`.

[SER00] Sun Microsystems, Inc. *Java Servlet API*, February 2000. `http://java.sun.com/products/servlet/`.

[Syr01] Jari Syrjärinne. *Studies of Modern Techniques for Personal Positioning*. PhD thesis, Tampere University of Technology, 2001.

[TBMM01] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures*. World Wide Web Consortium, May 2001. `http://www.w3.org/TR/xmlschema-1/`.

[TOM02] The Apache Software Foundation. *The Apache Tomcat Project*, 2002. `http://jakarta.apache.org/tomcat/`.

[Tuo97] Juhani E. Tuovinen. Cognitive load and discovery learning. In *Proc. of AARE 1997 Conference*, 1997. `http://www.aare.edu.au/97pap/tuovj113.htm`.

[UAP01] WAP Forum. *WAG UAProf, WAP-248-UAPROF-20011020-a*, October 2001. `http://www1.wapforum.org/tech/documents/WAP-248-UAProf-20011020-a.pdf`.

[UPNM02] Martina Umlauft, Günther Pospischil, Georg Niklfeld, and Elke Michlmayr. LoL@, a Mobile Tourist Guide for UMTS. submitted to Journal of Information Technology & Tourism, April 2002.

[W3C01] World Wide Web Consortium. *Mobile Access – Working towards seamless Web access from mobile devices*, February 2001. `http://www.w3.org/Mobile/`.

[WAP] WAP Forum. *Wireless Application Protocol Specification*. `http://www.wapforum.org/`.

[Wen01] Peter Wenzl. Report 5.2.a HSS Specification. Technical report, FTW Project C1, 2001.

[WML] WAP Forum. *Wireless Markup Language version 1.3 Specification*. `http://www.wapforum.org/`.

[XAL02] The Apache Software Foundation. *Xalan XSLT Stylesheet Processor*, 2002. `http://xml.apache.org/xalan-j/index.html`.

[XER02] The Apache Software Foundation. *Xerces XML Parser*, 2002. `http://xml.apache.org/xerces-j/index.html`.