DISSERTATION

Architectures of Web Applications

Design and Implementation of Database backed Information Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

o.Univ.Prof. Dipl.-Ing. Dr.techn. Richard Eier
Institut für Computertechnik

und

o.Univ.Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
Institut für Informationssysteme

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik
von

Dipl.-Ing. Karl M. Göschka
Matrikelnummer: 8625510
Karl–Meißlstraße 7/17, 1200 Wien

Wien, August 1998 _____

Abstract

Information processing is the key issue of the 20th Century. Databases are designed to store information and the World Wide Web has turned out to be *the* place for the gathering and distribution of information. Some scientists even consider the Web itself to be a huge world-wide distributed database. While these two seem to be made for each other, there are inherent difficulties in linking them together due to almost 35 years of separate development resulting in quite different technologies. Nevertheless, good Web sites are database backed Web sites – it turned out that most real-life Web applications need an underlying database to be stable, flexible and scalable. Appropriate design methodologies are hence needed to implement complex functionality. Moreover, there are already many databases and legacy systems in existence which people want to connect to the Web with similar functionality as in classic implementations. Most of them rely on the proven technology of relational databases or on their object relational successors.

Consequently the main aim of this thesis is the integration of databases and the Web. Based on a short explanation of the differences between databases and the Web, a new design methodology is presented. The finite state machine model well known from static hypertext documents is remodelled to be suitable for dynamically generated hypertext. Groups of links are mainly used instead of forms to interact with the database. The key idea is to use an object-based client-server model to design the user interface layout, the middleware functionality and the database transactions in a homogenous way. Thereafter, the actual implementation of such a system is composed of several parts with potentially different techniques. For example, either a pure HTML interface with full functionality on the server side or a Java applet with greater functionality on the client side. A toolset has been implemented to automatically generate the different applications from a design language. The toolset itself is a Web application fostering unlimited Web collaboration. Using this technique, complete information systems can be built, including logical page flow and bidirectional crosslinks, multi lingual support and frames with different window modes. The design methodology further guarantees stable and robust applications with sophisticated user interactions, compatibility with almost any browser and flexible layout design separated from the application functionality.

The usability of Java is compared with the pure HTML approach, investigating different client-server tradeoffs and persistence frameworks. Object oriented design and implementation has some drawbacks with typical Web–based information systems. It is, however, shown how Java can supplement the HTML approach but not supplant it.

In addition to the differences in design and implementation, relational databases and the Web also differ considerably in the ways they can be searched. Universal relations and natural language interfaces known from database theory are combined with keyword searches known from the Web to define a metadata model for searching a database backed Web application. The key idea here is the separation of structural and contextual meaning of words. Based on this idea, generic interfaces for both Intelligent Software Agents and robots from search engines have been implemented.

Evaluation of the proposed methods has been carried out with two real-life applications implemented successfully with both pure HTML and Java. A quantitative analysis of these applications helps to decide between pure HTML and Java: For less than 200-300 user interactions pure HTML typically outperforms Java in terms of network traffic. A qualitative analysis of these two applications resulted in the first users' feedback being encouragingly positive. The techniques described in this thesis will help to continue to produce such positive results and to make the design of database backed Web applications faster, easier and less prone to error.

Kurzfassung

Die Beherrschung der Informationstechnologie ist eine der wichtigsten Herausforderungen des 20. Jahrhunderts. Datenbanken dienen der Speicherung von Information und das Web hat sich als *das* Medium für Suche und Verteilung von Information herauskristallisiert. In manchen Publikationen wird sogar das Web selbst als gewaltige, weltweit verteilte Datenbank angesehen. Obgleich Datenbanken und Web füreinander bestimmt zu sein scheinen, sind in fast 35 Jahren getrennter Entwicklung sehr unterschiedliche Technologien herangereift, die nicht so einfach kombiniert werden können. Dennoch sind nur Datenbank-basierte Web-Applikationen auch gute Web-Applikationen: Es hat sich im Laufe der Zeit herausgestellt, daß reale Anwendungen eine Datenbank benötigen, um stabil, flexibel und skalierbar zu sein. Daher werden angepaßte Design-Methoden benötigt, um komplexe Funktionalität implementieren zu können. Zusätzlich besteht die Forderung, auch alte Datenbanksysteme an das Web anzubinden, möglichst mit der bisher gewohnten Funktionalität. Die meisten dieser Systeme sind relational oder objektrelational.

Die vorliegende Dissertation hat sich daher die Integration von Datenbanken und Web zum Ziel gesetzt. Aufbauend auf einer kurzen Beschreibung der Unterschiede zwischen Datenbank- und Web-Technologie wird eine neue Design-Methodik vorgestellt. Dazu wird das Modell des Zustandsautomaten, welches aus dem Forschungsbereich statischer Hypertext-Dokumente wohlbekannt ist, an dynamisch generierten Hypertext angepaßt. Dabei werden Gruppen von Hyper-Links anstelle der sonst oft vorgeschlagenen Formulare verwendet. Dem liegt der Gedanke zugrunde, im Zuge des Designs ein homogenes, objektbasiertes Client-Server-Modell zu verwenden, um das User-Interface, die Funktionalität und die Datenbank-Transaktionen zu definieren. Die Implementierung eines solchen Systems wird dann aus verschiedenen Teilen zusammengesetzt, welche durchaus unterschiedliche Techniken verwenden können, etwa eine reine HTML–Oberfläche ohne Funktionalität im Vergleich zu einem Java-Applet mit mehr Funktionalität. Verschiedene Software-Werkzeuge wurden implementiert, um die verschiedenen Applikationen aus einer gemeinsamen Design-Sprache generieren zu können. Diese Software-Werkzeuge sind zudem selbst Web-Applikationen und ermöglichen somit unbegrenzte Zusammenarbeit über das Web selbst. Mit dieser Technik können komplette Informationssysteme inklusive logischer Hyper-Text-Strukturen, bidirektionaler Hyper-Links, Mehrsprachigkeit sowie Frame-Technik entworfen und implementiert werden. Die spezielle Design-Methodik garantiert zudem stabile und robuste Applikationen mit fortgeschrittenen User-Interaktionen, Kompatibilität zu nahezu jedem Browser und ein von der Funktionalität unabhängiges, flexibles Layout-Design.

Im Rahmen der Untersuchung verschiedener Client-Server–Strukturen und unterschiedlicher Persistenz-Frameworks werden die Verwendung von Java und reinem HTML miteinander verglichen. Objektorientierte Methoden zeigen einige Nachteile bei Design und Implementierung typischer Web–basierter Informationssysteme. Es wird gezeigt, wie Java die HTML-Lösung zwar unterstützen, aber derzeit nicht gänzlich ersetzen kann.

So verschieden Datenbanken und das Web in bezug auf Design und Implementierung sind, so unterschiedlich sind auch die Möglichkeiten der Informationssuche für diese beiden Plattformen. Universalrelationen und natürlichsprachliche Schnittstellen der Datenbank-Technik werden mit der für das Web typischen Schlüsselwortsuche kombiniert. Daraus wird ein Meta-Datenmodell für die Suche in Datenbank-basierten Web-Applikationen hergeleitet, wobei die strukturelle von der inhaltlichen Bedeutung der Suchbegriffe getrennt wird. Basierend auf dieser Idee wurden Schnittstellen für Software-Agenten und Roboter von Suchmaschinen implementiert.

Die Bewertung der vorgeschlagenen Methoden erfolgte im Rahmen zweier realer Implementierungen, beide sowohl mit HTML als auch mit Java. Ein quantitativer Vergleich dieser Applikationen zeigt: Für durchschnittlich weniger als 200-300 User-Interaktionen verursacht HTML weniger Netzwerkauslastung als Java. Eine qualitative Analyse der beiden Applikationen brachte ermutigend positive Rückmeldungen der Benutzer der Prototypen. Ähnlich positive Ergebnisse werden auch für die Zukunft erwartet, wenn die vorgestellten Techniken das Design Datenbank-basierter Web-Applikationen schneller, einfacher und weniger fehleranfällig werden lassen.

Preface

> *Wie wird es sein, wenn wir mit der Schnelligkeit des Blitzes Nachrichten über die ganze Erde werden verbreiten können, wenn wir selber mit großer Geschwindigkeit und in kurzer Zeit an die verschiedensten Stellen der Erde werden gelangen und wenn wir mit gleicher Schnelligkeit große Lasten werden befördern können? Werden die Güter der Erde da nicht durch die Möglichkeit des leichten Austausches gemeinsam werden, daß allen alles zugänglich ist? Jetzt kann sich eine kleine Landstadt und ihre Umgebung mit dem, was sie hat, was sie ist und was sie weiß, absperren: bald wird es aber nicht mehr so sein, sie wird in den allgemeinen Verkehr gerissen werden. Dann wird, um der Allberührung genügen zu können, das, was der Geringste wissen und können muß, um vieles größer sein als jetzt. Die Staaten, die durch Entwicklung des Verstandes und durch Bildung sich dieses Wissen zuerst erwerben, werden an Reichtum, an Macht und Glanz vorausschreiten und die anderen sogar in Frage stellen können. Welche Umgestaltungen wird aber erst auch der Geist in seinem ganzen Wesen erlangen?*
>
> Adalbert Stifter, 'Der Nachsommer', 1857 [Wal81].

Although the novel 'Der Nachsommer' was written in 1857, the above quotation from the Austrian poet Adalbert Stifter is of immediate interest. It shows the dangers but also the new opportunities of globalization caused by the world-wide distributed information networks [Zöl97]. While some people fear the demand for higher qualification, the final question of Stifter gives us an idea of the new potentials of globalization: Even smaller countries and companies can stride ahead of larger countries or companies – provided they can retrieve the necessary information and acquire the needed knowledge at first. This was one main motivation for dealing with information technology in my thesis. Besides, playing with Web information systems and Web technology in general can be a lot of fun, too.

Acknowledgements

First of all I would like to thank my thesis advisor Professor Richard Eier for his support and for always providing the right mixture of flexibility and encouragement. I will always remember his "How's your thesis going?". I am also grateful to Professor Mehdi Jazayeri, my second thesis advisor, for his assistance despite the very tight time schedule during the final phase. I owe further thanks to Professor Heinz Zemanek for his interesting historic insights and valuable discussions about the holistic view of things.

Preface

Many of my students have contributed to my thesis through prototype implementations during practical work and diploma theses. I would like to thank them all, especially Jürgen Falb, Wolfgang Radinger, Christian Halter, and Wolfgang Kampichler for their diligence and also for their patience with my constant changes to the design. Thanks also go to my colleagues Bernd Petrovitsch for his system administrative support, and Konrad Kratochwil for his hints and suggestions for submitting papers for publication. Further thanks go to my colleagues Thilo Sauter, Martin Manninger and Michael Kunes for countless valuable discussions.

Most of all I would like to thank my family: My parents for 'always knowing' that I would get there some day and Caroline for her constant support and understanding.

Finally I would like to thank the whole Internet community – all the people who make their work available to anybody who wants it. This great idea of sharing knowledge helps to promote new ideas quickly and helped me to improve my work. Many of them I do not even know – but thanks to all of them anyway.

Contents

Contents

Abbreviations

AI	Artificial Intelligence
ANSI	American National Standards Institute
API	Application Programming Interface
ARPA	Advanced Research Projects Agency
ASCII	American Standard Code for Information Interchange
CAD	Computer Aided Design
CBUI	Character Based User Interface
CGI	Common Gateway Interface
CIM	Computer Integrated Manufacturing
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
DB	DataBase
DBMS	DataBase Management System
DCOM	Distributed Component Object Model
DHTML	Dynamic HTML
DNA	Distributed interNet Applications
DOM	Document Object Model
DSSSL	Document Style Semantics and Specification Language
ECMA	European association for standardizing information and communication systems (formerly European Computer Manufacturers Association)
ER	Entity Relationship
FSM	Finite State Machine
FTP	File Transfer Protocol
GIF	Graphic Interchange Format
GIS	Geographic Information System
GUI	Graphical User Interface
HTML	HyperText Markup Language

Abbreviations

HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IIOP	Internet Inter ORB Protocol
IP	Internet Protocol
JDBC	only a trademark – but often thought of as standing for Java DataBase Connectivity
JDK	Java Development Kit
JFC	Java Foundation Classes
JPEG	Joint Photographic Experts Group
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KBMS	Knowledge Base Management System
KQML	Knowledge Query and Manipulation Language
LAN	Local Area Network
MAC	Medium Access Control
MB	MegaByte
MIME	Multipurpose Internet Mail Extensions
NC	Network Computer
NLI	Natural Language Interface
NNTP	Network News Transfer Protocol
ODL	Open and Distance Learning
ODBC	Open DataBase Connectivity
ODMG	Object Data Management Group
OMG	Object Management Group
OODBMS	Object Oriented DataBase Management System
OQL	Object Query Language
ORB	Object Request Broker
OS	Operating System
PC	Personal Computer
PDF	Portable Document Format
PGP	Pretty Good Privacy
PHC	Passive HTML Controls
PHC/DL	PHC/Design Language
PHC/LL	PHC/Layout Language

Abbreviations

PHCI	PHC Interface
PL/SQL	Procedural Language with embedded SQL
PNG	Portable Network Graphic
POP	Post Office Protocol
PTN	PeTri Net
QBE	Query By Example
RDF	Resource Description Framework
RFC	Request For Comments
RMI	Remote Method Invocation
RSA	Algorithm by R. Rivest, A. Shamir and L. Adleman
SGML	Standard Generalized Markup Language
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
UI	User Interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VRML	Virtual Reality Modeling Language
WAIS	Wide Area Information Server
WAN	Wide Area Network
WIS	Web–based Information System
WWW	World Wide Web
XLL	eXtensible Linking Language
XML	eXtensible Markup Language
XSL	eXtensible Style Language

Chapter 1

Information Systems

During the 20th Century, the key technology has been information gathering, processing, and distribution.
Andrew S. Tanenbaum [Tan96]

This statement concisely motivates the integration of databases and the World Wide Web, which is the aim of this thesis: Databases are designed to process and store information and the Web has turned out to be the place for gathering and distributing of information. Some scientists even consider the Web itself to be a huge world-wide distributed database [AM98]. Although these two seem to be made for each other, a short historic overview explains why they are in fact so different and why this makes integrating them so difficult.

1.1 Historic Overview

Since the commercial usage of computers started in about 1950 [SW74], databases and the Web have had completely different histories: The historic overview of databases is cited from [Ull88]:

> "The earliest true DBMS's (Database Management Systems) appeared in the 1960s, and they were based on either the network or the hierarchical data models. [...] The 1970s saw the advent of relational systems [Cod70]. [...] A decade of development was needed, with much of the research devoted to the techniques of query optimization needed to execute the declarative languages that are an essential part of the relational idea. [...] We see the 1980s as the decade of object-oriented DBMS's (OODBMS) in the true sense of the term; i.e., they support both object identity and abstract data types. These are the first systems to provide well-integrated data manipulation and host languages. However, in one sense, they represent a retrograde step: they are not declarative, the way relational systems are."

Jeffrey D. Ullman further predicted that in the 1990s true Knowledge Base Management Systems (KBMS's) would supplant the OODBMS's. However, this has not taken place up until now. KBMS's are mainly used for expert systems, while even OODBMS's are not yet widely used. Furthermore it has turned out that there is currently no system which can replace the others and each of them has its own application area [HeuS97]:

Relational Databases: These are still the best choice if huge amounts of simply structured data have to be stored and processed with medium complex short transactions. Difficulties arise from the integration of object oriented applications and relational databases, hence object relational successors to the relational database products have currently become popular.

Object Oriented Databases: These offer the easy integration of object oriented programming languages and are well suited for medium amounts of highly complex structured data and long-lasting complex transactions.

Knowledge Base Systems: These are used in expert systems together with deductive programming languages, e. g. Prolog.

Engineering Databases: Successful in the areas of CAD (Computer Aided Design) and CIM (Computer Integrated Manufacturing) with very heterogenous and highly complex structured data but usually just a few entities of one type. The number of types in such a system is almost always as large as the number of entities.

Workflow Management Systems: Support group collaboration and communication. Need active database concepts and high sophisticated transaction control: the transaction model is enhanced to allow cooperation instead of the previously required isolation of transactions.

Spatial Databases: For example for Geographic Information Systems (GIS). Geometric structures and a geometric search are supported. Special data structures support geometric queries.

Document Databases: Office information systems allow collaborative work on documents. Special features are a full-text search and check-in/check-out procedures for documents.

Multimedia Databases: Single entities are very large and unstructured, functions for the manipulation of multi media data are required. Real time requirements become important with video servers.

Temporal Databases: Enhance the relational model to a temporal relational algebra. Temporal transactions, temporal queries and different aspects of time are some of the features of this type of databases. They are typically required to store historic data in repository systems.

Ideally we would like to access *all* of them with one single easy-to-use tool to retrieve the desired information. Could the Web browser be this interface?

The history of the Web has its early roots in the birth of the Internet and can also be roughly measured in decades: In the 1960s scientists started to connect some standalone computers over telephone lines. These early experiments were sponsored by the U.S. Department of Defense under the management of the Advanced Research Projects Agency (ARPA) with the aim of finding a flexible network which would survive the 'big bang' of an atomic war. In the 1970s different networks were connected with standardized protocols (TCP[1]/IP[2]) resulting in the birth of the *Inter*net. In the 1980s TCP/IP was integrated into the Berkeley UNIX operating system. Many universities, research institutes of large companies and government authorities joined the Internet. Information was gathered using FTP (File Transfer Protocol) and Telnet which proved to be a user unfriendly way of searching information. However, since the majority of people on the net were computer literate, this did not prove to be too great a problem.

Things became quite different with the Web: Tim Berners-Lee proposed the first version of HTML (Hypertext Markup Language) in 1989[3], the first server and browser prototypes came into being between 1990 and 1992 [Con98]. From then on, the Web caused an exponential growth of the Internet [Rut98, ⇒RIPE, ⇒NW][4]: In June 1991 about 500.000 nodes were recorded world-wide, 63.000 of them in Europe and 1728 of them in Austria. As of June 1998 there were about 35 millions nodes recorded world-wide, 6.65 millions nodes in Europe and 133.000 of them in Austria. The monthly growth rate is currently about 1 million world-wide, over 200.000 for Europe and over 2500 for Austria. Hence today's monthly growth rate far exceeds the total number of nodes only 7 years ago.

The main reasons for the exponential growth rate the Web causes were the easy to use point-and-click graphical user interface and the integration of all relevant services (Ftp, gopher) into one tool – the Web browser. As more and more Web servers appeared all over the world, the number of users accessing them also grew. At this point in time electronic commerce started to take the Web by storm and companies today are making and saving money by going online - up to two million U.S. dollars per day. Internet commerce in goods and services between companies was estimated at US$ 8 billion in 1997, according to Forrester Research in Boston. The International Data Corporation estimates that business-to-business sales over the Internet will represent US$ 81.2 billion by 2000 [Cla97].

While all this was happening, surfers were very much still taking the back seat in a sophisticated slide show. The user was being driven through a site, as opposed to driving the session. This is why information retrieval and user-friendly interactivity have been key issues in Web design [Cat97, MP97] since the beginnings of the Web. Using some scripts (e. g. Perl) behind the CGI (Common Gateway Interface), the first results were rather poor. It soon turned out that most real–life applications need an underlying database on the server side to deal with large amounts of data and a design methodology to implement complex functionality. Moreover, there are already many databases and legacy systems

[1] Transmission Control Protocol

[2] Internet Protocol

[3] Although the term hypertext was initially coined in the 1960's, its popularity first grew when the Web was invented.

[4] Citations containing an arrow "⇒" mark a reference to the Web.

in existence which people want to connect to the Web with similar functionality as in classic implementations [Adi97c]:

> "Good Web sites are database backed Web sites. [...] To have a site that is stable, flexible and scalable, you really need the database advantage."

Unfortunately almost 35 years of separated development have provided quite different technologies. The next sections explain why relational databases and the Web are two different worlds, before the following chapters present solutions to this problem. However, no comprehensive overview of all Web technologies is provided - it would be outdated within three months. Instead the mainstreams are categorized with respect to their usability for database backed Web applications.

1.2 User Interface: HTML, JavaScript and Java

If the standard user interface of the applications has to work with nearly any browser and platform and should not depend on proprietary non-standard extensions like Plug–Ins or Active–X, then the user interface is restricted to the possibilities of pure HTML. This concerns the representation, but the user interaction even more so – the only functional elements are following a link, pressing a submit button of a form or clicking on an image or imagemap. They always result in an HTTP (Hypertext Transfer Protocol) connection. Hence, any functionality resides on the server side.

Of all the different approaches to make the browser more powerful and interactive, JavaScript [Fla97, ⇒JS] is a way of doing almost anything within a browser. JavaScript is emerging as the client-side scripting language of the Web, at least since version 1.2, when a serious security model was implemented. Currently JavaScript is moving towards the formal acceptance of it as an international standard. The European standards body [⇒ECMA] has approved ECMA–262, the language specification derived from Netscape JavaScript and submitted it to ISO/IEC JTC 1 for adoption under the fast–track procedure.

However, *the* programming language of the Web is Java from Sun [⇒Java]. It can be used on both the client (applets) and the server side (servlets). It provides very promising concepts for the design and implementation of object oriented distributed applications of nearly unlimited complexity and with few restrictions on the user interface. Despite its indisputable benefits, there are some drawbacks: to implement servlets efficiently, the virtual machine must be integrated with the database kernel. Only a few database companies have announced this so far. Even more importantly, an applet must be downloaded to the client first. This is no problem for intranets. However the internet has a smaller bandwidth, especially in Europe. For typical Internet applications consisting of just a few client–server interactions, the long download–time at the beginning is often not worthwhile.

On the other hand, more 'intelligence' on the client side can take some load off the server and the network, thus making Java applets more efficient for long lasting transactions.

Moreover, there is additional functionality, especially for the user interface. This is why this approach makes use of Java as an alternative implementation technique, but still keeps the pure HTML interface available. A quantitative analysis will help to compare the Java and pure HTML approach with each other based on the use of typical end user profiles observed during the pilot phase of the real–life applications. Nevertheless, what *can* be done with pure HTML without incurring a disadvantage, *will* be done with pure HTML.

1.3 State Maintenance: HTTP and JDBC

Another problem arises due to the different nature of databases and the Web: In a standard client-server application, both sides have their session state (e. g. the current state of a database transaction on the server side and the current state of the user interface on the client side) with a connection oriented protocol connecting them. The Web, on the other hand, was never designed for client-server applications but rather for a quick and simple delivery of linked hypertext documents. Hence the HTTP (Hypertext Transfer Protocol) is stateless [Iye97] and the browser does not store its user interface state[5] but connection orientation is inevitable for real-life database transactions.

To overcome this deficiency, two approaches are possible:

Long URL encoding: The complete session state information is passed back and forth between client and server. This can be achieved by encoding this information into each single URL (Uniform Resource Locator) of all the anchors within the page and into an `INPUT` field of the type `hidden` into every form respectively. The advantage is: The server does not have to keep the session state information in its memory because it gets it back from the client with each new HTTP request. This technique is feasible for simple and short state information especially if no database is available on the server side.

Short URL encoding: Only a short session identifier, called a handle, is passed back and forth between client and server with every HTTP connection using the same techniques as above. In this case the complete session state information (database transaction state and user interface state) of *all* open sessions is kept on the server side, usually in a database. This causes more load on the server but takes some load off the network, especially if we use pages with lots of links.

The second approach scales better to complex state information and it even has one further advantage: If we use frames, the first method would not work, because if one frame is reloaded and the session state has changed during this HTTP connection, then the session state information encoded in all the other pages of a frameset is outdated.

[5] In fact, it does, otherwise the back button would not work. However, without JavaScript it is not possible to make use of this information.

Alternatively Netscape's Cookies ([KM97] and RFC 2109 [⇒RFC], Requests For Comments) can be used both as long and short Cookies to implement the same two possibilities of state maintenance. In this approach, short URL encoding is used for state maintenance. This technique allows even more than one concurrent session stemming from the same client.

JDBC [⇒Java] – which is a trademark, but often thought of as standing for 'Java Database Connectivity' – is a Java API (Application Programming Interface) for executing SQL (Structured Query Language) statements. It provides a connection-oriented alternative for database connectivity with Java applets. Hence with JDBC all the possibilities of standard client-server applications become thereby available: The applet contains its user interface state as object states of the GUI (Graphical User Interface) objects. As mentioned above, both approaches will be compared.

1.4 Database Connection: CGI, FastCGI and API

In general, a pure CGI compliant interface between Web server and database server is sufficient. For a sequence of HTTP connections, however, CGI is rather ineffective. This is due to the overhead of spawning a new process each time which, in turn, opens and closes the connection to the database. This performance loss is normally overcome by two approaches [Adi97a]. A proprietary approach *not* used here links server programs directly with the Web server [NSAPI [⇒NSAPI], ISAPI [⇒ISAPI]]. Another approach involves preforking multiple processes, which communicate with the Web server and which stay connected to the database. This is done with FastCGI [Adi97, ⇒FCGI] by Open Market and the Web Request Broker (WRB) of the Web Application Server by Oracle [Gre97a, ⇒Oracle]. Both approaches improve server performance, but only FastCGI is a non–proprietary standard with easy migration from CGI. The Oracle WRB is at least CGI compatible.

1.5 Wide Area Distribution

The decentralized structure of many systems causes additional problems. New information nodes should be able to join the system but user queries involving more than one node should also be possible. Everything is thus put into the database, even informational pages without any control element. No single static HTML page therefore exists; each page is generated from the database dynamically. This allows usage of the various replication concepts for distributed databases already available.

Another problem of wide area distribution is the latency of the net itself. This restriction tells us to use small images, small pages and as few HTTP connections as possible. The simple solution is to use structures which are not too deep, in order that the user only has to follow a few links to receive the desired information. An optional frame structure is also provided to group the information into logical units where less information has to be transferred with each HTTP connection.

The next chapter introduces a new approach for the design and implementation of Web applications based on relational databases with only pure HTML on the client side. Chapter 3 evaluates alternative implementations with Java using object oriented techniques. Chapter 4 investigates possibilities for searching in database backed Web applications – from search engines to Software Agents. Chapter 5 finally presents real-life examples used for the evaluation of the proposed design and implementation methods. After a quantitative analysis the thesis concludes with references to related work, a short summary and proposed future work.

Chapter 2

Relational Databases and pure HTML

This chapter introduces a new approach of how to design and implement Web applications based on relational databases with only pure HTML on the client side and explains the *key ideas*. The aim is to create a *design language* to describe the user interface layout, the functionality and the database transactions in a *homogenous way* following a virtual client-server model. The actual *implementation* of such a system is then *composed* of several parts with potentially differing techniques. For example, either a pure HTML interface with full functionality on the server side or a Java applet with greater functionality on the client side. A toolset has been implemented to *generate* the different applications automatically from the design language, thus implementing complex user interactions on database information nodes distributed over the Web.

The first section introduces the theoretical state machine model used for design and implementation. Based on this model, a design language is presented and it is shown how an application is generated from this description using a set of tools and how this application can be integrated into a complete information system. A short summary evaluates the approach, some limits are revealed and further improvements are suggested.

2.1 State Machine Model

Being limited to an HTML-only user interface, we are confronted with a disadvantage with HTML forms: no functional dependencies between different input fields of a form can be implemented. Therefore input into one field cannot cause a restriction on the possible inputs for another input field. An HTTP session has to be performed before functional dependencies or restrictions can affect other inputs. To achieve this HTTP session, a submit button has to be explicitly pressed. It is not usually enough to fill in an input field or make a selection.

The key idea of this approach therefore is to use links to solicit user input. Control elements are thus constructed of sets of links grouped together and called *Passive HTML*

Controls (PHC). The links of a PHC are called elements. Each element has two parts: The anchor tag defining the link of the element and the content of the tag defining the output string of the element. This approach is reasonable for a typical Web application where most user interactions consist of browsing, selecting and collecting information from the database. The possible values for user input are predefined in these cases. Form input is only used where user input has to be gathered (e. g. name and address for an online order).

The linked structures and functionality of hypertext documents are usually modelled with Finite State Machines (FSM), Petri Nets (PTN) or colored Petri Nets. This approach uses FSMs instead of PTNs because the advantages of petri nets[1] do not justify the greater complexity in this specific case. However, the general idea of modelling hypertext documents as automata is well proven, one recent work is [SFC98][2]:

> "What is required is a particular view of thinking about a document, i. e. one must view a document as an abstract automaton that specifies the process of browsing within it. Such a view is easily obtained for the hypertext systems in use today. In fact [...] the linked structure of a document can usually be thought of as the state transition diagram of a FSM."

Hence each document defines a state (i. e. the state that this document is displayed in with only one document being displayed at a time) and the state transitions are provided by the hyperlinks. Clearly this describes a FSM with a finite set of pages (states) and a finite set of links per page. Before this idea is adopted to database backed Web applications, here is a short recall of some classic definitions of FSMs [HU96].

Definition 1 (Finite State Machine) *A non-deterministic Finite State Machine is a quintuple* $F = (\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{R}, \mathcal{F})$*:*

- $\mathcal{I}$ *is a finite set of input symbols.*
- $\mathcal{O}$ *is a finite set of output symbols.*
- $\mathcal{S}$ *is a finite set of states.*
- $\mathcal{R} \subseteq \mathcal{S}$ *is the set of initial states.*
- $\mathcal{F} \subseteq \mathcal{I} \times \mathcal{S} \times \mathcal{S} \times \mathcal{O}$ *is the transition relation.*

Definition 2 (completely specified) F *is completely specified, if for all* $I \in \mathcal{I}, S \in \mathcal{S}$ *there exists at least one* $S' \in \mathcal{S}, O \in \mathcal{O}$ *such that* $(I, S, S', O) \in \mathcal{F}$*, i. e. if the FSM has at least one choice of next state/output for each input/present state combination.*

[1] The Petri Net remains manageable in size compared to the growth rates of state machines for the representation of parallel and composite systems [JH98].

[2] With many references to other work about using formal automata to define and verify hypertext systems.

Definition 3 (deterministic) *F is deterministic if $\mathcal{R}$ is a singleton and $\mathcal{F}$ is a function $\mathcal{F} : (\mathcal{I} \times \mathcal{S}) \mapsto (\mathcal{S} \times \mathcal{O})$, i. e. if the FSM has at most one choice of next state/output for each input/present state combination.*

In this approach deterministic FSMs are used, but they are not always completely specified. The most important difference between a database backed Web application and a static document tree is the number of pages: In the latter case this number is well defined, whereas in the first case the number of dynamically generated pages is infinite due to the arbitrary nature of the Turing machine which is formed by the database application. Even so, in order to exploit this model a large number – which might even be infinite – of states can be grouped together for purposes of easier analysis, design and implementation[3]:

Each PHC has its own state machine. When the user clicks on the link of an element, a state transition can occur causing different output to be dynamically generated. The complete state of a PHC consists of all the values required for the presentation of the HTML page. These states can be grouped together to form a small number of explicitly defined states. Each state now defines a class of dynamically generated pages with a very similar appearance. Hence the complete state of a PHC consists of one state value and a set of parameters $\mathcal{P}$ which define the slightly different output for all the pages from the same class (i. e. state). The parameters are mostly filled with values from the static database $\mathcal{DB}$ during state transitions. This separation of the database-dependent appearance of the page from the state value makes the state machine independent of the arbitrary contents of the database. Definition 4 defines a PHC as a deterministic FSM:

Definition 4 (PHC State Machine) *A PHC State Machine is a quintuple $F = (\mathcal{I}, \mathcal{H}, \mathcal{S}, R, \mathcal{F})$:*

- *$\mathcal{I}$ is a finite set of links.*
- *$\mathcal{H}$ is a finite set of output functions $H(\mathcal{P}, \mathcal{DB})$ defining the HTML output in each state.*
- *$\mathcal{S}$ is a finite set of states.*
- *$R \in \mathcal{S}$ is the initial state.*
- *$\mathcal{F} : (\mathcal{I} \times \mathcal{S}) \mapsto (\mathcal{S} \times \mathcal{H})$ is the transition function.*

Figure 2.1 shows the complete transition graph of an example PHC implementing a hierarchical selection, figure 2.2 shows the respective browser view during state S1. The PHC consists of six elements *Overview*, *OneUp*, *Province*, *Region*, *District* and *SelList*, the edges of the arrows are labeled with the name of the clicked element: Provinces consist of regions which in turn consist of districts. `Overview` always leads to selection at the highest level (province), while `extend region level` enhances the actual selection by one level.

[3] A method well known, for example, with protocol machines [Tan96].

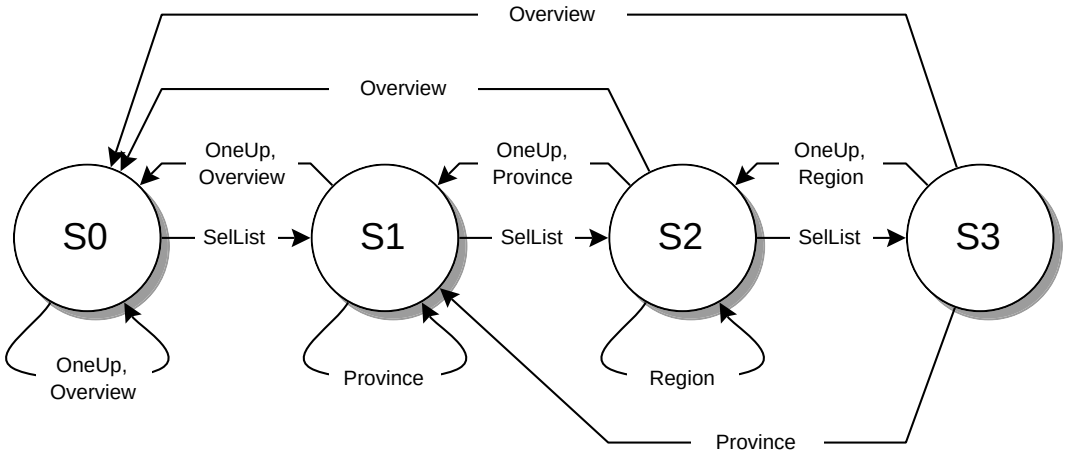


Figure 2.1: State machine model.

Region selection

overview
extend region level
Niederösterreich

Donauregion
Industrieviertel
Mostviertel
Waldviertel
Weinviertel

Figure 2.2: Browser view of the PHC.

Above the horizontal rule, the actual selection is shown: the province `Niederösterreich`. The `SelList` element finally provides a list of the regions of `Niederösterreich` from which to select. The selection of `Mostviertel` for instance causes a transition to state S2 setting the parameter `Region` to a value which identifies the region `Mostviertel` in the database.

The HTML output function $H \in \mathcal{H}$ of a PHC is *only* determined by the PHC's actual state $S \in \mathcal{S}$, the contents of the parameter set $\mathcal{P}$ and the underlying database $\mathcal{DB}$. It is *not* influenced by the last input ('click'):

$$\begin{aligned} S &\Rightarrow H(\mathcal{P}, \mathcal{DB}) \\ (S, I) &\Rightarrow S'. \end{aligned} \tag{2.1}$$

Hence the output H of the state machine is not determined by $\mathcal{S} \times \mathcal{I}$ but only by $\mathcal{S}$,

splitting the function $\mathcal{F}$ from definition 4 into two functions $\mathcal{F}_I : (\mathcal{I} \times \mathcal{S}) \mapsto \mathcal{S}$ and $\mathcal{F}_O : \mathcal{S} \mapsto \mathcal{H}$. This strict separation between state transition and output generation is a key feature of this model to achieve stable and robust Web applications: It is necessary to guarantee a deterministic generation of the dynamic HTML pages independently from the last user interaction. Hence, as long as the affiliated state information is not changed, a dynamically generated Web page will always look the same, regardless of which user interactions have taken place in the meantime. Web applications which ignore this rule can easily produce unpredictable results and a corrupted output. Again, the two reasons for this are the stateless nature of the HTTP protocol and the inability of the browser to store user interface state information. Hence, if a user interaction has to produce a different output, it *first* has to change the session state information, making it possible to generate the new output from the changed session state information afterwards.

One special property of the PCH state machine is the relation between input and output provided by the two parts of the elements: A click on the link of an element is the input and each output function $H(\mathcal{P}, \mathcal{DB})$ is the composite of a set of functions $O_E(\mathcal{P}, \mathcal{DB}) \quad \forall E \in \mathcal{E}$, one for each element, because the HTML output of a PHC is apparently the composition of the output of the elements of the PHC.

With this different point of view a click on one of the output elements E from the element set $\mathcal{E}$ can be seen as input to the state machine and can set a new state. The output O of each element $E \in \mathcal{E}$ is still determined by the PHC's actual state $S \in \mathcal{S}$, the contents of the parameter set $\mathcal{P}$ and the underlying database $\mathcal{DB}$,

$$\begin{aligned} S \Rightarrow O_E(\mathcal{P}, \mathcal{DB}) \quad \forall E \in \mathcal{E} \\ (S, E) \Rightarrow S'. \end{aligned} \tag{2.2}$$

This modifies the output function $\mathcal{F}_O$ to $(\mathcal{S} \times \mathcal{E}) \mapsto \mathcal{O}$ while the input function has become $\mathcal{F}_I : (\mathcal{S} \times \mathcal{E}) \mapsto \mathcal{S}$ replacing $\mathcal{I}$ by $\mathcal{E}$ as the new input set of the state machine. Both functions now have equal left sides and can hence be combined again. The following is the final definition of the 'delayed output' state machine:

Definition 5 (Delayed Output State Machine) *A Delayed Output State Machine is a quintuple $F = (\mathcal{E}, \mathcal{O}, \mathcal{S}, R, \mathcal{F})$:*

- *$\mathcal{E}$ is a finite set elements, a click on an element is an input to the state machine.*
- *$\mathcal{O}$ is a finite set of output functions $O(\mathcal{P}, \mathcal{DB})$ defining the HTML output for a particular element in a particular state.*
- *$\mathcal{S}$ is a finite set of states.*
- *$R \in \mathcal{S}$ is the initial state.*
- *$\mathcal{F} : (\mathcal{E} \times \mathcal{S}) \mapsto (\mathcal{S} \times \mathcal{O})$ is the transition function.*

The delayed output state machine from definition 5 looks quite similar to the PHC state machine from definition 4 but the transition function $\mathcal{F}$ has a quite different semantic meaning: For each state S of the PHC, the transition function $\mathcal{F}$ defines the output and the link for each element $E \in \mathcal{E}$ in that particular state S. The output is called *delayed*, because for each state transition E/O from one state S to another state S', O defines the output of E for the *old* state S and *not* for the new state S'. In other words, the element E is presented with O in state S and a click on E causes a state transition to S' (designer's point of view). This model is also the basis for the design language described in section 2.3 on page 19. This is later used in the opposite way for *look-ahead link generation* (see section 2.6 on page 40): To generate the HTML output for a PHC in a particular state S, all state transitions E/O leaving state S have to be combined: Each E/O defines output O and link destination S' of one element E in state S. This allows it to encode all *possible* state transitions from that page into the links of the elements in the generated HTML output in advance (implementation's point of view).

Finally, the parameter settings are introduced into the model: A click on an element $E \in \mathcal{E}$ in state S can set a new state S' and eventually set some parameters $P_k \in \mathcal{P}, k \in \mathcal{K}(E)$ to their new values $p_k(E)$:

$$\begin{aligned} S &\Rightarrow O_E(\mathcal{P}, \mathcal{DB}) \quad \forall E \in \mathcal{E} \\ (S, E) &\Rightarrow S', P_k := p_k \quad \forall k \in \mathcal{K}(E). \end{aligned} \tag{2.3}$$

Since HTML is used to implement the state transitions, every possible state transition has to be translated into the hyperreferences of the output elements. Thus, the actual state S, the parameter set $\mathcal{P}$ and the database $\mathcal{DB}$ determine the element set $\mathcal{E}$, where each $E \in \mathcal{E}$ consists of an output function $O_E(\mathcal{P}, \mathcal{DB})$, a potential new state $S'(E)$ and eventually some parameters $P_k \in \mathcal{P}, k \in \mathcal{K}(E)$ and their new values $p_k(E)$:

$$(S, \mathcal{P}, \mathcal{DB}) \Rightarrow O_E(\mathcal{P}, \mathcal{DB}), S'(E), \{P_k, p_k(E) | k \in \mathcal{K}(E)\} \quad \forall E \in \mathcal{E}. \tag{2.4}$$

While notation (2.3) is the basis for the theoretical client-server model and the design language, the equivalent notation (2.4) is the basis for the actual implementation of this model with dynamically generated HTML anchor tags, where $S'(E)$, P_k and $p_k(E)$ are encoded in the hyperreference URL and where $O_E(\mathcal{P}, \mathcal{DB})$ is the content of the tag (2.5). Notation (2.6) shows the final HTML statement for the element `Mostviertel` in the browser view figure 2.2.

$$\begin{aligned} &< \texttt{a href} = '' \texttt{phc?handle} = h \& \texttt{state} = S'(E) \\ &\qquad \& P_k = p_k(E) '' > O_E(\mathcal{P}, \mathcal{DB}) < /\texttt{a} > \end{aligned} \tag{2.5}$$

$$\begin{aligned} &< \texttt{a href} = '' \texttt{S\_Geo?handle} = 17 \& \texttt{state} = 2 \\ &\qquad \& \texttt{pRegion} = 17 '' > \texttt{Mostviertel} < /\texttt{a} > \end{aligned} \tag{2.6}$$

To demonstrate this, let us trace an example of a possible user input as in figure 2.3. The triangle marks the click while the circles at the bottom show the state transitions with respect to the state machine in Figure 2.1. The edges of the arrows contain the name of the clicked element and possibly the name and value of a parameter which is set during

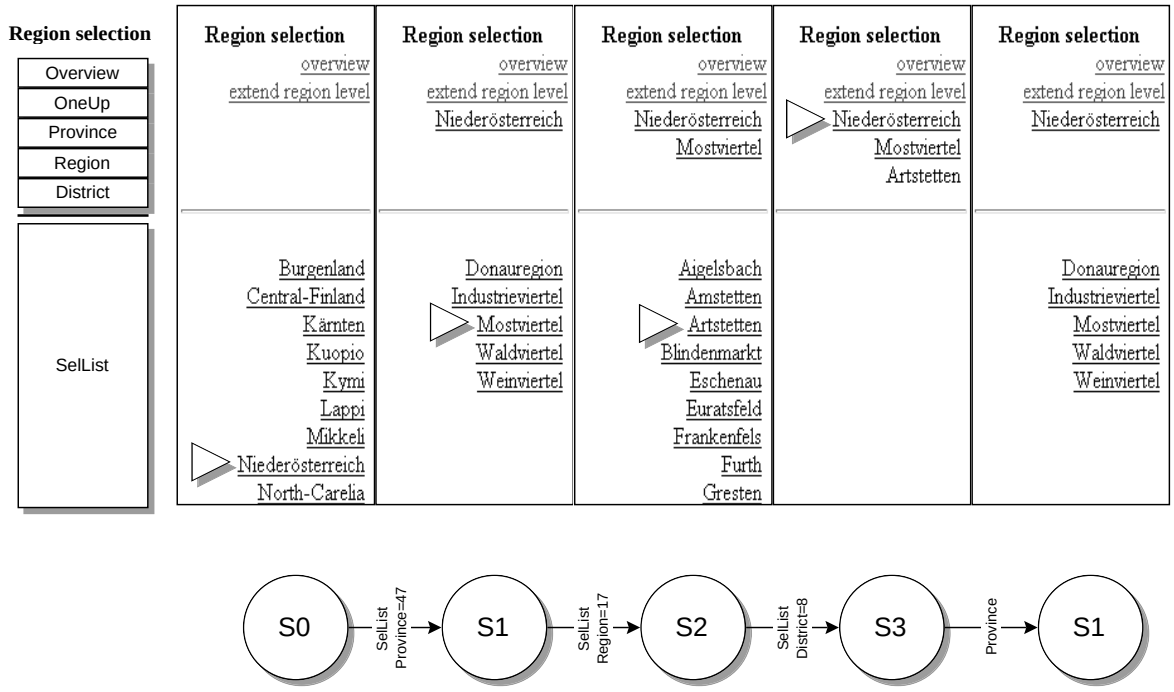


Figure 2.3: Geographic Selection.

the state transition. At the beginning the province `Niederösterreich` is selected. The next state shows `Niederösterreich` in the element `Province`, while the element `SelList` now shows the regions of `Niederösterreich`. During the next two state transitions, a region and a district are selected and a click on `Niederösterreich` in state 3 brings up the regions of `Niederösterreich` again. Figure 2.1 shows the complete transition graph of this PHC. The edges of the arrows are labeled with the names of the elements which fire the transition. The value to which each parameter is set during the state transition is also marked.

2.2 Abstraction: Passive HTML Controls

The last section defined a PHC as a set of links grouped together to fulfill a common control task and showed the state machine functionality of a PHC from the user's point of view. Figure 2.4 shows an object based approach to the complete functionality of a PHC from the application designer's point of view:

- A PHC has an internal data structure containing the session state information including the session parameter set. This state is private but parts of it can be made public to other PHCs.
- A PHC consists of a set of interactive user interface elements. In this model, the only way a user can interact with a PHC is by 'clicking' on an element[4]. Each

[4]This reflects the situation on the Web where the only possibility for a user interaction is by clicking on a link, if forms are not used.

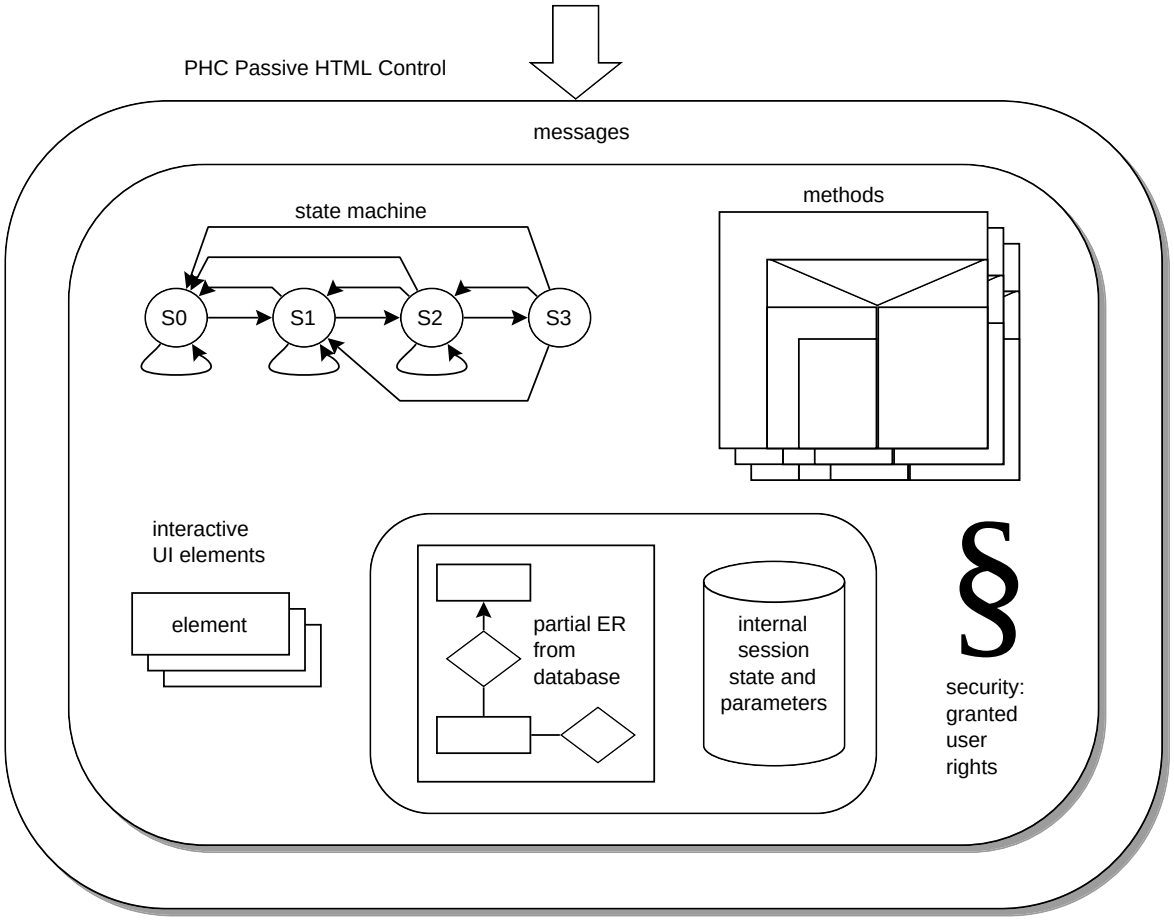


Figure 2.4: Object based approach to Passive HTML Controls.

element contains the following rules:

  - A rule for a string and/or an image representation within the user interface.
  - A rule indicating what to do if the user selects this element (e. g. by clicking on it).

  These rules usually use the database contents and the session state information.

- The behaviour of a PHC is modelled with a delayed output state machine (see definition 5 on page 12).
- A PHC is associated with a particular part of the underlying database. This part can easily be described as a part of the complete ER (Entity Relationship) diagram of the database.
- A PHC may contain additional methods to be called via messages from other PHCs or from the user interface (click on a link). These methods implement more complex functionality and can also cause state transitions.
- To the user a PHC presents itself as a set of links grouped together to implement a particular control task.

A PHC does *not* contain any information about layout or HTML representation. This information is completely separated into layout elements (see section 2.4 on page 25).

Figure 2.5 shows how these PHCs interact with the user interface and with each other to produce dynamic HTML pages. A PHC can be 'called' in two different ways:

1. The functional part (state machine and additional methods) of a PHC is called from the user interface when the user selects a particular element ('clicks' on it) or from another PHC's functional part.

2. The output part (rules for the presentation of the elements and rules for the action caused by the selection of each of these elements) is called from an HTML page containing this PHC. This sort of call is made via the PHCI (PHC interface) from a layout description. Each PHC can have more than one layout (section 2.4).

These two calls reflect the strict separation between state transition and output generation mentioned earlier: A user interaction may result in a changed session state information, but the output of a PHC is only dependent on the new session state information but not on the last user interaction. This is the key feature to achieve stable and robust Web applications: As long as the affiliated state information is not changed, a dynamically generated Web page will always look the same, regardless what user interactions have taken place in the meantime. It is easy for Web applications ignoring this rule to often produce unpredictable results. The two reasons for this are, again, the stateless nature of the HTTP protocol and the inability of the browser to store user interface state information.

To be callable in these two ways, two methods are mandatory for each PHC: A method for state manipulation and a method to generate the output of the PHC. Further methods are optional. In figure 2.5 the user selection of element E of PHC 0 causes func1 of the PHC 1 to be called, eventually a method or a state machine action. This method can now call another method of any PHC or cause a page output. The selected page then calls the output part of the contained PHCs, thus producing the final HTML page. In the special case of the PHC `Geo` in figure 2.6 a click on the element `Mostviertel` of the list `SelList` causes the state transition from S0 to S1 of the same PHC Geo and also sets the parameter `Region` to the identifier belonging to `Mostviertel`. The same page is then regenerated and all the PHCs contained within this page are called with the output-method to produce the appropriate HTML code finally delivered to the client, as shown in figure 2.6. From the user's point of view the click on `Mostviertel` has simply provided him with a list of all the districts located in the region Mostviertel.

So far the inherently parallel nature of the distributed system [CDK94] 'Web' has been ignored: Many almost concurrent requests can make parallel use of one PHC and the underlying database. With modern relational database products parallelity is no problem at all, nor is it a problem with PHCs: From the designer's point of view, each session has its own PHCs with their own session state information. Hence the designer can design the application as if only one user is connected at a time. He does not have to worry about concurrency. As described later, the state management mechanism holds the session state

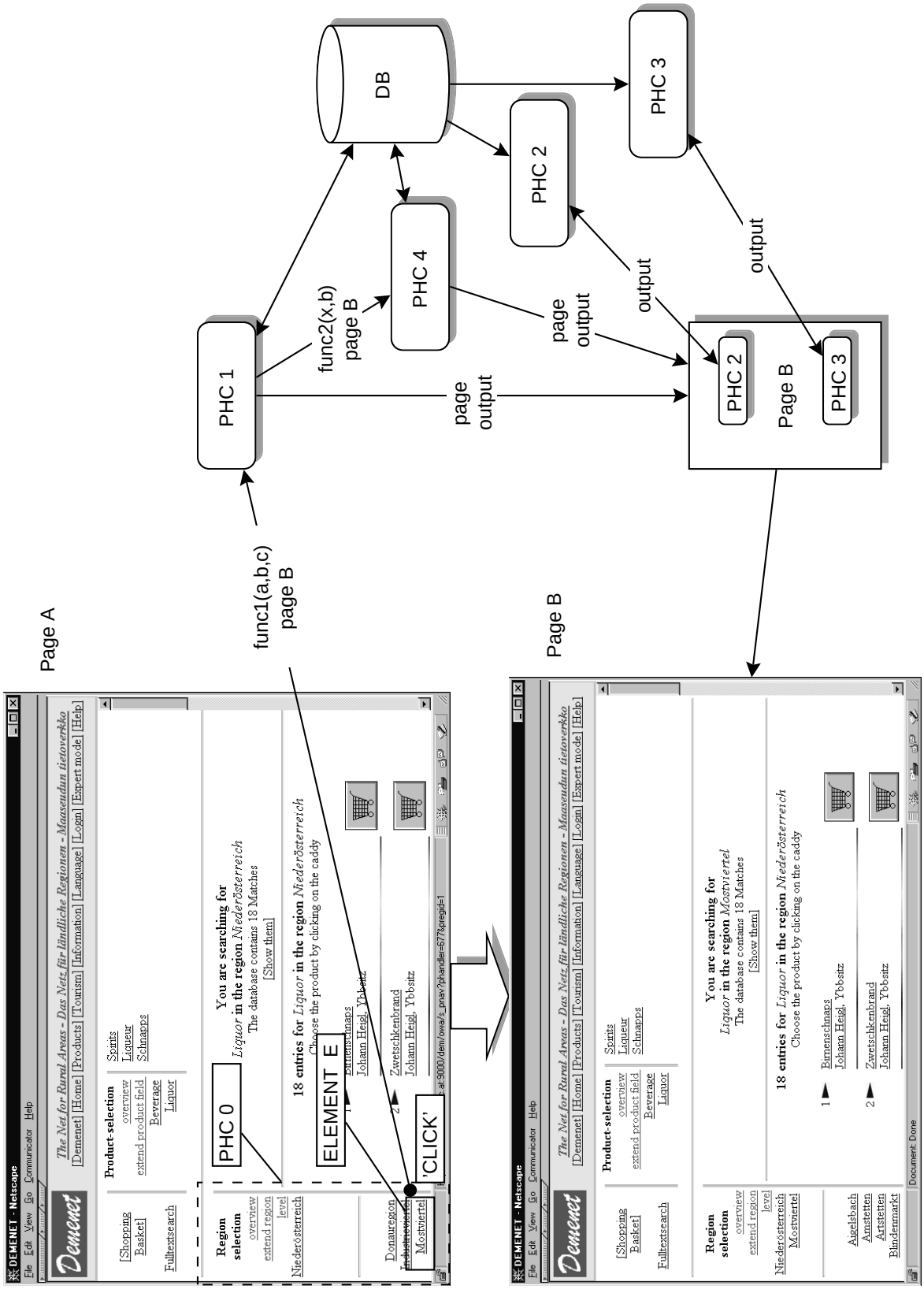


Figure 2.5: How PHCs interact with the user interface and with each other.

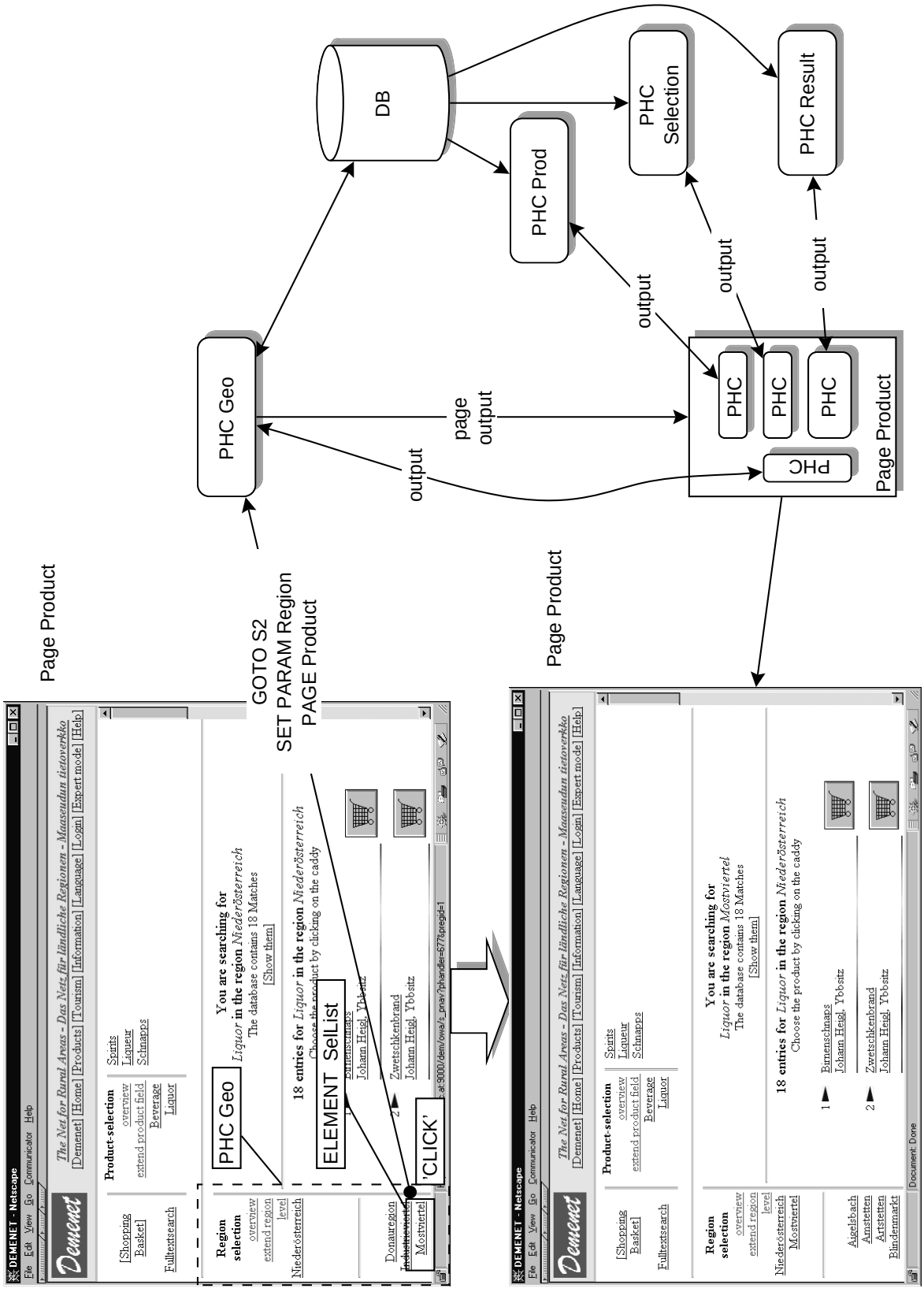


Figure 2.6: How the PHC Geo interacts with the user interface, with itself and with the generated HTML pages.

information for each state in a special data structure within the database, making the database itself responsible, in addition, for the concurrency management of the PHCs.

2.3 Design Language

In classic database design [HeuS97, GR95], following the requirements' analysis, further development steps are strictly separated into information design and functionality design. The functionality design starts with the functionality analysis and ends with the procedural implementation of the application functionality using standard methods of structured-analysis oriented Software Engineering. The information design starts with information analysis, followed by methods of semantic data design such as Entity Relationship [Che76] and after normalization [Ull88] of the relational design ends with an SQL implementation of a relational database. Though highly dependent on each other, these two separate paths were often followed by different teams of developers with different knowledge. Only strong organizational efforts or mature toolsets can bring these two completely different design methods together, resulting in one final product. Despite these inherent difficulties, which object orientation tries to overcome (see next chapter), a large number of large-scale software systems have been successfully implemented in this way and many mature design and implementation tools are available from the major database vendors. Hence in the area of relational databases the PHC approach belongs to the functionality design and it is assumed that a relational database is either already available (legacy systems), or is designed and implemented at the same time with standard database design tools for semantic data design.

A design language – PHC/DL – has been developed to describe PHCs on an abstract level (functionality, database operations, state machine, element representation and action). In addition, a layout language – PHC/LL – has been developed to describe how the PHCs' output is integrated into HTML pages. Using these two languages, the structural part is separated from the layout. An interface (PHCI) is defined between them to make them as independent from each other as possible. Hence the abstract description of a PHC follows a virtual client-server model, whereas with pure HTML the complete implementation is finally located on the server side. The developer of an application can thus focus on the design, while the otherwise error-prone implementation is generated automatically (see section 2.6).

The PHC/DL is a domain specific language designed to describe all aspects of PHCs as shown in figure 2.4 on page 15. The main structure of a PHC consists of three parts as shown in figure 2.7:

Interface The interface definition contains the forward declaration of the PHC's content (mandatory) and the declaration of the session state information parameter set. It also defines the default state of a PHC as the first state in the forward declaration.

Implementation The implementation of the *functional methods* (optional).

```
PHC Geo

INTERFACE

// interface definition of the PHC

IMPLEMENTATION

// optional methods

BEGIN

// state machine model including mandatory methods:
// 1. ouput method of the PHC
// 2. state manipulation method of the PHC

END
```

Figure 2.7: Overall structure of a PHC/DL description.

Body The very core of the PHC: The state machine model and the element definitions implicitly defining the two mandatory methods for *state manipulation* and *output generation*.

Both parts – implementation and body – can contain database transactions.

The complete PHC/DL syntax and semantics fall outside the scope of this thesis, but the key ideas will be illustrated by examples. Details can be found in [Fal98], where the implementation of this language is described in detail. Figure 2.8 shows the PHC/DL description belonging to the example in figures 2.3 and 2.1.

The most important language elements are:

`ELEMENT` Elements are the base of the body of a PHC/DL description and provide the only possibility of user interaction with a PHC. Depending on the current state, they may or may not be 'clickable' and have a well-defined presentation string and/or image. In notation (2.4) these are named $E \in \mathcal{E}$. Every E has a `PRINT` clause and/or an `IMAGE` clause to define the output $O_E(\mathcal{P}, \mathcal{DB})$ of the element and a `GOTO` clause to define the state transition $S'(E), \{P_k, p_k(E) | k \in \mathcal{K}(E)\} \forall E \in \mathcal{E}$, which a click on this element would trigger. The clauses `STATE`, `ELEMENT`, `GOTO` and `PRINT` together define the delayed output state machine of definition 5 on page 12. The special clause `NULL` in combination with `GOTO` means that no link is generated (no state transition is possible from this element). In combination with `PRINT`, it defines an element as invisible (and thus not clickable) in the corresponding state. All the `PRINT` (and `IMAGE`) clauses of all the elements of a PHC together implicitly form the output method of the PHC. All the `GOTO` clauses of all the elements of a PHC combined form the state machine and the state manipulation method of this PHC implicitly.

`PARAM` A parameter is a data structure belonging to the internal session state information of a specific PHC and hence persists for a whole series of HTTP connections. The type `ROWID` defines a parameter to hold the identifier for a tuple

```
PHC Geo

// geographical selection
// version 1

INTERFACE

PARAM
    Province PUBLIC ROWID PRINT tProvinz.aBezeichnung[aProvId];
    Region PUBLIC ROWID PRINT tRegion.aBezeichnung[aRegId];
    District PUBLIC ROWID PRINT {SELECT aBezeichnung FROM tGemeinde WHERE aGemeinId = PARAM;};
ELEMENT
    Overview, OneUp, Province, Region, District;
LIST
    SelList;
STATE
    S0, //nothing selected, the first state is the start state
    S1, //province selected
    S2, //region selected
    S3; //district selected

IMPLEMENTATION

BEGIN

ELEMENT Overview (
    ALWAYS { PRINT 'overview'; GOTO S0; }
)

ELEMENT OneUp (
    ALWAYS { PRINT 'higher region level'; }
    STATE S0,S1 { GOTO S0; }
    STATE S2 { GOTO S1; }
    STATE S3 { GOTO S2; }
)

ELEMENT Province (
    ALWAYS { GOTO S1; }
    STATE S0 { PRINT NULL; }
    STATE OTHER { PRINT PARAM Province; }
)

ELEMENT Region (
    ALWAYS { GOTO S2; }
    STATE S0,S1 { PRINT NULL; }
    STATE OTHER { PRINT PARAM Region; }
)

ELEMENT District (
    ALWAYS { GOTO NULL; }
    STATE S0,S1,S2 { PRINT NULL; }
    STATE OTHER { PRINT PARAM District; }
)

LIST SelList (// very concise inline definitions and implicit assumptions
    STATE S0 QUERY IS
        SELECT aBezeichnung { PRINT; }, aProvId { GOTO S1 SET PARAM Province;}
            FROM tProvinz;
    STATE S1 QUERY IS
        SELECT aBezeichnung { PRINT; }, aRegId { GOTO S2 SET PARAM Region; }
            FROM tRegion
            WHERE tRegion.aProvId = PARAM Province;
    STATE S2 QUERY IS
        SELECT aBezeichnung { PRINT; }, aGemeinId { GOTO S3 SET PARAM District;}
            FROM tGemeinde
            WHERE tGemeinde.aRegId = PARAM Region;
    STATE S3 { NULL; };
)

END //Geo
```

Figure 2.8: PHC/DL description.

of a database table. The optional PRINT clause defines the output of the parameter which is printed instead of the parameter value if the parameter is used directly as part of the generated Web page[5]. Other possible datatypes are CHAR, VARCHAR, INT, FLOAT, STRUCT (a structure of other data types) or BAG (a relation of other datatypes). A parameter is implicitly defined as private. If a parameter is to be visible for other PHCs, it must be declared as PUBLIC. In addition, a default value can be defined as the start value of the parameter until it is otherwise set.

VAR A variable holds transient local information which is valid during the method execution within only one single HTTP connection but not the whole session. It can be of the same datatypes as the parameters.

STATE The state clause is used to define which branch is valid in which state. Special states are ALWAYS which is valid in all states and OTHER which is the default, if no explicitly named state is set. The possible combinations of all states and elements form a two dimensional table Each table cell contains the rules to be followed for PRINT and GOTO clauses for the respective combination. Thus in PHC/DL it is possible to define either different states nested within an element or different elements nested within one state. The first method has proven to be more convenient in many cases because different states can more often be combined to one rule (e. g. STATE S0,S1,S2) than different elements can. This results in a shorter and clearer overall description.

LIST A list can contain one or more elements. It defines a series of tuples, where each tupel contains an iteration of the element(s) of the list. The elements have the same functionality as a standalone element but an element in a list requires rules for its presentation and functionality *within each tupel* of the list. Hence a list definition always contains a QUERY clause, used as a cursor loop, to define the values used in the rules of each element for each tupel iteration. The parameter setting is more important for differentiating between the element iterations in the list; It is, however, not mandatory.

QUERY An SQL (Structured Query Language) query [Mis95] operation is needed to define a cursor for the values used in the ELEMENT and STATE clauses within a list. There are three possibilities for nesting queries, states and elements within a list: State – query – element if different queries are necessary in different states. Query – state – element or Query – element – state if only one query is used within the list, again with the two possibilities – either nesting states in elements or elements in states. The inline definitions of the PRINT and the SET PARAM clauses are used as a convenient abbreviation in many cases. The long form of the definition would include the declaration of a cursor name (directly after the keyword QUERY) and the explicit usage of the cursor variables within the cursor loop. An example of this is given later. If a query contains only inline definitions, the query need not be named. If a list contains just one element, this elements need not be named: It inherits the name of the list implicitly.

[5]This is essentially an abbreviation for a common case. The parameter can always be used within a PRINT expression to produce a different output.

`PRINT` The `PRINT` clause can only be used within an element as described above. It can have the following attributes:

- A parameter with the type `ROWID`: The output has to be defined with the parameter declaration.
- A parameter or variable of the type `CHAR`, `VARCHAR`, `INT`, `FLOAT`: The content is implicitly converted to a string.
- An SQL query which has a single result (one attribute, one tupel). This can be thought of as an abbreviation of having first a query of the type `SELECT ... INTO var FROM ...`, which fills a variable with the result and then a `PRINT var` statement to use the variable as output of the element.

`GOTO` The `GOTO` statement is used within elements to define the state manipulation that a user 'click' on this element would trigger and the new page delivered to the client afterwards. It has the following attributes:

- Directly after the keyword `GOTO`, the new state is set explicitly. If no state is named, the default state is taken instead.
- The optional `IN` attribute is used to set the state and parameter of *another* PHC instead of the current. The current PHC can be explicitly referred to as `SELF` or `THIS`, all other PHCs are referred to by their names.
- An optional list of parameter settings with the `SET` attribute determines which parameter will be set to which expression during the state manipulation method. If some parameters of another PHC have to be set, they have to be declared as public within this other PHC. The expression usually contains cursor attributes from the surrounding query.
- The `PAGE` attribute determines the logical name of the new active page and is a means of user guidance. This attribute is optional, the default causes the same page to be reloaded, which can also be explicitly named with `THIS` or `SELF`. In a single window mode without frames this simply means the next page to be delivered to the Web browser. If frames are used, the active page has to be displayed in the appropriate frame. If it has already been displayed, it has to be refreshed.
- The `REFRESH` attribute finally can be used to determine, which pages will change due to the change of the session state information. This clause is only necessary with frames and causes the named pages to be reloaded, if they are currently being displayed. Hence – with frames – the `PAGE` and `REFRESH` attribute together determine which page is (re-)loaded into which frame. This is described in more detail in 2.5 on page 29.
- Instead of the `PAGE` attribute an `URL` can be declared to be loaded into the current frame after completion of the state manipulation or functional method. This allows external pages to be displayed.
- The `TARGET` attribute can be used to define a target frame for the page or URL to be loaded. It is especially useful for URLs but it can also be used with pages. In the latter case this overrides the default frame of the page.

Embedded into procedural blocks, the `GOTO` clause may look like a jump statement from the old BASIC days. It is emphasized therefore that the `GOTO` clause defines the state transition, caused by a click on the actual element, and the new page(s) delivered to the client. It is *not* a means of control flow *within* the PHC/DL blocks.

`MESSAGE` A message poses an alternative to the `GOTO` statement: It defines a message with parameters[6] to be sent to a PHC upon user interaction (i. e. 'click') on this element, instead of the built-in method for state manipulation[7] defined by the `GOTO` clause. The `MESSAGE` statement can also have a `PAGE` or an `URL`, a `TARGET` and one or more `REFRESH` attributes. Furthermore, the method implementation itself can also contain a `GOTO` statement with some of these clauses: `PAGE` and `URL` would override the respective clauses from the `MESSAGE` call, but `REFRESH` pages are not possible. In contrast to the standard state manipulation method, a functional method implementation can also send another message to the same or another PHC.

`METHOD` To enable a PHC to *accept* a message, a *method* has to be defined within the implementation part of the PHC with the `METHOD` clause. There is a tradeoff between state manipulation and functional methods resulting from the question of what to store in the local session state information parameters of a PHC: Either the user's selection or the result based on a query using these selections. The answer is: If there are not too many different user selections, it is easier to use the parameters to store these selections and to use the state manipulation methods to change them. If there are many different user selections (and especially if their number cannot be determined in advance, for instance as is the case with a shopping basket), then it is easier to use a functional method to incorporate the actual user selection into a new result and store the result within the session parameters.

`CONDITION` With this clause, conditions needed for layout purposes can be defined within PHC/DL and later used within PHC/LL. Conditions are named and set within the branch of an `IF` statement. They can be used within the PHC/LL to define variations of the user interface layout depending on the condition. Especially useful if defined in PHC/DL within a list definition, they can be used in PHC/LL only in combination with an iterator defined from the same list.

More sophisticated examples of the usage of all these clauses are given in appendix A on page 89.

[6] Parameters are handed over by name, not by position.

[7] The state manipulation method can be thought of as a special form of message. However, the implicit definition with all the `GOTO` clauses of all the elements of a PHC has to be proven to be more efficient and convenient than an explicit definition of a state manipulation method.

2.4 Layout Language and Interface

Whereas the PHC/DL describes the PHC itself (functionality *and* output generation for each element), the PHC/LL is a procedural language embedded into HTML and describes *how* the PHCs' elements' output is integrated into the surrounding HTML code. The PHCI separates the PHC from the layout as far as possible[8] by allowing the PHC/LL to make use of the following parts of the PHC/DL only:

- The elements' `PRINT` and `IMAGE` methods can be used.
- The lists can be used for loop definitions.
- Predefined conditions can be questioned.
- The PHC's states are available.

It is emphasized that PHC/LL just defines the layout of the surrounding HTML page, whereas the structure, the string and image representation of the PHCs are already defined within the PHC/DL. This separation provides some advantages over the direct integration with HTML:

- It makes the PHC independent from the layout and thus from future changes to HTML.
- One PHC can be used a multiple number of times within different pages with different layouts, e. g. to satisfy user's needs. An example will show this later.
- The PHC/DL can be developed by application programmers and database experts, whereas the PHC/LL can be implemented by graphical designers and multimedia experts (often a more artistic than technical task). These two groups, with very different skills, only have to share a common PHCI instead of working together on one code.
- Even other frontends instead of HTML (VRML – Virtual Reality Modeling Language – or Java) are possible because of this strict separation.

Figure 2.9 shows the PHC/LL description belonging to the PHC/DL in figure 2.8. The generated browser view, depending on the actual state of the PHC, is shown in figure 2.3 on page 14. In state S2 we receive the generated HTML code as shown in figure 2.10 resulting in a browser view as shown in figure 2.11.

The language example is simple and self–explanatory: the embedding of the PHC/LL into the HTML code is done via SGML (Standard Generalized Markup Language) comments.

[8] Of course there are *some* relations between PHC/DL and PHC/LL: On the one hand PHC/LL is tailored towards the functional structure of the PHC/DL and on the other hand a difficult layout may require additional PHC/DL code. However, the key feature is that this mutual dependency is well defined to a very small amount of information - the PHCI.

```
<TABLE  ALIGN="right" WIDTH=115 BORDER=0>
    <TR><TD ALIGN="center" VALIGN=top><B>Region selection</B></TD></TR>
    <TR><TD><FONT COLOR="green">
        <!--$PHC PRINT(Geo.Overview); --><DUMMY>Geo.Overview</DUMMY>
    </FONT></TD></TR><TR><TD><FONT COLOR="green">
        <!--$PHC PRINT(Geo.OneUp); --><DUMMY>Geo.OneUp</DUMMY>
    </FONT></TD></TR>
    <TR><TD>
        <!--$PHC PRINT(Geo.Province); --><DUMMY>Geo.Province</DUMMY>
    </TD></TR><TR><TD>
        <!--$PHC PRINT(Geo.Region); --><DUMMY>Geo.Region</DUMMY>
    </TD></TR><TR><TD>
        <!--$PHC PRINT(Geo.District); --><DUMMY>Geo.District</DUMMY>
    </TD></TR><TR><TD ALIGN=center><HR></TD></TR>
<!--$PHC FOR item IN Geo.SelList LOOP -->
    <TR><TD ALIGN="right">
        <!--$PHC PRINT(item); --><DUMMY>Geo.Overview</DUMMY>
    </TD></TR>
<!--$PHC END LOOP; -->
</TABLE>
```

Figure 2.9: PHC/LL description.

```
<TABLE  ALIGN="right" WIDTH=115 BORDER=0>
<TR><TD ALIGN="center" VALIGN=top><B>Region selection</B></TD></TR>
<TR><TD><FONT COLOR="green"><a
href="s_pnav?phandler=515&pprovid=0&pregid=0&pgemeinid=0">overview</a></FONT></TD></TR>
<TR><TD><FONT COLOR="green"><a href="s_pnav?phandler=515&pregid=0">extend region
level</a></FONT></TD></TR>
<TR><TD><a href="s_pnav?phandler=515&pregid=0&pgemeinid=0">Nieder&ouml;sterreich</a></TD></TR>
<TR><TD><a href="s_pnav?phandler=515&pgemeinid=0">Mostviertel</a></TD></TR>
<TR><TD></TD></TR>
<TR><TD ALIGN=center><HR></TD></TR>
<TR><TD ALIGN="right"><a href="s_pnav?phandler=515&pgemeinid=24">Aigelsbach</a></TD></TR>
<TR><TD ALIGN="right"><a href="s_pnav?phandler=515&pgemeinid=6">Amstetten</a></TD></TR>
<TR><TD ALIGN="right"><a href="s_pnav?phandler=515&pgemeinid=12">Artstetten</a></TD></TR>
<TR><TD ALIGN="right"><a href="s_pnav?phandler=515&pgemeinid=23">Blindenmarkt</a></TD></TR>
<TR><TD ALIGN="right"><a href="s_pnav?phandler=515&pgemeinid=1">Eschenau</a></TD></TR>
<TR><TD ALIGN="right"><a href="s_pnav?phandler=515&pgemeinid=36">Euratsfeld</a></TD></TR>
.....
</TABLE>
```

Figure 2.10: Generated HTML code of PHC Geo in state S2.

Figure 2.11: Browser view of PHC Geo in state S2.

Thus, any HTML editor can be used to design the surrounding layout without being confused by the embedded language. The disadvantage is that a standard HTML editor or browser would display nothing instead of the elements' output, ending up with an ugly and unrepresentative layout in most cases. To solve this problem, the dummy tag is introduced: A standard HTML browser or editor would ignore this tag and treat the contents of it as normal text. The tools on the other hand remove the dummy-tag including the content. With this 'trick' similar to the `NOFRAMES` tag for frame-incapable browsers a representative layout can be achieved with the PHC/LL source code.

The most important language elements of PHC/LL are the `PRINT` and the `IMAGE` statements, which include the textual or image output of an element into the HTML code. The `FOR` loop defines an iterator (`item` in figure 2.9) for a list. Within this loop all the element(s) of the list can be named using the iterator and the name of the element. If the list contains just one element, the iterator alone is enough. The naming of the iterator also allows nested loops.

2.5 Building a Complete Information System

To build a complete, pure HTML based Web application one has to tackle the following tasks:

- Design of the functional elements (PHCs and forms).
- Description of the pages, either static pages or pages containing functional control elements embedded into HTML.
- Logical arrangement of the pages and overall structure.

- Crosslinks from one page to another (internal or external).
- Design of the frame layout of the pages, i.e. which page has to appear in which frame.
- Multilingual support. Although the Web 'speaks' mainly English, multilingual support is necessary as a convenience and courtesy to the end user, especially in Europe.
- Security and user rights

With the PHC framework these design tasks are solved as follows: Functional Elements and Database transactions are defined with PHC/DL as shown in the previous sections. The pages are pure HTML and control elements are embedded via PHC/LL. Each page has a unique logical name under which it is stored in the database. Even static pages without any control element are stored in the database. This makes the Web-Management [Beh98] easier, too.

Logical Page Flow and Crosslinks

Links have to be bidirectional to enable maintenance of the complete system. Within the PHC framework every anchor tag is either generated dynamically from a PHC function, from a logical page arrangement or from the crosslink database. The logical page arrangements include:

- hierarchical document collections
- linear document collections
- nested structures
- table of contents, index pages, full text search, menues
- predefined logical page flows (e.g. one or more guided tours)

Crosslinks, on the other hand, are defined within the HTML pages with the PHC/LL clause `GOTO` (e.g. `GOTO PAGE Result`). Each time a static page in the database is inserted, modified or deleted, a trigger will cause a parsing of this page and update the bidirectional logical link database from the `GOTO`-statements contained in this page. Hence it is impossible that internal links can fail. If a page should be removed but links are pointing to that page, a warning is generated. Ignoring this warning results in the removal of all the links pointing towards this page. External links are also defined by the `GOTO` clause (e.g. `GOTO URL "http://some.where.org/page.html"`) and the links are also stored in the database. In this case, a Web robot has to check periodically if the remote links are still valid and sends an email to the webmaster if at least one link fails.

Frames and Window Modes

PHC/LL is used to define the different framesets and pages: Each page and frameset is defined with PHC/LL as shown in figure 2.12 and each page or frameset has its place in one (or more) framesets, depending on the actual mode. Figure 2.13 shows an example of the overall frame structure of an application. A window repository is extracted from the PHC/LL definition of the framesets and stored in the database. Each time a dynamic HTML page is generated, the anchor tags within that page are provided with the appropriate target attributes. Also the `REFRESH` and the `PAGE` clauses of the `GOTO` clause are resolved at that time: The page from the `PAGE` clause has to be displayed in the appropriate frame and may therefore determine which new frameset is to be displayed. If the page is already displayed, it has to be refreshed. All the pages from the `REFRESH` clause – but only if displayed in the actual frameset – have to be refreshed (reloaded). To reload more than one frame without JavaScript, the special HTML target values "`_parent`" and "`_top`" are used. Hence, when generating a page, the window repository is used to automatically generate the correct target values from the `PAGE` and `REFRESH` definitions of the respective elements, computing *in advance* which pages will have to be reloaded into which frame if the respective element is 'clicked'.

The generated button bars for navigating the document collection structure and all internal links are also adapted to the current frame structure: If a page is already displayed in a frame of the actual frameset, the navigation buttons and links pointing to that page disappear. The reason for this is that a user would get no reaction from clicking on such a link unless named tags are used to jump to a particular position within a page (in this case the appearance of the links is not suppressed).

Another nice and useful feature has been implemented into the PHC framework: A page can be configured to have a full-screen button  which enables the user to display the page in a predefined parent frameset as shown in figure 2.12 with the page named "navigation":

```
<!--$PHC PAGE navigation MODE Expert IN nav FULLSCREEN main -->
```

When the page appears in full-screen mode, another button  allows the user to switch back into framed mode. Both buttons are simply implemented as images with anchors.

Multilingual Support

Multilingual support is 'built-in' in the PHC framework. Everything printed with the `PRINT` clause in the PHC/LL description, either user interface strings or element output, is multilingual from the start. One language has to be selected as the default language. In addition a set of database attributes is usually defined to be multilingual thus replacing the attribute by a foreign key to the dictionary, which contains the attribute in all the

```
<!--$PHC
ROOT Demenet;
LANGUAGES English DEFAULT, Deutsch, Suomi;
MODES Beginner, Expert;
-->

<!--$PHC FRAMESET f_overview MODE Beginner, Expert IN ROOT; -->
<HTML>
<TITLE>DEMENET</TITLE>
<FRAMESET ROWS="65,*" border=1>
<FRAME NAME="title" SRC=<!--$PHC FRAME title; --> scrolling="no" marginheight=1
marginwidth=1 noresize>
<FRAME NAME="main" SRC=<!--$PHC FRAME main; --> marginheight=1 marginwidth=1 >
</FRAMESET>
</HTML>

<!--$PHC FRAMESET f_product MODE Expert IN main AS DEFAULT; -->
<HTML>
<TITLE>DEMENET - Farmer Products</TITLE>
<FRAMESET cols="380,*" border=1>
<FRAME name="nav" src=<!--$PHC FRAME nav; --> marginheight=1 marginwidth=1>
<FRAME name="output" src=<!--$PHC FRAME output; --> marginheight=1 marginwidth=1>
</FRAMESET>
</HTML>

<!--$PHC FRAMESET f_results MODE Expert IN output AS DEFAULT; -->
<HTML>
<TITLE>DEMENET - Results</TITLE>
<FRAMESET rows="50%,50%" border=1>
<FRAME name="out" src=<!--$PHC FRAME out; --> marginheight=1 marginwidth=1>
<FRAME name="caddy" src=<!--$PHC FRAME caddy; --> marginheight=1 marginwidth=1>
</FRAMESET>
</HTML>

<!--$PHC FRAMESET f_admin MODE Expert IN main; -->
<HTML>
<TITLE>DEMENET</TITLE>
<FRAMESET cols="320,*" border=1>
<FRAME name="menu" src=<!--$PHC FRAME menu; --> marginheight=1 marginwidth=1>
<FRAME name="edit" src=<!--$PHC FRAME edit; --> marginheight=1 marginwidth=1>
</FRAMESET>
</HTML>

<!--$PHC PAGE welcome IN ROOT AS DEFAULT; -->
<HTML>
...
</HTML>

<!--$PHC PAGE menu MODE Beginner, Expert IN title; -->
<HTML>...</HTML>

<!--$PHC PAGE home MODE Beginner IN main MODE Expert IN out, output; -->
<HTML>...</HTML>

<!--$PHC PAGE product MODE Beginner IN main AS DEFAULT -->
<HTML>...</HTML>

<!--$PHC PAGE navigation MODE Expert IN nav FULLSCREEN main -->
<HTML>...</HTML>
```

Figure 2.12: PHC/LL definition of the various framesets and pages.

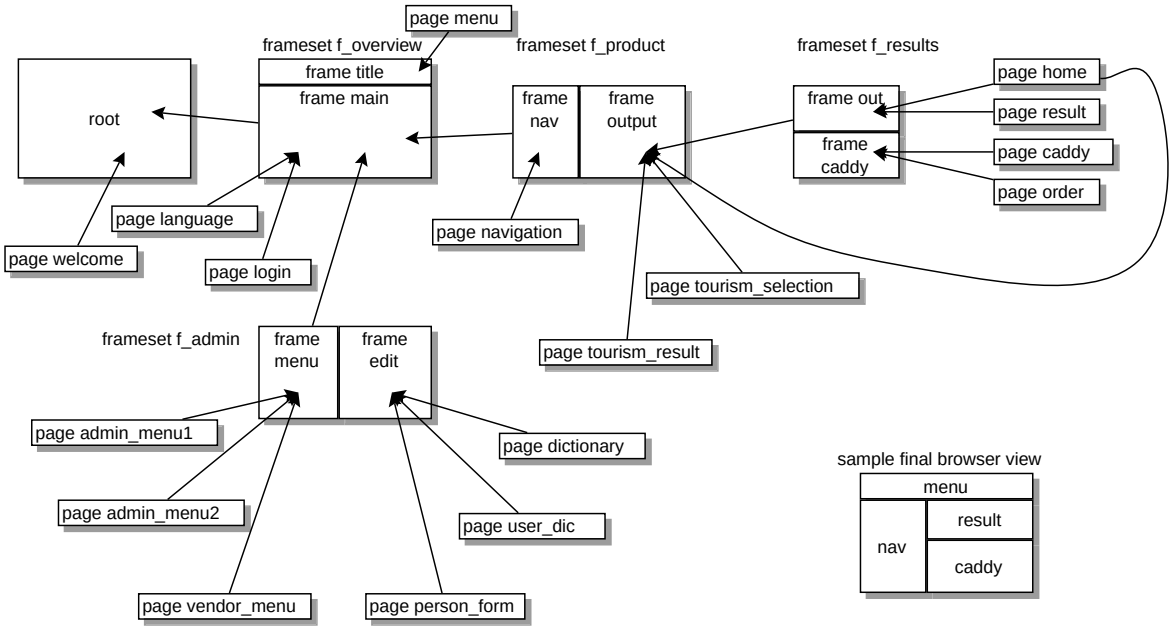


Figure 2.13: Overview of frameset and page definitions in the mode 'Expert'.

required languages. For administration purposes a Web interface is provided to edit the various languages in the dictionary. This enables Web collaboration for administrative purposes on the final application. Interpreters from different countries can translate the dictionary entries into the different languages. The advantage of that structure is: In order to add a new language, nothing has to be recompiled. It is only necessary to make one single entry indicating that the new language is available. The dictionary has then to be filled in for that language by interpreters from all over the Web.

Security and User Rights

First users and groups of users have to be defined. Rights can then be granted to users or groups. Rights include system privileges (e. g. to edit the dictionary) and object privileges. Object privileges can be the right to see a page or frameset or to use a form or a PHC. With PHCs the rights can further be distinguished between:

- the right to use the state manipulation method of a PHC
- the right to use a particular functional method of a PHC
- the right to see the output of a PHC

2.6 Implementation

Whereas the advantages of a client-server model are exploited for design, in the case of pure HTML the final implementation of a PHC is done on the server side. No functional

elements are needed on the client side. All 'intelligence' is thus located on the server making the control element *passive* in this sense. Every Web page, including all links, is dynamically generated from the database and the session state of all PHCs within this page. The hyperreference URL of every link contains parameters for the global session handle and state transition and parameter information of the appropriate PHC.

In this current pilot implementation these parameters are passed in clear text, which is especially useful during development. For the final release, all parameters will be encoded together into one string including a randomly generated authentication identifier to avoid the possibility of misuse. The lifetime–problem of state information is overcome by a timeout mechanism. It purges old state information from the database if the session was not closed properly, which is typical on the Internet. The problem with the browser cache that every URL-based state maintenance mechanism has to face can be overcome by setting the HTTP response header field 'Expires' to a date in the past [Gra96].

Different Implementation Methods

Considering the way an abstract description of a Web application is finally implemented we can differentiate the following approaches: Generation and interpretation. As with general purpose programming languages, interpretation results in a more flexible development environment at the cost of a slower runtime system: Modifications can be incorporated into the running system without the need for recompilation. On the other hand generation (or compilation) results in a faster final version at the cost of a more complex development environment: Each modification demands a recompilation of at least parts of the system, but finally the compiled code is faster. Hence a design decision has to be made primarily as a tradeoff between performance and flexibility.

A straight-forward approach is to generate a static HTML document tree from the abstract description of the structure of the logical pages. This results in the fastest possible Web interaction – but only as long as no database interaction is required during runtime. However, this approach is reasonable for information systems whose content does not often change and where seeing 'old' content until a new tree is generated from the continously changing database is not a hindrance. Equally importantly, the Web service causes almost no additional load to the database as the database is only required during generation.

If database interaction is required during runtime (e.g. either because user input into the database or up-to-date information is required) the generation of a static document tree is no longer possible. Hence, with Web applications consisting of static[9] *and dynamic* pages we can further differentiate between the following possibilities concerning the runtime environment of the PHCs, the static pages and the dynamic pages respectively:

PHCs Two possibilities exist for the runtime environment of PHCs:

[9]In this context 'static' means that no database interaction is included within this page. It does *not* mean that the page is a pure HTML page within a file system.

- After lexical analysis and parsing, the PHC/DL and PHC/LL descriptions are stored in a design repository within the database. If a message for a state manipulation method or a functional method or an output method is received, the interpreter manipulates the database and constructs the output according to the appropriate PHC information stored in the design repository.
- Alternatively, a generator generates PL/SQL (Procedural Language with embedded SQL) procedures directly implementing the methods for state manipulation, functionality and output generation. In this case, if a message is received, a compiled procedure is called with the appropriate parameters.

static pages Furthermore, these same two possibilities exist for static pages: Either an output procedure is generated for each page doing nothing else other than printing out static HTML code or the templates are stored in the database containing meta-information for the logical structure. During runtime and before a page is delivered to the client, the meta-information is interpreted and replaced by anchors for navigation and crosslinks dynamically generated from the meta-tags. The first approach is moderately faster of course, but it is rather inflexible: Even a simple change to a static page requires the recompilation of at least some parts of the system.

dynamic pages Dynamic pages can also either be stored as templates with meta-information marking where to insert the PHCs' output, or they can be directly implemented as compiled PL/SQL procedures printing 'calculated' HTML code.

All but the first approaches have one point in common – a database is required for the runtime environment. Hence it is reasonable to put everything into the database, even static templates. The database tools for distribution, replication and backup can then be used in a homogenous way for the whole system. The following subsections describe both approaches and further details can be found in [Fal98].

Interpreter and Design Repository

Interpretation and generation can efficiently be combined to form one approach shown in figure 2.14: The interpreter is used as a design tool and the generator finally compiles the design repository into procedures, runtime repository and the schema for the dynamic session state information (see next section) thus building the final application.

The PHC/DL and PHC/LL source files are firstly lexically analyzed and parsed and a symbol tree consisting of tokens and their attribute values is built. This symbol tree is then stored in the database as relational data structure and called 'design repository'. During interpretation, the current session state information (states and parameter values of all PHCs of all active sessions) is also stored in the design repository[10]. With the runtime interpreter, this design can now be immediately tested. Although this works more slowly than the final implementation, interactive design tools provide direct access

[10]Hence the design repository is also used as runtime repository with the interpreter.

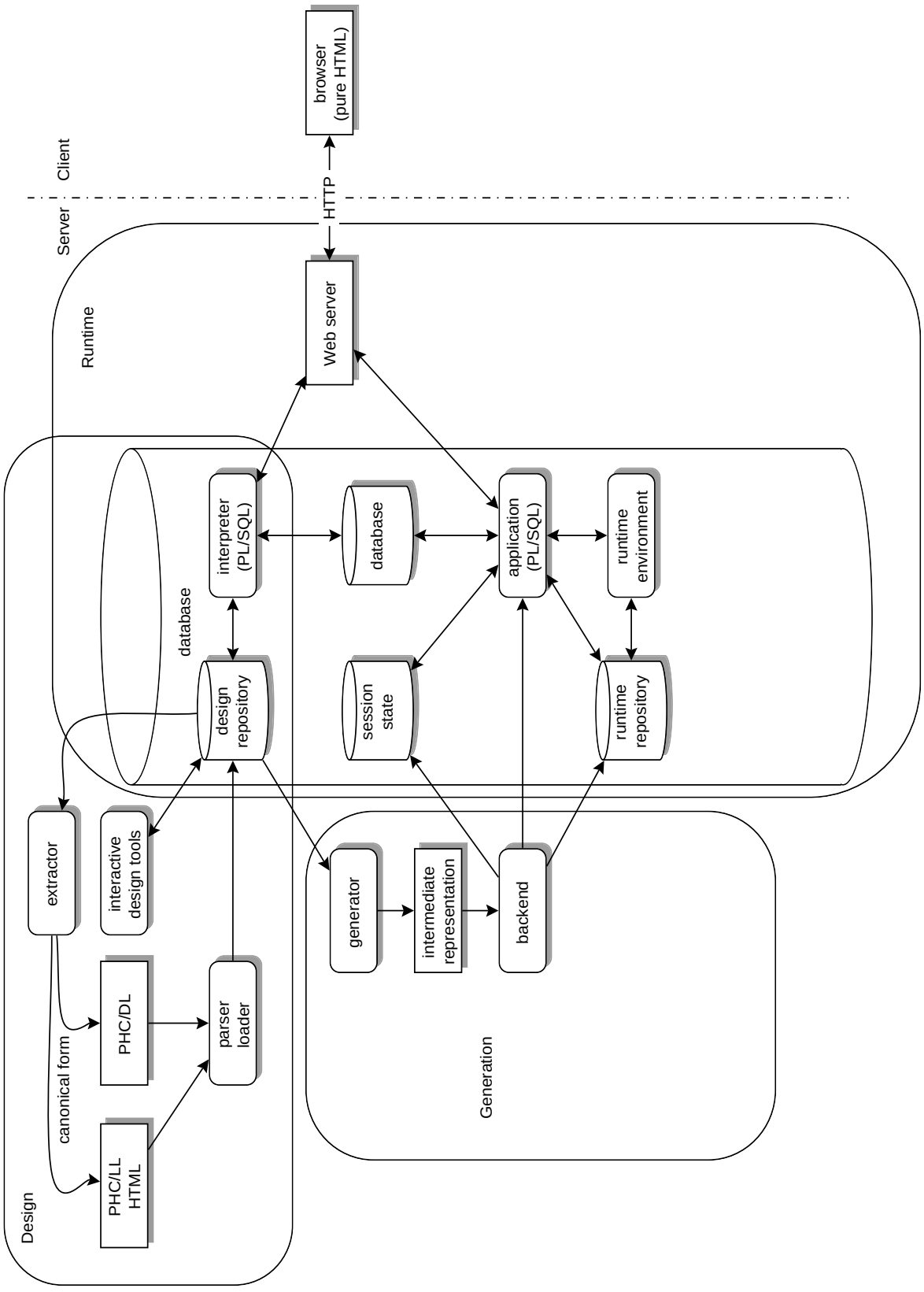


Figure 2.14: Design and two different implementation methods: interpretation and generation.

to the design repository via a graphical user interface: This allows direct manipulation of the design within the repository and immediate testing of the changes with the interpreter. One of the design tools is an extractor rebuilding canonical PHC/DL and PHC/LL descriptions of the changed design repository.

It should be mentioned that the design tools themselves are also implemented as a database backed Web application. Hence some design teams distributed all over the world can collaboratively work on the same application by simply using a Web browser with pure HTML on the client side!

The scenario for a runtime client-server interaction using the interpreter is shown in figure 2.15: Initially a message is received from the user interface. The interpreter reads the appropriate PHC/DL method definition (either state manipulation or functional method) from the `GOTO` definition of the 'clicked' element in the repository and starts to interpret it. In the case of a state manipulation method, a simple lookup into the state machine table – stored explicitly as relation in the design repository as $old\_state \times element \Rightarrow new\_state$ – is all that is needed. This may result in state manipulation and database transactions and the new state and parameter values are stored in the session state information.

The next output page is then determined and its PHC/LL is interpreted. This causes the interpreter to read the PHC/DL output method definitions of all the contained PHCs with respect to the current session state. Each SQL statement is first parsed and the placeholders for the parameters and variables are replaced by the actual values. The SQL statement is then executed with dynamically embedded SQL and the result set is included in the HTML skeleton. As a result the complete HTML page has finally been constructed.

This is the most complex and most time consuming part of the interpretation. One main reason for the lacking performance of the interpreter approach is the heavy use of dynamically embedded SQL instead of statically embedded SQL as is the case with generation.

Generator and Runtime Repository

The generator leaves the final choice of what to generate and what to interpret to the designer:

- For a set of static pages defined solely with PHC/LL a static document tree can be generated.
- If there is a large number of static and dynamic HTML pages, it is reasonable to store these pages as templates within the runtime repository with meta-information for navigation. Procedures are only generated for the methods of the PHCs (state manipulation, functional and output). This scenario is shown in figure 2.16: A page interpreter reads the template and then calls the output procedures (O–procedures) of the contained PHCs. These O–procedures have been generated from the PHC/DL description alone, while the PHC/LL descriptions are converted into the page templates.

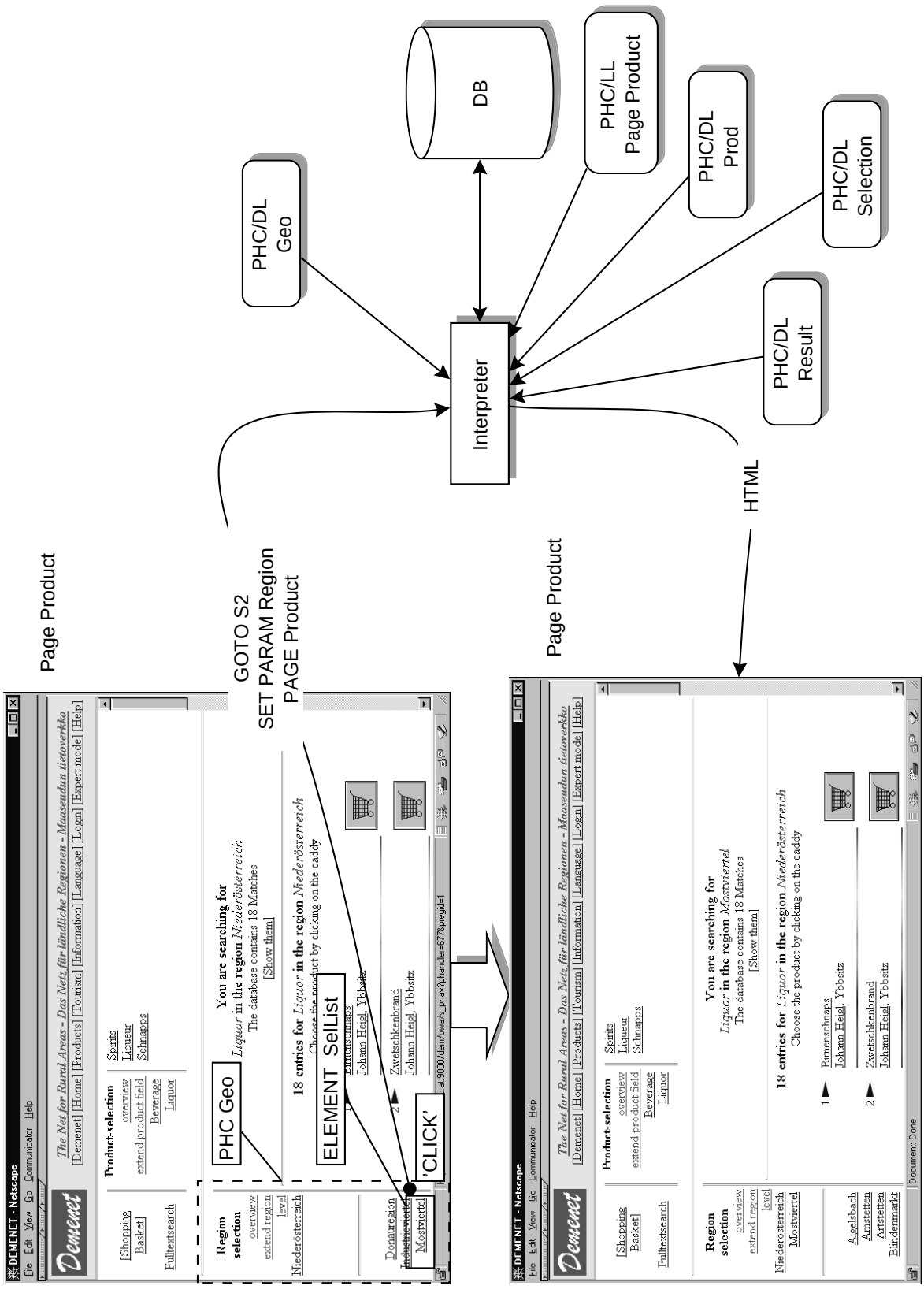


Figure 2.15: Interpreter and repository.

- If there are lots of static pages but only a few dynamic pages, the dynamic pages are also implemented as compiled procedures instead of templates. These procedures are generated from the the PHC/LL descriptions of the dynamic pages and incorporate the output methods (PHC/DL) of the contained PHCs resulting in the fastest possible generation of the dynamic pages. This scenario is presented in figure 2.17 and is used for the prototype implementations described later.

- If there are only a few static pages and performance is crucial on the server side but flexibility in changing the pages' contents is less so, then the static pages can also be implemented as compiled PL/SQL procedures.

The generator takes the design repository as input and compiles it into an intermediate representation. Different backends finally produce the sourcecode of the methods for different database systems and different programming languages[11]. A runtime repository, which contains HTML templates for static and dynamic logical pages, images, window mode and frame definitions, dictionary, user management, rights and security is also generated. Finally, the schema for the dynamic session state information has to be created in the database dictionary.

As mentioned previously, a PHC can be 'called' in two different ways: These two abstract ways now find their concrete equivalence in two different kinds of procedures used to implement the PHC functionality:

- The S–procedures implement the structural methods – either the state manipulation or the functional methods of a particular PHC. A state manipulating S–procedure typically takes an element and eventually some parameters as input and then calculates the new state and the new parameter set. Functional S–procedures typically take some parameters and perform some database transactions. At the end of the S–procedure, the ouput page is determined. When the page was generated from which the current message came, the next output page has usually been calculated from the active page, the refresh pages and the window mode *in advance*. This page has been encoded into the anchor and was hence sent to the current S–procedure as a parameter. However, a functional S–procedure can also calculate its own output-page thereby overriding the predefined page. This mechanism has been described by the `MESSAGE` clause on page 24. Finally, the S–procedure calls the appropriate O–procedure.

- The function of the O–procedures depends on the implementation method used for dynamic pages: If the dynamic pages are stored as templates, each PHC has its own O–procedure which prepares the output strings of all elements of the PHC. If dynamic pages are implemented as stored procedures, one O–procedure for each page prepares the output strings of all PHCs within this particular page and combines them with the surrounding HTML code to generate the complete Web page. In

[11] Only PL/SQL for an Oracle database is currently considered, but the separation into generator and backend allows for future cartridges e. g. Informix, Sybase, MS SQL server or even a generic backend to generate Perl code for a FastCGI interface connected to an ANSI (American National Standards Institute) SQL database.

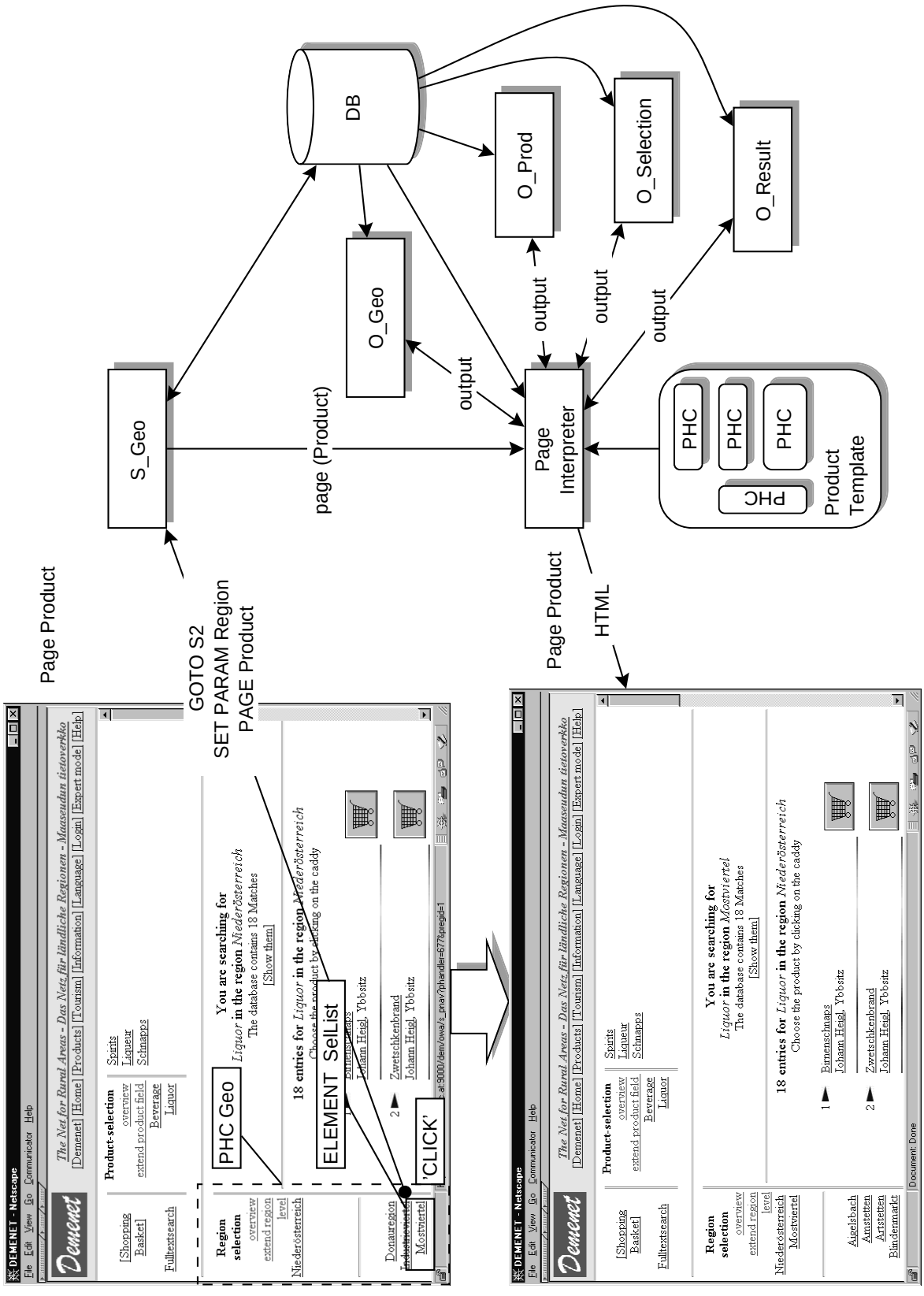


Figure 2.16: Generator and runtime repository with pages implemented as templates.

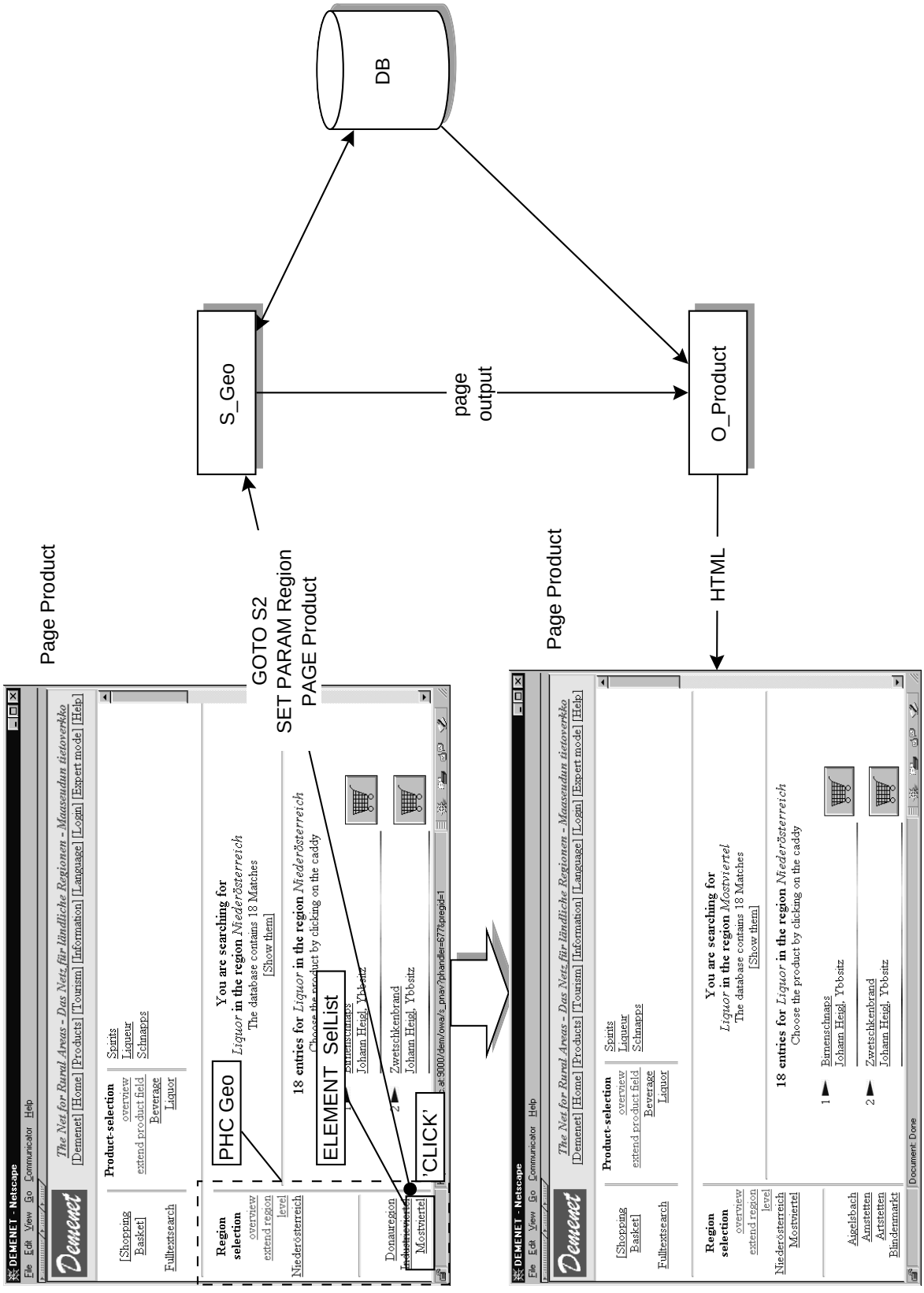


Figure 2.17: Generator and runtime repository with pages implemented as procedures.

both cases the most important task of the O–procedures is to generate the anchor tags for all elements of all the PHCs within this page. From the `GOTO` clause of an element, especially from the `PAGE` and `REFRESH` attributes, the next output page – i. e. the page to be loaded after a click on this element – is computed in advance and encoded into the hyperreference of this element as an additional parameter. If the page has to be loaded into another frame or more than one page has to be reloaded, then the appropriate target value also has to be determined and the output page may be a whole frameset instead of a single page.

Hence the separation into S– and O–procedures reflects the strict separation into state transition and output generation mentioned earlier – the S–procedures produce no output and the O–procedures cannot change the session state. Among other advantages it allows a more stable and robust pure HTML interface. Figure 2.17 shows the implementation model belonging to the design model from figure 2.6 on page 18.

Look-Ahead Link Generation

Both approaches, interpreter and generator, dynamically generate a sequence of HTML pages. In both cases the most important task of the O–procedures is to generate the anchor tags for all elements of all the PHCs within this page. There are two possibilities for generation of the hyperreference URLs of the page:

1. The first possibility is quite straightforward: The link of an element contains the session handle (for state maintenance), the name of the element and the name of the PHC the element belongs to[12], and the name and value of the parameters to be set:

```
<a href="fsm?handle=17&phc=Geo&element=SelList&pRegion=17">Mostviertel</a>
```

   This information is always sent to the same function `fsm` implementing the state machine for the whole system. From the `GOTO` or `CALL` clauses of the PHC/DL declarations of the appropriate element on the one hand and from the current state of the corresponding PHC (stored in the current session state information) on the other hand it can then be determined which function has to be called or which new state has to be set. Afterwards, the next output page can be determined from the `PAGE` attribute and finally be generated completing one circle.

2. The other possibility is to compute *in advance*, what has to be done when an element is clicked on – and to generate this information into the hyperreferences of the elements' anchor tags (see definition 5 on page 12): Which functional or state manipulation method has to be called, possibly which *new state* to be set, some parameters with their values and the name of the next output page.

[12] Element names are local within a PHC and need not be unique throughout the whole application.

```
<a href="S_Geo?handle=17&state=2&pRegion=17&pPage=Product">Mostviertel</a>
```

Now when an element is clicked, the appropriate functional or state manipulation method is directly called with the parameters already encoded into the URL. One of these parameters is the next output page, explicitly defining which page to be generated next, completing one circle. This is in accordance with notation 2.6 on page 13. Figure 2.18 shows one circle. In this case, the O–procedure is enhanced by some parts of the S–procedure in order to compute the possible new states in advance. The S–procedure only takes the parameters and sets the new state.

At a quick glance the first method seems to be very intuitive for the interpreter implementation[13] because it is easier to resolve only *one action after* a particular element has been clicked rather than to compute all possible actions for *all elements* within one page *in advance* and to encode them into the URLs. The second method, on the other hand, seems to be more convenient for generation because the appropriate functions are called directly with their parameters. Moreover, with generation the time consuming look-ahead takes place during compile time and hence needs no additional computing power during runtime.

However, both statements are true as long as we do not use frames. If we do, the first method causes problems: From the `GOTO` clause of an element, especially from the `PAGE` and `REFRESH` attributes, the next output pages – i. e. the pages to be loaded after a click on this element – have to be computed in advance and encoded into the hyperreference of this element as an additional parameter. This is required if the page has to be loaded into another frame or more than one page has to be reloaded. The reason for this is that the appropriate target value also has to be determined and the output page may then be a whole frameset instead of a single page.

Hence the interpreter has to at least compute in advance which page or frameset to load into which target frame in order to generate the appropriate target values. The other information, such as which action to call or which new state to set can still be determined *after* the element has been clicked on, taking some computing load off the interpreter. The resulting anchor might then look like this:

```
<a href="fsm?handle=17&phc=Geo&element=SelList&pRegion=17&pPage=Product">
```

Library Concept and Reusability

Though working with PHCs is very efficient compared to direct PL/SQL implementation, it is still a method of database programming and needs programming skills. A library of

[13] This method has in fact been used on page 33 to explain the interpreter because it is the more intuitive approach. The interpreter can of course also use the other method of link generation.

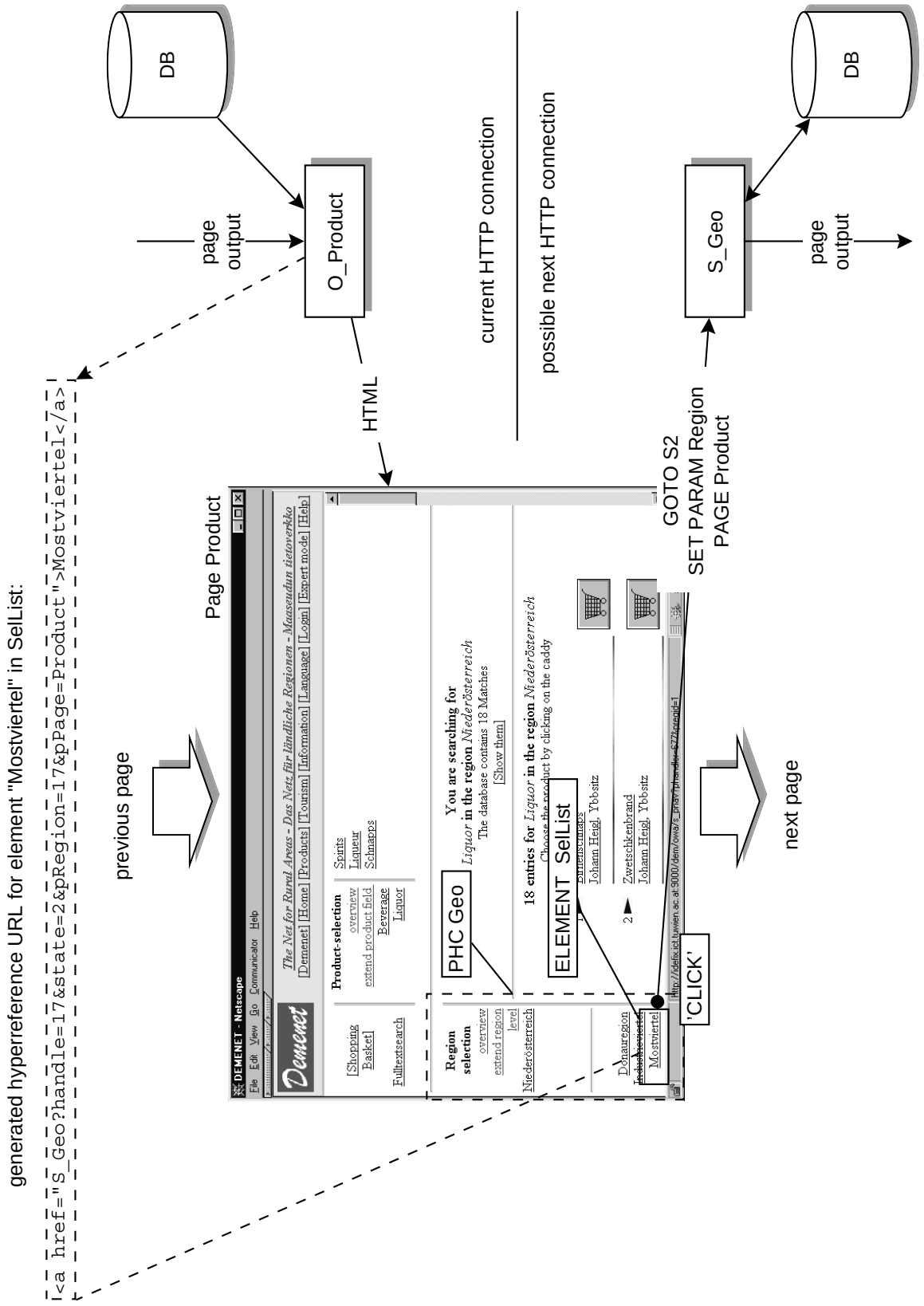


Figure 2.18: Look-ahead link generation.

PHC/DL modules can overcome this problem by providing a rich set of ready-to-use PHCs for many purposes. This requires the PHCs to abstract from the ER part of the concrete underlying database to an ER structure with some minimum structural requirements. The PHC can then later be used with any database – if the structural requirements are fulfilled – by simply assigning the appropriate database entity and attribute names to the parameters. Applications can hence be built easily by putting together some PHCs from the library. This introduces a method of reusability [GJM91] of PHCs.

2.7 Limits and Improvements

Firstly this section summarizes the main features and advantages of the PHC approach. A critical view on PHCs then leads to possible future works:

- Strict separation between the layout and the structure of PHCs. The functionality, internal state information and the database transactions are completely independent from the user interface representation of a PHC. A PHC is neither assigned to a single HTML page nor is it limited to HTML as client side user interface representation. This is one main difference between PHC and other approaches [BS98, ⇒heitml]. It allows PHCs to be used in more than one page, in different HTML layouts or even with different client side implementations (Java, VRML).
- Compact integration of functionality, database transaction and session state handling.
- PHCs use links instead of form elements to implement user interaction in most cases. Forms are only necessary to solicit user data[14].
- The sound theoretical model of a state machine is useful for later theoretical and practical work on design tool support and on automated testing and verification (correctness proof) of the implemented applications.
- Web applications designed with PHCs are robust and stable despite the stateless nature of the HTTP protocol and the inability of the browser to store user interface state information. This is achieved through the strict separation of state transition and output generation mentioned earlier: A user interaction may result in a changed session state information – the output of a PHC is, however, only dependent on the new session state information but not on the last user interaction. As long as the affiliated state information is not changed, a dynamically generated Web page will always look the same, regardless of the user interactions which have taken place in the meantime.
- The pure HTML approach guarantees compatibility with almost every browser on the Web without getting involved in the competition between Netscape [⇒NS] and

[14] Of course one could implement a HTML keyboard consisting of links making forms completely obsolete. However, this would definitely *not* result in a more comfortable user interface.

Microsoft [⇒MS], while the World Wide Web Consortium [⇒W3C] tries to find a consensus on the standards [RCK98]. A citation from Tim Berners-Lee – the inventor of the Web himself – says it all [⇒Any]:

> "Anyone who slaps a 'this page is best viewed with Browser X' label on a Web page appears to be yearning for the bad old days, before the Web, when you had very little chance of reading a document written on another computer, another word processor, or another network."

The following improvements are planned in future work:

- A graphic version of the design language PHC/DL including tool support. Methods used for structured hypermedia design [ISB95] seem to be a good starting point for further development in this area.
- Enhancement of the generator tools to provide Java (see section 3.3) and VRML [BW98] as alternative user interface implementations. Especially the integration of Java and VRML provides new possibilities [Bru98].
- Enhancement of the pure HTML interface with a scripting language as soon as it becomes clear which one[15] will be the major client side scripting language.
- Complete object oriented PHC/DL instead of the current object based approach, e. g. a subset of Java could be used as PHC/DL. When Java replaces the current procedural database programming languages and the Java virtual machine is integrated with the object relational database kernels [Ros98], this approach starts to become interesting.
- Object oriented integration of information design and functionality design, ideas can be found in [BBB+95].
- Object oriented notation also for the PHC/LL – e. g. `<PHC:Geo.OneUp.PRINT>` – by possibly using the Extensible Markup Language XML [Hol98] or a user defined SGML [Jel98, Bra97] subset. Object oriented approaches like [BS98, ⇒heitml] will be evaluated with respect to their usability as a PHC/LL replacement.
- In discussing the Internet we have subsequently to talk about security: A security framework must therefore be integrated into the PHC approach:
  - To avoid misuse of the session handle, encoded handles are used. Differently encoded Cookies[16] can be used *in addition* to URLs - forming a handle consisting of a Cookie and an URL - to further reduce the possibility of misuse.

[15] Currently Netscape's JavaScript and Microsoft's VBscript have to be taken into account.

[16] URLs and Cookies can both be modified by a malign user and therefore do not provide full security. However, the combination of both at least reduces the possibility of erronous user behaviour and makes it very unlikely that a malign user could guess a correct combination of Cookie and URL.

  - Sequence numbers (i.e. a new encoded session handle for each HTTP connection) would provide even better security, but this method is not compatible with frames. However, with frames a pool of URL-Cookie pairs can be used and each page contains a randomly selected handle from this pool.
  - Where user authentication is necessary for database updates, Netscape's SSL (Secure Sockets Layer [Smi98], possibly using RSA [BSW98]) is used for better security.
  - Integrated SmartCard solutions will be tested as they become available [MS98].
  - For authentication purposes a combined method of database authentication (process based), Web listener authentication (URL based) and application authentication (user based) provides the maximum possible security.

- More sophisticated versions of the HTTP (Currently version 1.2 is under development [⇒W3C] though 1.1 is not yet used in all servers and browsers) promise better support of connection orientation. In fact this would avoid the need for a session handle to be passed back and forth between client and server. With pure HTML on the client side, the user interface state would still have to be stored in the database however. The look-ahead link generation would also remain unchanged.

Chapter 3

Object Oriented Approaches

This chapter compares different object oriented approaches with the previous pure HTML approach and with each other. After defining some criteria to differentiate between these approaches some of them are discussed in more detail with practical results from prototype implementations. Finally, a short comparison concludes this chapter. Java is *the* object oriented implementation language of choice on the Web, therefore this chapter focuses on Java implementations for the Web. Whereas some of the tradeoffs described here are wellknown e. g. from C++ implementations, Java changes the boundary conditions and as a result new possibilities arise.

3.1 Client-Server and Persistence

The first rough criteria to separate the different approaches is the decision whether to use

- no Java at all. The possibilities of pure HTML combined with relational databases have been described in detail in the previous chapter.
- Java with a relational database. Different client-server models and different persistence approaches will be discussed.
- Java with orthogonal persistence in an object oriented database.

The first two approaches use relational databases. The motivation to enable relational databases for the Web is based on the following reasons:

- Large number of existing legacy database systems[1] – organizations want to leverage their investments in relational technology.

[1]Some of them not even relational: Many network and hierarchical databases are still in use. Nowhere else it is so important to deal with legacy systems than with the large old database dinosaurs.

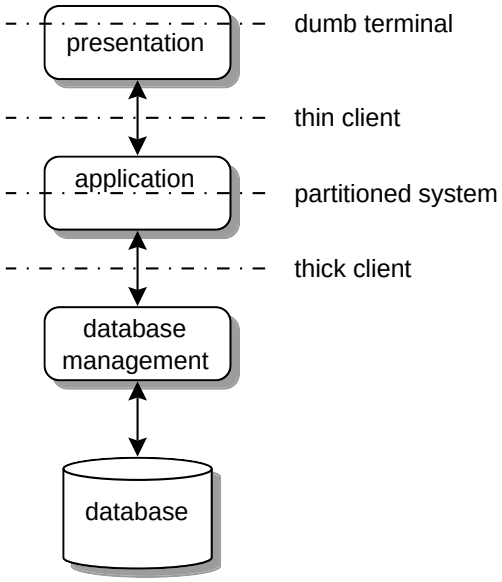


Figure 3.1: Client-server model.

- Relational database products have needed a long time to become mature, whereas object oriented databases still suffer from some childhood-illnesses.
- There are tasks which can best be solved with relational databases; Typically, large amounts of simple structured data.
- Sophisticated design tools are available to overcome the inherent deficiencies of the design method (see page 19).

Java makes it possible to implement functionality not only on the server side but also on the client side. We therefore have to face the classic client-server tradeoffs, but with different boundary conditions introduced by the Web and Java. Figure 3.1 first recalls some possibilities for distributing an application between client and server:

Dumb terminals: Software control remains with a mainframe host, dumb terminals use a Character Based User Interface (CBUI). Whereas this is a very traditional architecture, a pure HTML interface is not very different[2]: Some of the presentation is done by the standard client (Web browser) but most of the presentation logic lies on the server side, where the HTML page is dynamically generated. This architecture is very reliable and easy to upgrade but uses host and network resources extensively.

Thin clients: The client is now responsible for the presentation logic: It generates requests to the server and presents, and eventually stores, the results. This architecture takes some load off the server and the network but introduces more complexity due to usual heterogenous environments: Many layers and components have to be compatible with each other on the network.

[2]The main difference is the rich graphic user interface provided by HTML instead of the CBUI.

Partitioned client-server system: Application functionality (or middleware) is distributed between client and server with the aim of achieving optimal performance (throughput and latency) by minimal resource usage on the client, the server and the network.

Thick client: The server is reduced to the database management system, the complete application functionality resides on the client side. This option becomes reasonable, if much computing power is available on the client side: Modern PCs allow easily for thick client software. In fact it often costs less to put together a system of much smaller computers which has the equivalent power of a single powerful machine.

Java has changed the boundary conditions of client-server architectures: Java applets bring the advantages of dumb terminals to even thick clients with rich graphical user interfaces: A Java applet is platform independent[3] and located on the server side but is finally executed on the client side, taking some load off the server and the network. As a result, in order to upgrade such an application, the server only has to be upgraded for one single platform: The upgraded applet is then downloaded to the client. To sum this up, a Java client-server application has all the ease of maintenance and upgrade of a basic dumb terminal environment but also has the previously mentioned advantages of a much thicker client-server environment [WM97]. This is also the key idea for the concept of *network computing*[4]. What looks like the comeback of dumb terminals on the first glance is in fact a new way of exploiting the advantages of client-server applications without the disadvantages formerly experienced.

If we consider the combination of Java and relational databases in addition to the client-server considerations, we also find ourselves confronted with some well-known persistence tradeoffs. [⇒ODI] recalls a short definition of persistence:

> "The lifetime of a persistent object exceeds the life of the application process that created it. Persistent object management is the mechanism for storing the state of objects in a non-volatile place so that when the application shuts down, the objects continue to exist"

Since object orientation has started to be widely used to implement software, the question of how to deal with the design discontinuity between object oriented design and semantic data design has always been present. We therefore have to differentiate between different persistence approaches. Figure 3.2 gives a short overview of the different persistence approaches with respect to the usual object layers: user interface objects, controller objects, business objects, database access objects and the relational database itself (the broken line marks the design break between the object oriented world and the relational database world):

1. Object serialization (section 3.2).

[3] as long as a Java capable browser is used on the client side.

[4] Currently, the question "PC or NC?" is dominated by marketing considerations rather than technical requirements [Bor98].

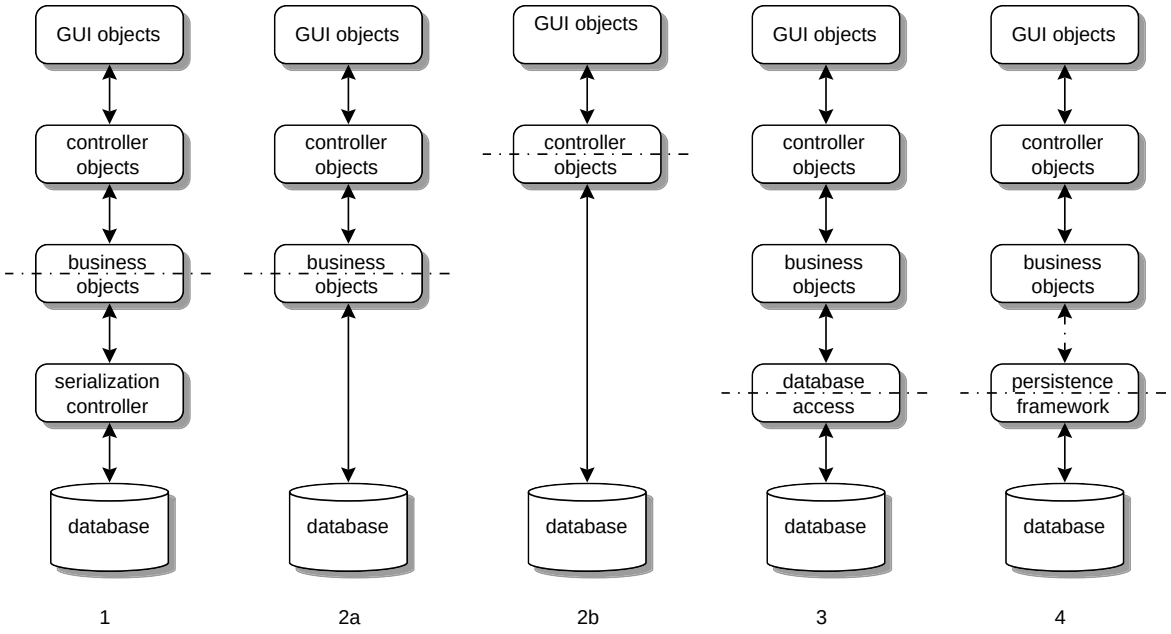


Figure 3.2: Different persistence approaches.

2. Each business object connects directly to the database for its own purposes (section 3.3). A special case (2b) is typical for an informational system with almost no middleware.

3. A better approach is to encapsulate the database access within some database access controller objects (section 3.4).

4. For very large and complex projects a persistence framework is the best solution (section 3.5).

The greatest problem common to all these approaches is the structural discontinuity between the objects' relations in the class diagram and the entities' relations in the ER diagram. Moreover, for more efficiency it might even be necessary to implement some parts with procedural SQL on the server side, introducing a further design break. At least this deficiency will be overcome, when the Java virtual machine is integrated into the database kernels of the object relational successors of the relational databases currently available.

The different persistence approaches are described in more detail in the following sections, including reasonable client-server architectures for each model.

3.2 Object Serialization

One of the easiest ways to achieve a basic form of object persistence is already 'built-in' with Java – object serialization [⇒Java]:

> "Object Serialization supports the encoding of objects, and the objects reachable from them, into a stream of bytes; and it supports the complementary reconstruction of the object graph from the stream. Serialization is used for lightweight persistence and for communication via sockets or Remote Method Invocation (RMI). [...] A class may implement its own external encoding and is then solely responsible for the external format. [...] The key to storing and retrieving objects in a serialized form is representing the state of objects sufficient to reconstruct the object(s). Objects to be saved in the stream may support either the Serializable or the Externalizable interface. For Java(tm) objects, the serialized form must be able to identify and verify the Java(tm) class from which the contents of the object were saved and to restore the contents to a new instance. For serializable objects, the stream includes sufficient information to restore the fields in the stream to a compatible version of the class. For Externalizable objects, the class is solely responsible for the external format of its contents.
>
> Objects to be stored and retrieved frequently refer to other objects. Those other objects must be stored and retrieved at the same time to maintain the relationships between the objects. When an object is stored, all of the objects that are reachable from that object are stored as well."

Designed to transfer objects as streams over the network or to store them in the file system, it can also be used to store them in a database. The database has no structure in this case and simply serves as a repository for serialized objects. The disadvantage is that if we want to search a particular object by a key, we first have to re-instantiate all the objects from the database. Moreover, when an object is serialized or instantiated, all the objects it refers to are too. Serialization has to read or write entire graphs of objects at a time. This is known as the 'banana-gorilla' problem: The instantiation of the small banana causes the instantiation of the large gorilla directly or indirectly referenced by the banana. Serialization can be sufficient for applications that operate on small amounts of data, are not frequently updating these objects and where reliable storage is not an absolute requirement.

However, serialization is not optimal ([⇒ODI]) for applications that

- manage hundreds of megabytes[5] of persistent objects: storing and retrieving the entire graph will be too slow.
- are frequently updating those objects: Again, always the entire graph has to be written, even if just one attribute of one single object has changed.
- have to ensure that the changes are reliably saved: If the system crashes during serialization, the contents of the file are lost.

[5]For relational databases this would still be considered a small amount of data. The world's largest databases today already exceed some tens of terabytes of information [ES97]. For object oriented databases, however, 100MB are already considered medium or even large.

In contrast to serializable objects, in the case of externalizable objects the class is solely responsible for the external format of its contents. This allows us, however, to implement our own method of serialization. A reasonable approach is to use the implementation of the externalizable interface to generate a SQL insert or select statement, which is then forwarded to an open JDBC stream. With this method one object can be stored as one entity in the database, key and persistent information can be stored in table attributes instead of just one character stream and the restoring of the objects can make use of these keys. Moreover, object references can be emulated in the database with foreign keys to other serialized objects. Secrecy is violated because the object has to know that it is persistent and has to implement this on its own. At least reasonable encapsulation is possible.

3.3 PHCs go Java

The easiest possibility for dealing with the design discontinuity between object orientation and semantic data design is to simply ignore it. This means that each object directly connects to the database for its own purposes but not just to store or retrieve its *own* state. This approach may still be useful for smaller projects but it introduces a strong mutual dependency between the whole database and all classes, making maintenance difficult and enhancements prone to error.

This method can be used successfully, however, if the object oriented part is almost reduced to the user interface (thin client). This is true for information systems, whose main task is retrieving, presenting and eventually updating relational database information with simple application functionality (middleware).

In this area PHCs implemented as Java objects provide a promising alternative to the pure HTML interface. Figure 3.3 shows a comparison between object oriented design and implementation, PHC design and PHC implementation. Whereas PHC implementation is finally done procedurally on the server side, the PHC design layers are related to the object oriented layers (marked with dotted lines). This similarity encourages the Java implementation of PHCs[6].

To integrate a Java GUI (Graphical User Interface) instead of a pure HTML interface with the same PHC/DL description, a runtime environment library provides a framework with controller classes implementing the connectivity between the user interface classes and the generated PHC classes, the manipulation of the runtime repository and the database connection via JDBC (figure 3.4). All the functional constructions of PHC/DL can easily be translated into Java source code, including the functional methods and the user elements with the methods 'print', 'image' and 'click'. Instead of a PHC/LL description the Java applet implementing the GUI is directly programmed using Java source code. A runtime repository – much smaller as in the case of pure HTML – is also generated, containing images, dictionary, user management, rights and security.

[6]The name '*pure HTML* control' looses its context in this case but it is still used for convenience. If PHCs are mentioned in combination with Java, *functionality and structure* are meant rather than the pure HTML characteristic.

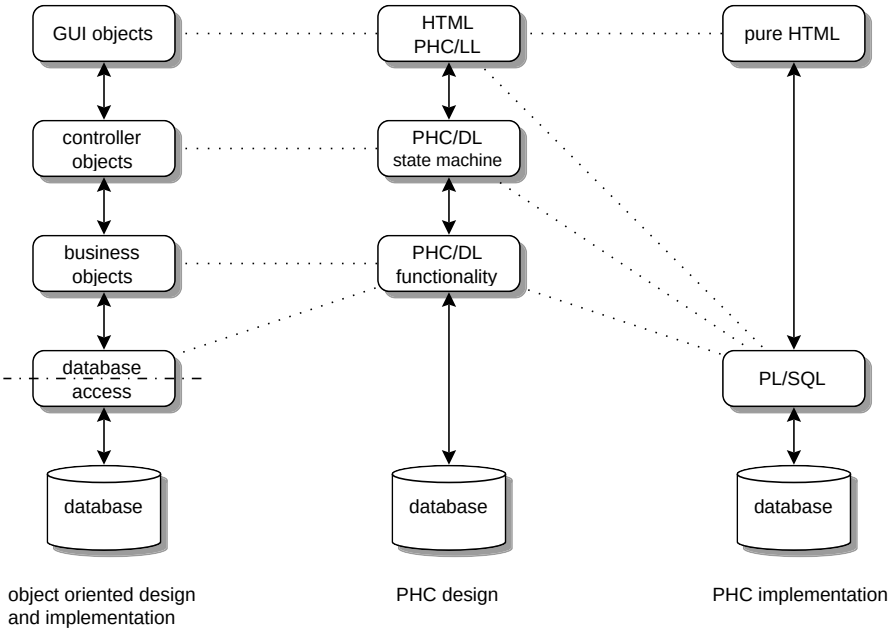


Figure 3.3: Similarities between object oriented design and PHC design.

At the beginning of a new session the applet first has to be loaded from the Web server via HTTP. The applet then opens a direct JDBC connection to the database and downloads the Java runtime repository. The session state information is implicitly stored as the PHC objects' internal states. During the session the GUI objects send messages to the PHC objects via controller objects. The PHC objects eventually change their internal state, call a method or execute a predefined SQL statement and finally send a message back to the controller.

The drawback is that the Java objects are reduced to the limitations of PHCs. However, the advantage is that they can be derived from the same design and hence implement the same functionality as the pure HTML interface.

In principle, there are three different approaches to a Java user interface with PHCs:

1. One Java GUI applet for the complete application.
2. Small Java GUI applets embedded into HTML, one for each PHC.
3. Java GUI applets and HTML PHCs mixed, even on the same page[7]. PHC applets can be used for recurrent tasks, whereas e. g. the results of the queries are displayed with HTML.

Despite the longer download time at the beginning, the first approach has proven to be the most convenient, especially from design and implementation but also from user acceptance: The mixed approaches tend to be confusing to the end user.

[7]In this case some JavaScript is needed to establish interaction between HTML and Java.

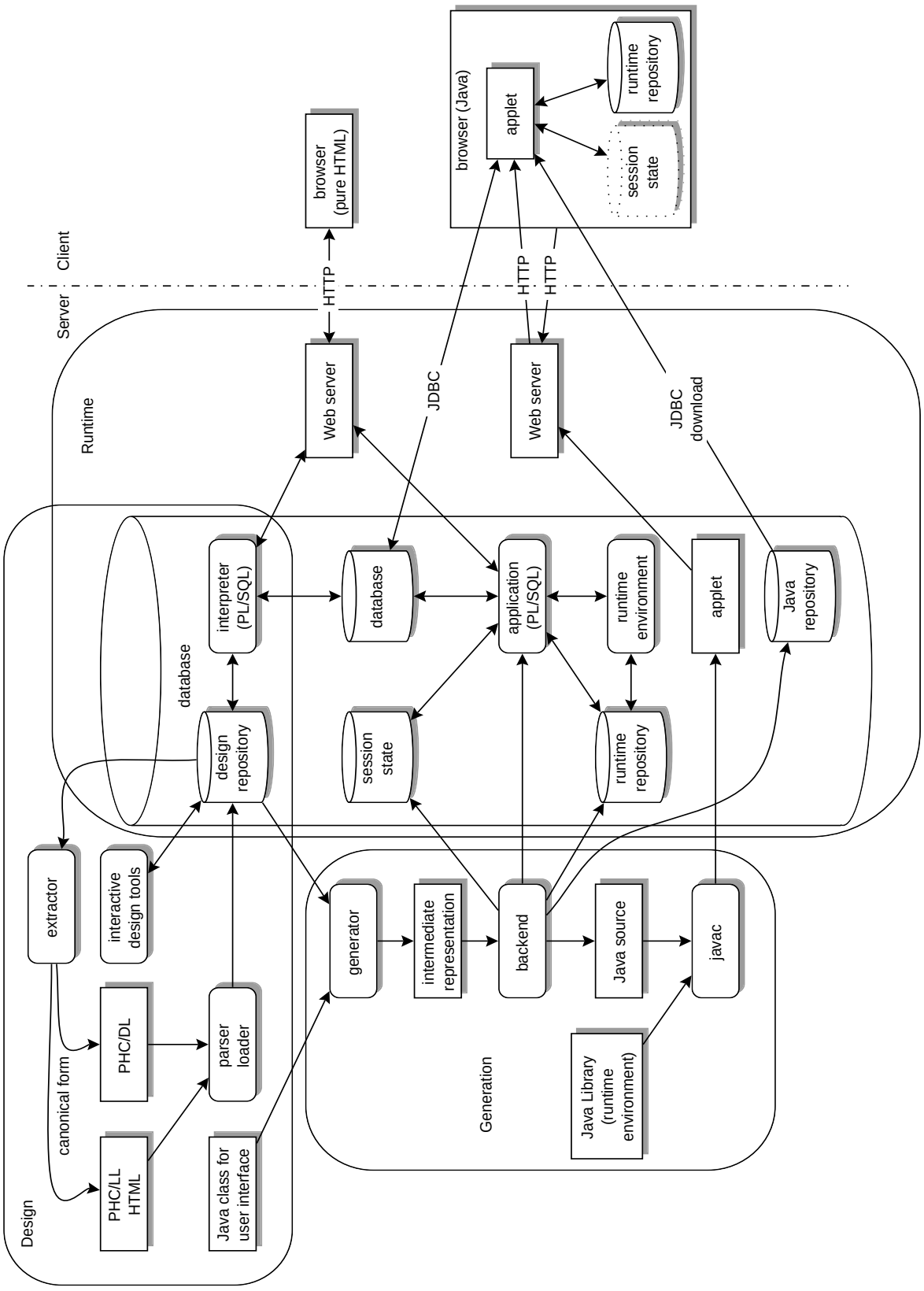


Figure 3.4: Design and implementation with alternative Java applet interface.

As with pure HTML there are two possibilities for the runtime environment: generation and interpretation. With interpretation a generic class from the framework has to interpret the runtime repository now containing the state machine and the output definitions. This makes the applet smaller but the repository larger. However, interpreting a repository with an interpreted language gives the impression of intentionally slowing things down.

3.4 Encapsulated Database Access

Another approach, which is better for applications with a thicker middleware, is to encapsulate the database access within some controller objects thus focusing the mutual dependency between classes and database tables on these few objects. Each of these objects is now responsible for a well-defined part of the database and set of classes. These controller objects access the database and instantiate objects with data from the database. On the other hand, these controllers have the responsibility of reading the objects' data and writing it back to the database. The other objects then know nothing about persistence and can hence be designed with a pure object oriented design method. However, the database design still has to be done concurrently using classic semantic methods and transaction control has to be handled by the controller objects.

This approach has been used to implement a prototype variation of the pure HTML application. Java frames (appearing as windows on the user interface) are used instead of the HTML frames as shown in figure 3.5. Whereas the appearance is very similar to the pure HTML solution, this implementation is different to the Java implementation of PHCs described in the previous section because the application functionality has been designed using object oriented design methods and UML [⇒OMG] instead of PHC/DL.

Figure 3.6 shows the roadmap through the object oriented design process as described in [Ove97] originating from "The Rational Objectory Process" [⇒Rational]. Firstly the goals and actors were identified. As is typical for information systems, most of the goals were searching, retrieving or manipulating information from the database. Afterwards the use cases have been defined and a first class diagram. The sequence diagrams where used to refine the use cases, resulting in some modifications and refinement of the class diagram. A specification was derived from the final version of the class diagram and the sequence diagrams. Some more complex objects were further specified with state diagrams. Finally the database access controllers were added before the implementation with Java took place. The detailed design cycle and the implementation are described in more detail in [Rad98].

The following practical experiences could be derived from this implementation:

- As is usual with object orientation, the modelling was very time consuming but the final implementation was easy and straight-forward.
- It turned out that container objects [Boo94] holding search structures for the keys and references to the extension of one or more classes are crucial for efficient handling of larger sets of objects, especially for searching.

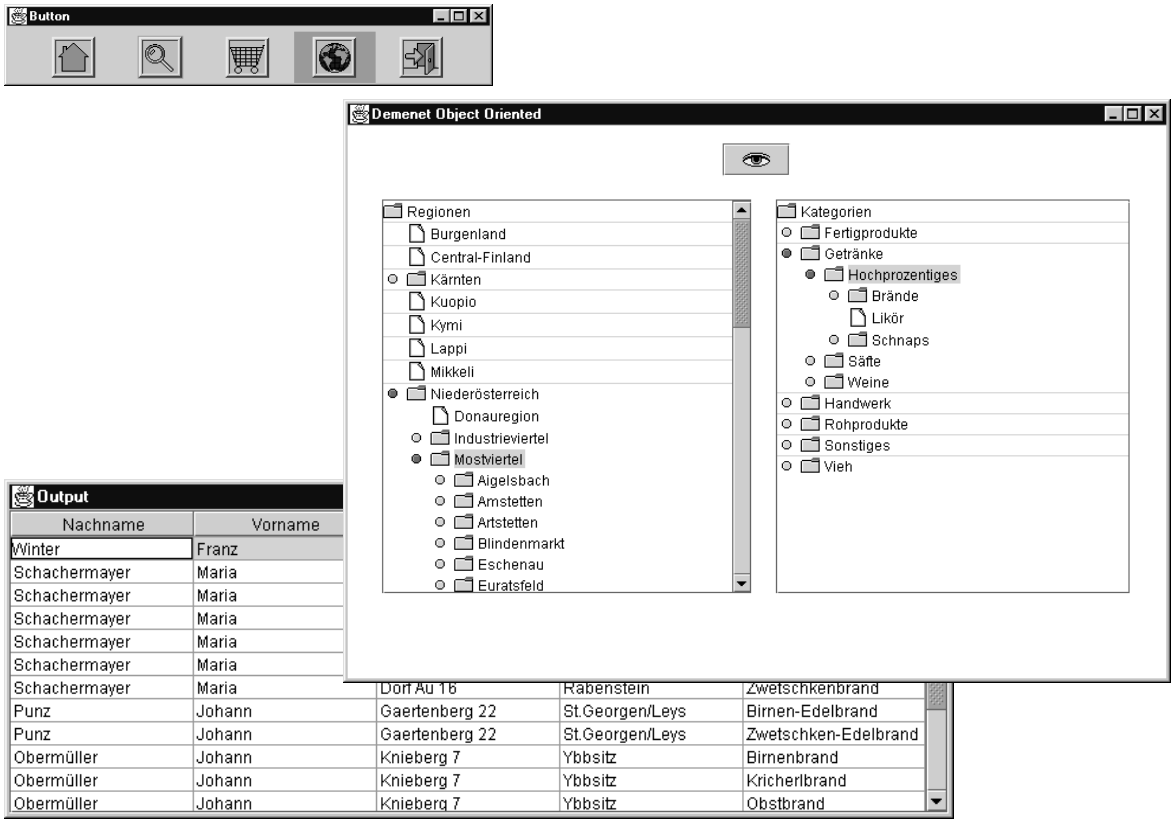


Figure 3.5: Geographical Selection with Java.

- Lazy instantiation of objects saves resources on the client side. Often the container object can be enhanced to hold one or two attributes of a complete database table. This often prevents the application from instantiating all the objects of a class for the mere purpose of user interface representation of one or two attributes. The container holds enough information for the user interface objects to enable a user to browse and select items. Only the really necessary objects are instantiated afterwards. Hence, the container objects are a special form of database access controllers.

- All database access controllers together encapsulate the relational database access. In typical information systems this encapsulation provides sufficient secrecy between the object oriented world and the database world.

- In this case the relational database was already available from the pure HTML approach but the relational database would normally have to be designed concurrently using classic semantic data design methods, e. g. entity relationship diagrams. Methods are available to translate an object model into an entity relationship model and vice-versa using meta-models [BPS94].

To further enhance efficiency with an intelligent preloading mechanism, different classification methods of data resulting from database queries are firstly introduced:

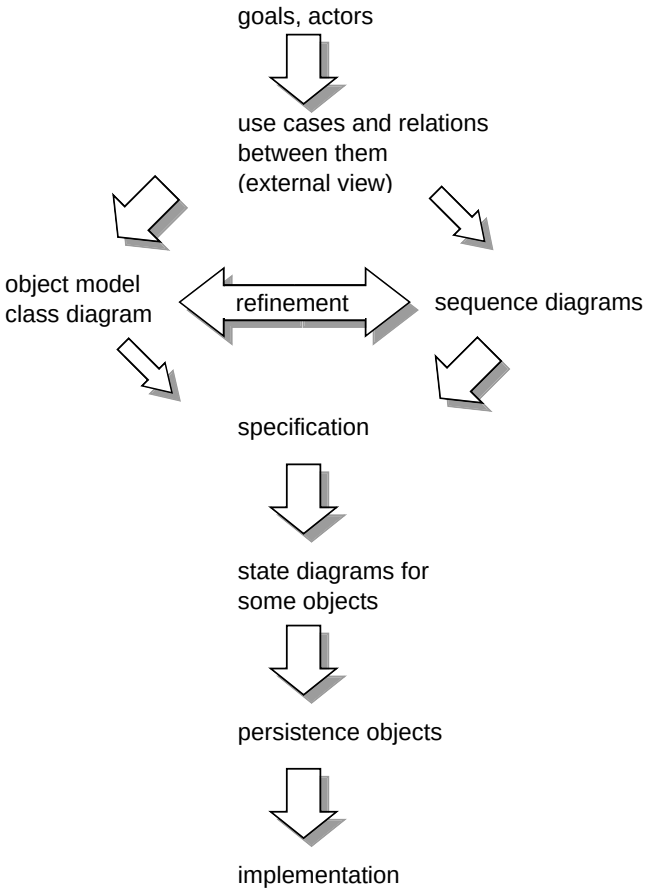


Figure 3.6: Roadmap through the Rational Objectory Design process.

Reuse Data typically often reused (like the geographical selection in previous examples) versus data needed only once (like a particular search result).

User Interface Complete object data needed for interaction of business objects versus partial object data merely used for user interface presentation.

Size Large data versus small data, required to compute a particular result.

Using these classification criteria, a preload mechanism can further reduce the average response time of the user interface: Initially, the applet loads small data often typically used for browsing and selecting of information. During runtime a branch prediction task using probability observes the user interface actions and makes a prediction of what data will be needed next. Divided by the size of data to be preloaded, a first assessment is made to determine whether it is worthwile to preload this data. Furthermore, it is determined whether this data will be needed more often for other results or just once for one result. The expectations for the frequency of usage of this data is multiplied by the former value. This value is finally compared to a threshold, to determine if the data has to be preloaded or not. All data which is not preloaded can, of course, be loaded on demand when necessary (like a cache miss in hardware caches).

Starting from an algorithmic (causal) definition of branch probability and reuse expectation value, an intelligent mechanism could *learn* during runtime and modify these values. This is a typical task for software agents (see chapter 4). Sending these values back to the server before termination of the applet could result in a long-term improvement of this preload method according to the average user.

Usually the applet caching does not introduce the same problems as for instance with multiprocessor hardware caching, because in the case of typical information systems it does not matter whether the underlying information changes slightly during browsing and selecting of information or not. However, the application must at least be resilient to such changes and should not result in behaviour which is confusing for the end user.

If an application does require exact caching methods the publisher subscriber pattern [Joh95] can be used to implement a bus snooping protocol similar to hardware caches. Database triggers (active databases) help to implement such a system.

3.5 Persistence Frameworks

For large projects a persistence framework can solve the problem. Persistence frameworks try to achieve the best possible secrecy[8] between the business objects and the database: The idea is that the business objects can be designed with (almost) no knowledge of persistence. Some special methods are introduced to instantiate an object from the database or to commit a transaction. The database access is now encapsulated within the framework.

The framework described in [Vio96] has been partly implemented with the same prototype implementation used for the pure HTML approach. The key features are mirror objects that accompany the business objects and handle the database transactions without the knowledge of the business objects themselves. The following experiences have been made:

- The implementation is more extensive, because for each business class a mirror class has to be implemented. However, this was easily done following the design rules given in [Vio96].
- Changing the database structure requires a reimplementation of the database objects but leaves the business objects unchanged. A redesign of the business objects may also result in a reimplementation of the database objects but leaves the database unchanged. The encapsulation therefore works.
- Difficulties arise with database queries resulting in more than one tuple. The framework provides no support for such a situation, it has to be implemented manually within the database objects' methods.

[8] Secrecy (a means of encapsulation) is one of the most important features in achieving the advantages of object orientation.

- Due to the strict secrecy, it is not possible to just select a list of table attributes for presentation purposes and forward it to the user interface. The complete set of objects always has to be instantiated resulting in far more objects than the approach in section 3.4.
- This is, on the other hand, an advantage, because the application can be designed with pure object oriented design methods without being disturbed by persistence consideration. Persistence is first introduced during the implementation of the database objects.
- For Web based applications it is useful to leave the database objects on the server side (more efficient for multiple-tuple-operations) while the business objects may reside on either the client or server side. A thick client may take some load off the network and the server.
- As already mentioned, Java implementation (application or servlets) on the server side tends to be slow with regards to database connection. The integration of the Java Virtual Machine (JVM) into the database kernels will provide better performance and eventually replace the current procedural database languages.

Java Relational Binding [Gre98, ⇒O2] is another middleware product especially designed for Java applications: From the description of a set of classes a relational schema and methods of reading and writing objects in the database are generated. This eliminates the design discontinuity as all the relational parts are generated by the tools. On the other hand, this leaves no possibility for applying this method to already existing schemas of legacy systems. An object cache is used to improve the otherwise poor performance of object relational mappings. This lack in performance is also addressed by [⇒Persistence]. The key ideas are to optimize business object mapping and to perform object cache management.

3.6 Orthogonal Persistence and Object Oriented Databases

For object oriented applications the best approach – at least with respect to design – is orthogonal persistence in object oriented databases. Literature about object oriented databases [Heu97, SST97, Sch97, LV96] differentiates between three main streams:

1. Enhancement of object oriented programming languages (GemStone [⇒GemStone], ObjectStore [⇒ODI], POET [⇒POET] and others).
2. Enhancement of relational database systems (Postgres [⇒Postgres], Informix [⇒Informix] and others)
3. Completely new (O_2 [⇒O2], ITASCA [⇒IBEX], Jasmine (with promising multimedia features) [KDM98, HFPH98] and others).

Another possibility to differentiate between different object oriented databases is by the support features for multi media content [Die98]. In order to standardize object oriented databases and to bring these different approaches together, the Object Data Management Group has released the ODMG 2.0 standard 1997 [⇒ODMG, Cat97a]. The Object Query Language (OQL) standard also emerges [Eva98].

Client-server distribution of object oriented applications can be done in various ways. With a pure Java solution the communication between client and server can be implemented with a proprietary protocol, or – more efficiently – with Remote Method Invocation RMI [Mer97, ⇒Java], or by using the Common Object Request Broker Architecture CORBA [Say97, ⇒OMG]. In both cases objects send messages to remote objects as if they were local. RMI is tailored towards Java, as it is smaller and more efficient, while there are CORBA IDL (Interface Description Language) mappings available for Java and many other object oriented programming languages. Hence RMI is the method chosen in a pure Java environment while CORBA is the solution for heterogeneous environments.

If a thick client is used (business objects on the client side, at least partially), a check-out/check-in mechanism has to be used in order to check objects out of the database for a long-lasting transaction and check them in again later. This is typical for object oriented databases.

For Java the most interesting approach in this area is the PJama project [⇒PJama] at the University of Glasgow:

> "PJama is an experimental persistent programming system for the Java programming language, that embodies the notion of orthogonal persistence: an approach to making application objects persist between program executions with the minimum possible effort required from the application programs themselves.
>
> PJama supports the programming environment for Java being developed by the Forest [⇒SunLabs] Project. This project takes the position that the persistence and tool integration mechanisms of traditional operating systems are a major limiting factor in the development of powerful, multi-user software development environments. The intent is to replace the use of file systems and ad hoc persistence mechanisms with typed, persistent objects."

This is the most promising approach for object oriented pure Java applications.

3.7 Comparison

Figure 3.7 concludes this chapter with a comparison of the reasonable combinations of client-server and persistence tradeoffs from the previous sections. The main decision criteria is whether an information system or a processing system has to be designed[9]:

[9]This is not an exact definition. There are systems which fall into both categories and there are design methods suitable for both categories. However, it has proven to be useful as a rough differenti-

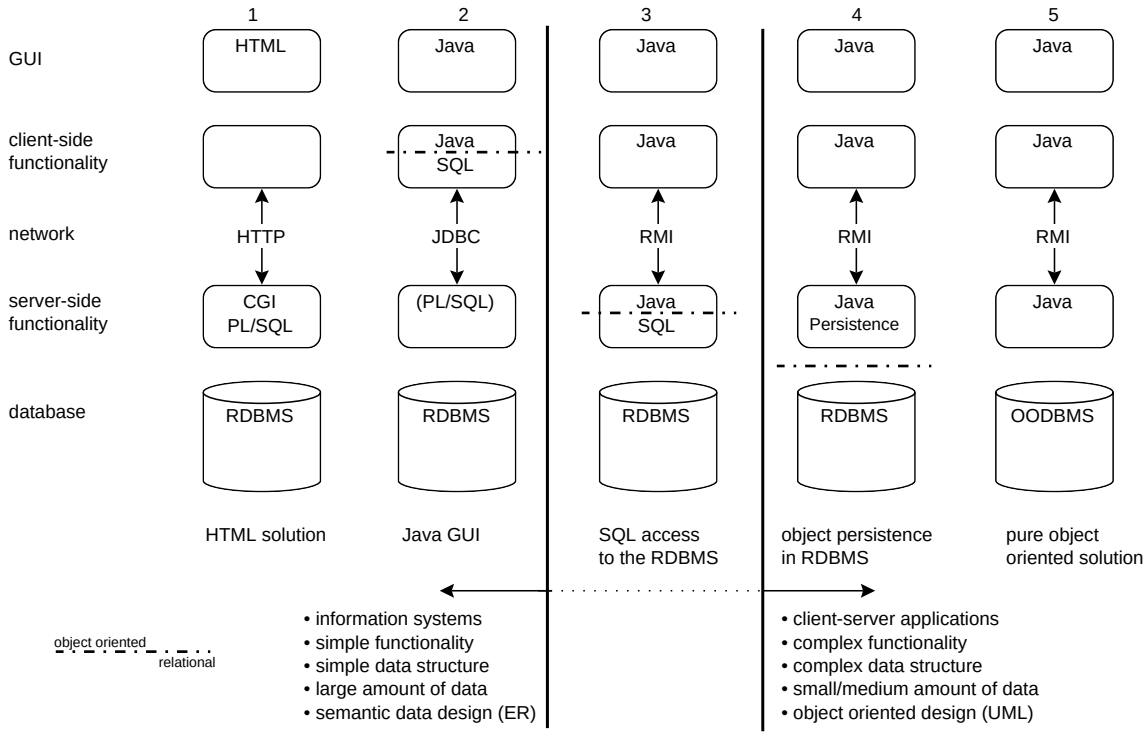


Figure 3.7: Comparison of the most reasonable approaches.

processing system: Client-server applications with typically complex structured data, complex functionality but usually small to medium amount of data (CAD systems, client-server games, etc.). Also, the server can play an active role and trigger events on the client side (which is not possible in number 1 and 2). Object oriented techniques and UML (Unified Modeling Language) can be used to design these systems.

information system: Large databases of simply to complex structured data but with only simple functionality for finding, retrieving and editing the stored data.

The techniques numbers 3, 4 and 5 show different structures of processing systems. They differ in their database integration: In number 3 controller objects access the database directly via SQL, which means a discontinuity in the design between the application and the database. The controllers provide a means of separation between database and business objects but they can also establish interaction between GUI and DB without business objects inbetween for better performance. This technique can be used for both small processing systems and information systems with a thicker middleware. One aspect of when to choose this approach to design an information system might be that a connection to another object oriented application is required via RMI or CORBA.

In number 4 a relational database is used to make the objects persistent through a special framework, hence the application design is homogenous object oriented. These

ation criteria. More recently the term 'Web–based Information System' (WIS) has been introduced in literature [IBV98].

approaches have their roots in a period of time where object oriented programming and design were already widely used but object oriented databases were in their early research stages. For new applications that do not have to connect to legacy database systems, object oriented databases nowadays provide a better possibility for orthogonal persistence. For pure informational systems, on the other hand, the larger design effort is often not worthwhile. In number 5, finally, a pure object oriented approach achieves orthogonal persistence through an object oriented database.

Of course, HTML is not an option for object oriented client-server applications. But this thesis has its focus on information systems. Numbers 1, 2 and 3 show different structures of information systems. Numbers 1 and 2 are passive systems, where the server cannot trigger events on the client side[10]. Number 2 is the Java alternative of the pure HTML design: Each object has its own database connection – there is no secrecy between database and objects.

In numbers 2 and 3 Java is used for more 'intelligence' on the client side to take some load off the server and the network, thus making Java applets more efficient for long lasting transactions. Moreover, there is additional functionality, especially for the user interface. Despite its indisputable benefits, there are some drawbacks: to implement servlets (number 3) efficiently, the virtual machine must be integrated with the database kernel. Only a few database companies have announced this so far. Even more importantly, an applet must be downloaded to the client first. This is no problem for intranets. However, the internet has a smaller bandwidth, especially in Europe. For typical Internet applications consisting of just a few client-server interactions, the long download–time at the beginning is often annoying.

A quantitative analysis in section 5.3 compares the Java and pure HTML approaches from 1 and 2 with each other based on the use of typical end user profiles observed during the pilot phase of our real–life applications. A unique design model for both techniques (number 1 and 2) gives us the possibility to offer both user interfaces with the same functionality leaving the last choice to the user. A qualitative analysis to compare the Java and the pure HTML approach with each other based on typical end-user profiles observed during the pilot phase of the prototype implementations gives initial results:

User interface: With pure HTML the user interface lacks the possibility of constraints and event handling: An HTTP session has to be performed before functional dependencies or restrictions can affect other inputs. To achieve this HTTP session, a submit button has to be pressed explicitly. It does not usually suffice to fill an input field or make a selection. Thus the user interface cannot react directly on user input. JavaScript on the other hand provides one possibility of implementing constraints and event handling into a HTML user interface: As far as JavaScript becomes standardized [⇒JS, ⇒ECMA] and widely implemented this disadvantage disappears. On the other hand HTML has the advantage over Java of easy layout and page design, eg. with style sheets. Recently, this advantage has shrunk: The Java swing classes – part of the Java Foundation Classes (JFC) contained in the

[10]Netscape's proprietary channels provide a possibility for active servers with pure HTML. It is, however, not encouraged to use such proprietary solutions in general design methods

Java Development Kit (JDK) 1.2 – provide the possibility of a rich user interface for the first time. However, the design of an HTML page remains much easier.

Performance: With pure HTML, each user interaction results in an HTTP interaction with the server, which in turn has to deliver a whole HTML page. Less client-server connections are necessary with Java because menues and other UI elements (like hierarchical selections) can be downloaded at once and then kept on the client side (but we have to take care about database updates in the meantime). In addition, less data has to be transmitted with each connection because only the data has to be transmitted without the HTML code. Unfortunately the Java applet has to be downloaded first, which can be annoying on slow network connections. We have observed that with our pilot implementation the download of the Java applet is not worthwhile (in terms of performance) if less than 200-300 user interactions take place (see section 5.3). In our pilots most user session have fewer interactions which is why we keep the pure HTML interface available: It is obviously faster than the Java version which also requires more computing power on the client side.

Server load: With pure HTML more capacity is required on the server side: the size of the dynamic session state information – corresponding with the number of concurrent sessions – adds to the database's size. The larger the database compared to the session state information the less this matters.

Browser compatibility: can easily be achieved with pure HTML. With Java it has to be tested carefully but is still possible - at least as long as the pure HTML alternative is available for older browsers.

Design methods: In both cases the functionality is designed with PHC/DL thus restricting the Java variant to a subset of its possibilities to be compatible with the pure HTML approach. PHC/LL and the Java GUI have to be designed separately based on the same PHC/DL. The user interface layout can easier be designed with HTML by designing static pages first and then enhancing them to PHC/LL.Without our design tools the direct implementation of the PL/SQL code is error prone and maintenance is difficult: The direct implementation of PL/SQL without any tools is therefore only recommended for medium sized application.

Multi media integration: Images and other Multimedia content can be integrated into both approaches: The advantage of Java is: There is no need for a proprietary installation of PlugIns. Moreover Java objects are available for playing almost any multimedia format available on the Internet.

File upload: The upload of files from the local file system to the server is not possible with Java due to the restrictive security model. A workaround is to build a combination of a server side Java listener-application and CGI perl-scripts to integrate the usage of a small HTML frame with a form upload into the Java applet in an almost homogenous way.

Server events and collaboration: For collaboration it is necessary that the server can trigger an event on the client side. With HTML Netscape has developed some

extensions (server push, channels) to implement active servers, but these are proprietary and not widely used. On the other hand, using Java servlets (number 3 in figure 3.7) an active server can be implemented (remote method invocation over sockets without HTTP).

Database transactions: With pure HTML (even with cookies) the connection to the database is stateless, resulting in complex state handling and timeout mechanisms which also compromise the security of such a system. With JDBC or RMI connection orientation can easily be achieved, thus guaranteeing safe and secure transactions. Performance considerations can be found in [Dic97].

Chapter 4

Searching in Database Backed Web Applications

As different as they are in design and implementation, relational databases and the Web also differ considerably in the ways they can be searched. This chapter firstly gives an overview of the current status on searching in these two worlds and then introduces several approaches for combining these methods to provide efficient search capabilities for database backed Web applications.

4.1 Search on the Web and in Relational Databases

In talking about the Web we cannot ignore the question of how to search *and find* relevant information. On the Web itself one can find a lot of research dealing with this question. Recent work and a good overview can be found in [⇒SoS]. The research on how to find relevant information in hypertext systems is in fact far older than the Web. Recent work on information retrieval can be found in [⇒IBIS].

On the other hand, there is also a long history of research on how to retrieve information from relational databases. The classical built-in method are, of course, relational query languages [Ull88] like SQL (Structured Query Language) or QBE (Query By Example). All the major commercially available relational database products implement these, or similar, query languages. However, the disadvantage with this is that one has to know the *exact* names of the tables and the attributes, the exact relational topology (e. g. foreign key structure) and the *semantic* meaning of this structure (e. g. entity relationship diagram) to formulate a useful query to the database.

An interesting approach to overcome *some* of this deficiencies are 'Universal Relation' query languages, a short citation from [Ull89], p.1026 explains the key ideas:

> "A *universal relation* is an imaginary relation that represents all of the data in the database. A query language that lets us refer to the universal relation, rather than to the actual database scheme, can be much simpler than typical

> relational query languages, because we need to mention only attributes, rather than attribute-relation pairs. As an especially important example, a natural-language interface to a database is designed for use by people who understand little or nothing of the scheme. All natural-language interfaces efficiently refer to the universal relation, and queries about the universal relation must be translated by the system into queries about the existing scheme. Interpreting queries over a universal relation is an admittedly difficult task. Yet it is one that must be performed if we are to have natural language interfaces..."

One remaining problem is that of naming: During design great attention has to be paid to the naming of attributes: Attributes representing the same thing in different relation schemes have to be given the same name, attributes representing different things have to be given different names. This guarantees globally unambigous attribute names.

The supporters and opponents of universal relations fight a constant battle. Some of the arguments against universal relations are:

- The universal relation does not always result in a reasonably interpretable representation of the data.
- The natural join is not always the best solution.
- If the data structure is cyclic[1], which path should be taken? Normally the shortest path is selected, leaving the question open as to whether the user's intuition matches that of the system's designer. Hence some systems provide an interactive method, leaving the final choice of the correct path to the user.

To cite [Ull89],p.1030 once again: "However, the real counterargument is that *there ain't nothing better.*" However, as will be shown later, universal relations are not the best solution for searching in database backed Web applications, but the idea of a path through the database will be used for the approach proposed here.

Whereas universal relations never got further than university research, the first natural language interfaces (NLIs) for databases for the English speaking community became commercially available about 18 years ago. Some of them use universal relations, others directly translate a subset of a natural language into SQL statements using similar techniques. In the meanwhile NLIs for other languages (including German) have followed. Common requirements of NLIs include [TB90]:

- No formal training required to use these systems.
- The acceptable language subset must be large enough to allow for natural communication.
- Handling: Recognition of proper names, automatic spelling correction, efficient starting phrase skipper and an abbreviation and pattern recognizer. Hence the system needs at least a basic vocabulary of words, patterns and phrases.

[1]This is the most common case: Almost no real-life design is acyclic.

- Accessibility: For every possible formal SQL query there exists at least one natural language query.
- Habitability: User is able to judge which natural language utterances are acceptable for the system.
- Portability: With respect to different hardware, operating systems, database management systems, domain knowledge, data models and connectivity to other software components.

On the Web – and hence also for database backed Web applications – only two methods of querying are realistic – keywords and NLIs (nobody would expect the average Web user first to learn the scheme of the underlying database and then to formulate a proper query directly with SQL.):

Keywords are the usual method of searching on the Web's search engines. Usually the boolean operators `AND`, `OR` and `NOT` can be used to combine keywords. Also inclusion/exclusion methods are usual, where some keywords are marked as required and others as not allowed. Interactive iterative refinement methods further improve this sort of querying by including or excluding associated keywords.

Natural Language is rarely seen on the Web nowadays, but would certainly be appreciated by the average Web user.

4.2 A Natural Language Interface for Database Backed Web Applications

This section introduces a heuristic straight-forward approach, usable with both keyword search and natural language query. The key idea is to differentiate between the structural and contextual meaning of keywords. The structural keywords are used to select a path within the database topology (i. e. a particular attribute of a table or a whole SQL statement). A full-text search then detects the occurance of the contents keywords *within* the preselected path. Similar ideas – called 'keyword separation' – can also be found to search static hypertext systems [Zen98]. Other papers suggest rich links [Oin98] or metalevel links [Tak98] for better navigation and queries. Yet compared with linked hypertext structures, the structure of a relational database (and especially the semantically richer ER description) contains even more information. With natural language utterances we use a starting phrase skipper, but then use the same simple method as with keywords.

The base is a configuration repository (stored as relational data structure) which defines a semantic metadata model of the database: The database dictionary is the basis for the metadata model. In addition to the dictionary it defines which tables, attributes and relations are visible under what names (implicit renaming) to the search interface. A list of uninteresting 'filling' words and starting phrases is used to eliminate information not

used as keywords. The core of the model is a table structure, each entry consists of the following attributes:

- A list of structural keywords. These keywords are used to select a set of paths.
- A set of SQL statements (called 'paths' according to the definition for universal relations) which take the content keywords as parameters (marked with `$$`). There can be simple select statements from just one table or complete paths of joins through a part of the database.

```
SELECT aNachname, aBezeichnung
FROM tPerson, tAnbieter, tAnbot
WHERE (tPerson.aPersId = tAnbieter.aPersId)
AND (tAnbieter.aPersId = tAnbot.aPersId)
AND (tAnbot.aProdId = tProdukt.aProdId)
AND (tAnbot.aBezeichnung = $$)
```

  The usage of the SQL clauses `like` and `sounds` can be used to further improve the search results. The rules for output and anchor (see next two items) can use both the found value (`$$`) and the selected attributes from the SQL statement (e. g. `$aNachname`).

- A definition of how to format the output of the SQL query for the user: This includes the usage of more descriptive names than the names of the tables and attributes or the output of additional information from the appropriate path. This decouples the names used for design from the names used for querying and result output and hence, in contrast to universal relations, takes some pressure off the naming rules for design.
- A URL[2] to jump into the application, which takes the name and values of the found attributes as parameters. If complex session state information has to be built in order to reach an appropriate entry point, an additional function may be required within the database application.

Finally, for each table the following additional information is provided (for an unstructured search): Which attributes to use for content search and which attributes from those results to use for presentation purposes and link generation. A parameterized URL (or even a set of URLs) is provided to jump into the right entry point of the application.

The algorithm: First, the starting phrase and uninteresting 'filling' words are eliminated. Then the query input is searched for structural keywords. For *each keyword found*, the other keywords are treated as content keywords and the following steps are performed.

[2]There could also be the necessity for more than one URL. In this case, each result is provided with both links.

1. The appropriate path[3] is selected from the repository. The parametrized SQL statement is filled with the content attributes and executed as often as necessary.

2. The results are formatted according to the presentation rules of the respective path.

3. These results are links. The anchor tags are built from the URL definitions filled with the names and values of the attributes found.

4. A click on one of these results brings the user directly to a point within the database application that reflects the query results.

If no structural keyword is found, an unstructured search is performed: All keywords are treated as content keywords and are searched within all visible attributes. For each table, one or more URLs are defined as starting points of the application, which take the name and value of the found attribute as parameters. The main difference to universal relations is that there is no need to calculate or guess the right path: All paths are predefined and assigned to their appropriate keywords.

The repository is defined in cooperation with test users in a brainstorming process and can continously be refined during the operation of the system (e. g. from user feedback). This is no self-learning, solving-the-problems-of-the-world solution, but it has proven to be successful in the prototype implementations and it can be applied to any Web application based on relational databases by simply constructing an appropriate repository. It is the author's opinion that a decent organizational solution on hand is always better than a wonderful technical solution which is not yet available[4]. Moreover, in this case, the task of the search interface is to bring the user into the right starting point within the application rather than giving him a perfectly precise search result: The user can then – with the possibilities of the application itself – proceed further to the desired information. Hence it is only important to place the user *somewhere near* the desired results within the application.

At this point it also becomes clear why universal relations are not the best solution for searching in database backed Web applications: Even if the 'right' path can be found, how can the appropriate entry point of the application be determined? The query interpreter would have to know *how* the application works. Thus the idea of *paths* is used, but within the scope of a fixed configuration rather than a runtime interpretation.

Figure 4.1 shows a user request "I want to buy a flan"[5]. First the starting phrase "I want" and the filling words "a" and "to" are eliminated, resulting in two keywords "buy" and "flan". The word "buy" is now detected as a structural keyword. The respective statement is called with "flan" as its parameter and delivers some offers of flans from the database. Figure 4.2 shows the answer – the output has been prepared according to the definition from the repository: Each line refers to a result and the link provides the

[3]There might also be more than one path for a structural keyword. The algorithm is simply repeated for each and the results are combined.

[4]Which of course does *not* mean that it is not worth working *towards* the ideal solution...

[5]There is only a german implementation as yet. To be multilingual, one repository has to be set up for each language provided.

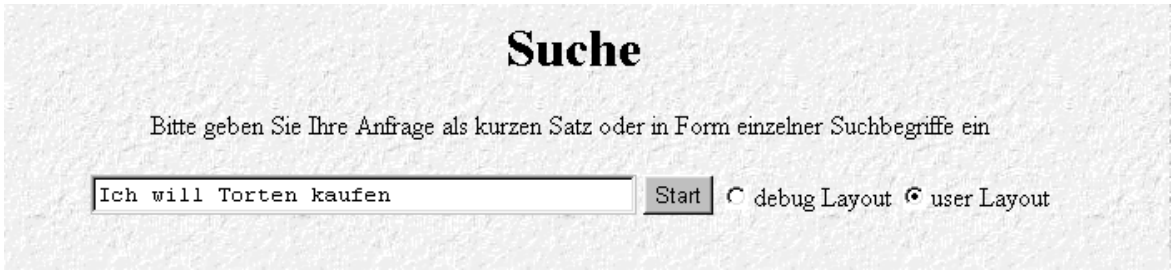


Figure 4.1: Sample query for the natural language interface.

Figure 4.2: Results from the sample query in figure 4.1.

possibility of directly entering the application at a point that reflects the query results. Figure 4.3 finally shows the entry point of the application: The product selection and the geographical selection are already opened according to the search result and the upper right-hand frame shows all the offers of "flans". The search sentences have to be short of course, otherwise many structural and content keywords would be combined with each other out of context in an almost random way.

If we consider a simple keyword search instead, e. g. "Torten Niederösterreich", which means we are searching for flans or for anything in Niederösterreich, no structural words could be found. Hence an unstructured search throughout all visible attributes of the database delivers the search result in figure 4.4. Now we can see different kinds of results, each of them with its own predefined, intuitive link into the database.

4.3 An Agent Interface for Database Backed Web Applications

One of the most common 'buzz' words of recent years is 'Agent'. Almost everything is named an agent and agents appear to be almost everywhere. Despite the fact that it is an emerging technology, the truth is that – following a more strict definition – there

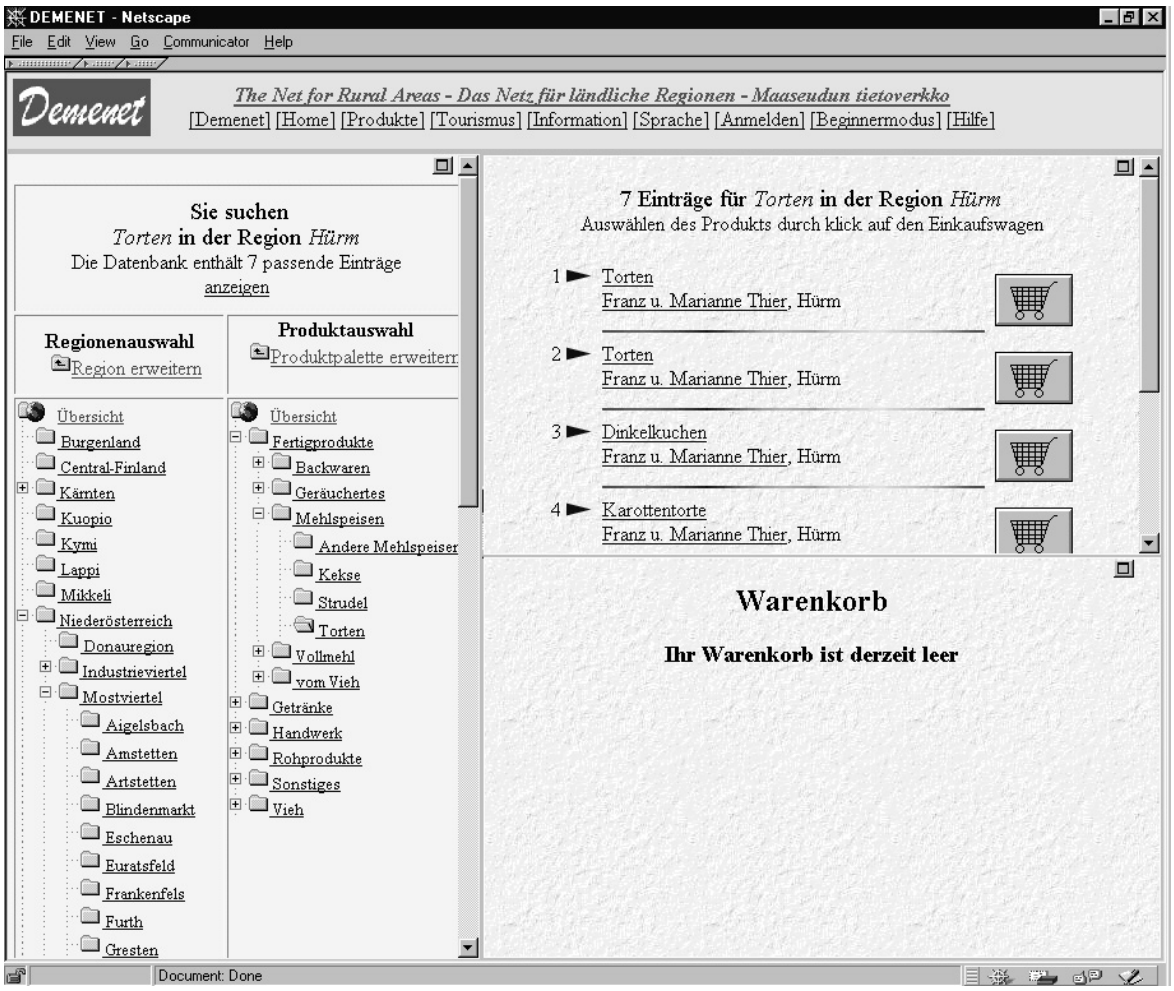


Figure 4.3: Destination point of search result link.

are not very many agents available. A recent overview is given in [KZ98] – important research groups are located at [⇒MIT, ⇒UMBC]. There are also some synonyms, sometimes even used with different meanings: 'Intelligent agents' is the most general term, an intelligent agent is not necessarily a piece of software (e. g. a smart battery systems, smart airbags). Software agents are software implementations of intelligent agents. A very general definition of software agents is provided in [Cho98]:

> "Agents are a new software paradigm. They are relatively small pieces of autonomous software that act in various roles on behalf of a specific function or user. [...] The agent may reside on the user's machine and be transmitted to distant computers where it carries out its functions. Or it may reside at network nodes. In the one as in the other case, the artifact will move autonomously. This [...] is one of the basic characteristics distinguishing an agent from other chunks of software.
>
> To its end user, or master, the agent is a personal assistant – his correspondent within the computer's communications and software landscape in which it works. It is a proactive artefact that can perform fairly sophisticated

Figure 4.4: Search result of unstructured keyword search.

tasks and can also exhibit learning abilities. The agent:

- Knows about the user, his or her wishes and preferred model of operation
- Is informed about other correspondent agents, including their profiles and work patterns
- Is able to collect, handle, and present information to its master's satisfaction
- Can structure system elements as required to tailor solutions to real-time users' needs

Should the agent be enriched with artificial intelligence (AI)? The majority of experts in this field say Yes!"

[Cho98] provides an extensive overview of state-of-the-art agent technology. A shorter definition comes from Prof. Pattie Maes who defines a software agent as "a process that lives in the world of computers and networks and that can operate autonomously to fulfill a set of tasks.". Ergo agents have a task and do not implement an algorithm. Their behaviour is often teleological, not causal and hence not predictable. And they have a sort of self-awareness [⇒MIT].

The key idea in providing an agent interface for a database backed Web application is to encapsulate the search functionality described in the previous section within an agent called 'database guardian'. In the first approach this piece of software is not an agent in the strict sense, because it is configured with the repository and has no learning capabilities. It should at least be able to talk to other agents, either with short messages,

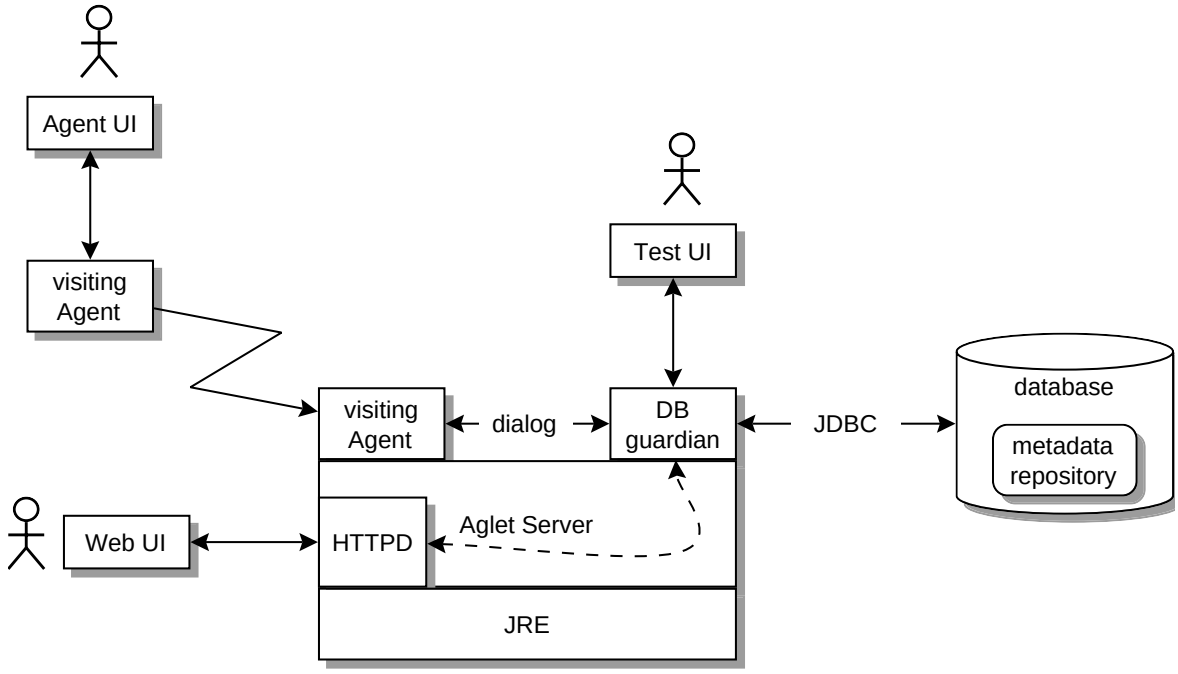


Figure 4.5: Agent environment for database backed Web applications.

or with an intelligent language like KQML (Knowledge Query and Manipulation Language) [Cho98, CH98, ⇒UMBC]. From the other agent's view this version of the natural language interface would look as if the guardian *knows* the database application and can hence communicate it to others. A personalized agent with a particular task can now migrate on its own through the Web and eventually find our database, communicate with our guardian and – in the case of success – report back home on what it has found.

Figure 4.5 shows an example of an agent environment using IBM's Aglets [⇒IBM]: The aglet server is a Java application and hence needs the Java Runtime Environment (JRE) for execution. The aglet server now provides a platform for aglets: One aglet is the database guardian which implements the search functionality. The other aglet might be a visiting agent, sent from somewhere, which has a particular question to ask our guardian. These two aglets can now communicate with each other. For testing purposes, a local user interface to the guardian is provided. In addition, the Aglet server provides a Web interface to the guardian, hence users without the possibility to send an agent can communicate directly with the guardian over the Web.

This approach provides the following advantages over classical client-server applications:

- The agent is self-migrating and might, therefore, eventually meet the database guardian despite the user not even knowing the database: The agent was just given a task and was sent away.
- The agent is self-learning and will also learn how to communicate efficiently with the database guardian. The agent will meet other agents and tell them about the database and what to find there.

- Agent communication is rather robust: What cannot be understood will simply be ignored. No error messages occur like in classical client-server protocols.
- Hence the implementations of both sides are almost decoupled.
- Independent from network protocols: The agent servers care for the travelling of the agents.
- With IBM's Aglets even the server itself is platform independent, because it is implemented with Java.

One obvious enhancement of this agent approach is to allow agents not only to search for information, but also to provide information. In this scenario the guardian has to be extended to be able to insert information into the database. Finally, the database itself could send out agents – either to provide the contained information actively, e. g. to meet information seeking agents at virtual marketplaces [GMM98], or to seek new information to be stored in the database.

4.4 A Robot Interface to Enable Database Backed Web Applications for Search Engines

Search engines use programs – called robots, wanderers, spiders or crawlers – to index the whole Web [⇒OUC, ⇒IBIS, Tan96, Gra97]. The problem with search engines and database backed Web applications is that the latter need links with lots of parameters in their URLs, which the former simply ignore. Hence we need static pages which the search engines can bookmark. The first straight-forward approach is to generate one large static page which includes all the keywords both in its META tags and in the contents. The links from these keywords then point directly to the application. Such a page can be directly generated from the repository described in section 4.2. The disadvantage, however, is that for a very large database this would mean loading the textual representation of the whole database over the Web. Even for a few megabytes (which database experts would call 'small') this is not reasonable.

The alternative is to repetitively build a static hypertext tree from the database, following the entity relationship structure. First, all m:n relations have to be transformed into 1:n relations, possibly introducing new entities [HeuS97]. An HTML page is then generated for each entity instance and for each 1:n relation a list of links to the related entity instances is included into the page belonging to the entity instance at the 1-side of the relation.

Afterwards the following simple heuristic algorithm determines a reasonable small start page from which all other pages can be reached:

1. All entities which are at the n-side of 1:n relations, but have no other relations, are removed including all relations of these entities.

2. Step 1 is repeated until there are no more entities with only 1:n relations.
3. Now entities with 1:n *and other* relations are removed one by one. The entity with the most 1:n relations is always removed first.
4. If there are no more 1:n relations, the same is continued with the is-a relations removing all the specialized entities in one step.
5. Finally, the entities with 1:1 relations are removed one by one until there are only entities left but no more relations.
6. For each entity left, a static page with a list of links to the entities' pages is now built.
7. Finally a root page is built, which contains the links to the previously mentioned lists.

This root page is the starting point for the generated hierarchical hypertext tree. The algorithm eliminates in each step only entities that can be reached from another entity, because it eliminates only entities with relations. The order of the steps *tends* to leave the smallest number of entities at the end, but it is possible to construct ER diagrams for which this does not hold.

An exact algorithm can be applied using graph theory: First the ER-diagram is converted into a directed graph. The entities become nodes and the relations edges. All the 1:n relations are directed from the 1-side to the n-side, all is-a relations are directed from the general side to the special side and all 1:1 relations are directed arbitrarily. Now in general a cyclic directed graph is built. The next step is to calculate the condensation of this graph by reducing the strong (cyclic) components (subgraphs) to nodes. The resulting graph is acyclic, hence all the nodes, to which an edge points, can be deleted because they can be reached from somewhere else. Finally from each of the remaining strong components one node is chosen arbitrarily, which gives the entities that can be reached directly from the root page. This algorithm is mathematically proven in [Har74].

The root page can now be added manually to a search engine (e. g. AltaVista [⇒Alta]), the rest of the pages will then be retrieved automatically. For a multilingual system it is reasonable to build one tree for each language. The contents of the generated pages is derived from the same repository as described in section 4.2, especially from the part, where the entry points for single tables are defined.

- What attributes are shown as contents of the anchor tags
- What additional information is displayed
- The URLs pointing into the database application

Within all the pages, the META tags `description` and `keywords` are also filled with attribute values from the database and with keywords associated with the appropriate tables (defined in the repository), e. g.

```
<META name="keywords" content="flan, cake">
```

A short explanation on each page will prevent the user from following the static link structure and instead encourage him to use the links pointing into the database applications. Tricks to hide the misleading static links are strongly discouraged by the providers of the search engines. Such tricks could be to use invisible links (using the background color) or to define the META tag

```
<META http-equiv="refresh" content="0; URL=http://database?entry=point">
```

to cause an immediate timed refresh to directly jump into the database application. [⇒Alta] for example does not bookmark pages, which use such tricks.

This method enables the bookmarking of the whole database contents by any search engine with reasonably small pages and the possibility to jump into the dynamic application. This method can also be applied with intelligent agents, that expect a static hypertext tree to traverse.

4.5 Future Work

The following ideas may provide hints for future work:

- Better linguistic analysis as in [TB90] with more powerful assignment of the application entry points. Usage of associative and related terms including a probability estimation. Context grouping for long sentences.
- Semantic description of the application's entry point to enable a more flexible detection of the right entry point from a search result.
- Learning abilities to extend the repository during runtime, e. g. what queries belong to which tables and attributes and – perhaps – which entry points. The questions here involve: How can the guardian be trained? How should an efficient internal knowledge representation look like? Research results in the areas of AI and expert systems should be applied to the database guardian.

Chapter 5

Results and Conclusion

Evaluation of the design and implementation methods has been done by real implementations. It is the author's opinion that there is no better way to show that it works. This chapter presents two successful pilot implementations: A product marketing and tourism information system for the project DEMETER and a Web accessible interactive database training server. A quantitative comparison between the Java and the pure HTML versions is provided. The chapter concludes with references to related work, a short summary and suggestions for future work.

The development environment of the pilot implementations is an Oracle Server 7.3.3 on a Windows NT 4.0 and on a Sun SPARC Solaris 2.5 Server. The implementation language is PL/SQL 2.3 [Stü96] at the server side. For a Web server the Oracle Web Listener 2.1 and 3.0 [⇒Oracle] are used. At the client side either pure HTML or Java is used.

5.1 DEMENET - The DEMETER Project

The first pilot implementation is a product marketing and tourism information system for the DEMETER project. The project DEMETER[1] is funded by the European Commission under the 4th Framework Programme TURA[2]. It is specifically tailored to farmers' needs, aimed at enhancing their quality of life and finding new sources for additional income.

The project contributes in several ways to rural development and enhancement of living conditions for farmers throughout Europe. Currently the flexible technical infrastructure – called DEMENET – offers a framework which provides

- Telematics courses in computer and telecommunications.
- Access to individualized courses in food technology, gardening, new European standards for food processing and other farming specific courses.

[1]Distance Education, Multimedia Teleservices and Telework for Farmers
[2]Telematics for Urban and Rural Areas

- Information systems for rural areas.
- Database for product marketing (regional, national and international)
- Tourism and village information systems.
- Management guidance and counseling (forms, accounting,...).
- other farming specific information, according to user needs.

The project also encourages the users to build their own contribution to a decentralized common European DEMENET structure. DEMENET is designed as a tool for all interested farmers, an easy to use, low cost system (implementation and participation) that provides ample possibilities for rural development. Currently, it is in the pilot phase.

The database for product marketing and tourism information was designed to meet the following requirements:

- The standard user interface has to work with almost any, inclusing older, browsers and platforms and should not depend on proprietary nonstandard extensions like Plug–Ins or Active–X. Thus the implementation is restricted to pure HTML.
- Decentralized global area distribution. The decentralized structure of the implementation must be extendable.
- At the moment, especially in Europe, the performance bottleneck of every Web based application over the Internet is the latency of the net itself. Despite that, this application has to run at a reasonable speed over the Internet.
- Multilingual: The system must be available in different languages. It must be possible to change the language at any time during use in a transparent way.It must also be possible, of course, to edit the system on a multilingual base: The multilingual user interface as well as the content must be editable and translateable during runtime.
- When speaking of the Internet we cannot ignore problems concerning security: This is a general problem and standard solutions e. g. secure sockets layer (SSL) or integrated SmartCard [DMG96, MGD97] solutions will be used.

The DEMENET application has been implemented successfully using the techniques described in this thesis both with pure HTML and with Java. Details about this application can be found in [RGR97]. Everyone is invited to visit the pilot server, and to understand how this actually works. Its URL is `http://land.ict.tuwien.ac.at/`[3]. Currently the system is being evaluated by a group of test users throughout Europe, mainly to improve the layout and the database structure. The first feedback from our users looks promising.

[3]This serves as a redirection URL to the current implementation.

5.2 Web Accessible Interactive Database Training

Distance education, in most cases now ODL (Open and Distance Learning) has been proven by modern research to be the most efficient method to enable growing numbers of students especially to get engineering education most quickly and to the highest standard possible [Gas95, FW94, Ren95, Dav95]. Creating interactive simulations and laboratories further enhances the efficiency of the learning process significantly. As one possible implementation of these concepts a Web accessible tutorial for interactive database training is presented. First results show that organizational and educational advantages can be obtained, e. g. better student lecturer interaction and personalized access to information combined with interactive simulation systems. With this approach to engineering education we hope to create the most efficient learning and teaching environments, fostering creativity and excellence in order to be prepared for the already enhanced international competition.

If hypertext material is to be successful, it is not enough just to translate the paper version of the lecture notes to the Web, but it is necessary to take advantage of the full range of hypertext features. The main attraction of the Web is interactivity, leading to the design of an interactive laboratory following the rules of open and distance learning concepts. This laboratory accompanies the lecture "Databases on the World Wide Web": Within the laboratory the students have the following tasks: Design of a small database with ER and implementation with SQL. Some queries with SQL. Design of a small document collection including a homepage. Connecting the database to the Web, creating some dynamic HTML documents from the database on the fly and doing some update operations in the database using the HTML FORM-interface.

The first part of the laboratory is an interactive SQL-tutorial: Each group of four students receives an example where 15 queries have to be solved on a given data structure. On the welcome screen the data structure is shown, including all base relations and a short description. After reading and understanding the data structure, the student can proceed to the first query task, where the expected results are also shown. The first SQL query is then entered.

If the try was syntactically correct but not a solution to the given problem, an error message is obtained. After entering the correct SQL statement the student receives a confirmation. If a syntactically incorrect statement is entered, the error message from the database is forwarded. To determine, if the student has entered a correct solution, the server does not parse the SQL input but rather compares the result of the student's SQL request with the result of the predefined correct SQL solution statement executed on the same data. If the student has already solved the task, the alternative correct solutions are provided afterwards.

This provides the possibility of misuse, when a student enters an SQL statement which is tailored towards the expected results but only works on the actual data. The system does determine this statement as correct and provides the alternative results, but every SQL statement which leads to a correct response is saved into a log file which can later be evaluated, if someone tried to cheat.

The second part of the laboratory is to design and implement a small database and to implement database interaction from the HTML user interface, e. g. to insert a form input into the database or to generate a Web page dynamically from the database contents. The implementation is done through PL/SQL programs located in the database. To complete the second laboratory part another Web interface is provided, which allows the execution of any SQL statement to create and modify the data structure, the editing of the student's database contents, the uploading and compilation of the PL/SQL code and the uploading of static HTML pages and pictures. Again, the only thing needed at the client side is a Web browser.

The outstanding feature of this laboratory is that there is no need to be present at the institute, which is unusual at the Vienna University of Technology. Advanced students (tutors) give assistance mainly through a mailing list. Once a week the office is open for personal contact. Thus the lab can be accessed from anywhere on the Internet, with just a Web browser at the client side. For the students this gives the possibility of learning at their own speed, when and where they want (student centered learning).

The laboratory itself is implemented as a database backed Web application, using the techniques described in this thesis to generate the Web interface pages automatically from the database contents and to compute the user input data. The only difference from the students' implementation is, that the laboratory implementation uses dynamic embedded SQL rather than static embedded SQL, since it has to pass the students' SQL statements to the database and then include the results in dynamically generated HTML pages. Thus the meta-level of tutorial implementation uses essentially the same technologies as the students have to use when completing the tutorial. As an alternative implementation technique the same functionality for the two parts of the laboratory has been provided with two Java applets. The resultant differences are described in chapter 3.

The complete laboratory is available at `http://aki.ict.tuwien.ac.at/` in both English and German, details can be found in [GR98, GRRF98, RGM97]. Only the second part of the laboratory is restricted to authorized access, the first part and the demo application are available without restriction and anyone is invited to visit them.

The feedback of the students was consistently positive: The vast majority of the students (97%) liked the ability to work at the laboratory anytime and anywhere, only 3% felt insecure and would have prefered a more restrictive course. However, the prototype implementation worked satisfactorily and the performance was good. With 90 engineering students having worked with the application for two months, the implementation method has gone through its baptism of fire.

5.3 Quantitative Analysis

This section introduces a decision criteria which helps to decide whether a pure HTML or a Java implementation performs better in terms of network traffic. The presented results are derived from measuring the prototype implementations both in their pure HTML

form and in their Java form. Firstly, the measuring method is shown and the results are discussed.

There are different methods of measuring the network traffic of a server application:

- The application itself can be modified to record the data sent to or received from a client. The advantage is, that the data can easily be prepared in a useful structure. However, the disadvantage is, that the application is changed. This might also affect the measuring itself, but the greatest problem is that for repetitive measuring during development the measuring has always to be adopted.
- A task is programmed that observes the network connection of the server. The application itself is no longer affected by the measuring because of this, but the normal network daemons have to be replaced by tasks that also record the network data. This might affect the speed of the server's network connection.
- Another network node is introduced in the form of an observer PC. The disadvantage is, that more resources are needed (additional PC, repeater hardware) and the reconstruction of the interesting data transfers from the network traffic is more complicated. However, the advantage is, that the measured system is not affected[4].

For its advantage of not affecting the normal operation of the measured server, the third method has been used. Figure 5.1 shows an overview of the measuring environment. Normally a network card discards all received frames but those addressed to the MAC (Medium Access Control IEEE 802.3 [Tan96],p.280) address of the card or to a group the card belongs to (multicast) or broadcast frames. Hence the network card of the observer PC (an Intel EtherExpress Pro 100 in this special case) is switched to promiscuous mode to record all frames on the network. The network is not affected, because the network card sends nothing in promiscuous mode. A sniffing tool is used to communicate with the card and to record all the frames into a raw text file. This file grows rather fast and can easily reach a size of more than 100MB within an hour. Of course this depends on the server traffic.

Therefore, the raw data file is directly streamed into a filter that eliminates all the frames not needed. Perl [WSCP96] has been used as implementation language, because it has proven to be very useful for the text detection and replacement operations needed. This filter eliminates wrong ports, e. g. 110 (Post Office Protocol POP 3) or 139 (NetBios Session Service) [WE96], reducing the size of the file to typically tens of megabytes.

During the next step a preprocessor loads the data into a database, stripping everything but the TCP and IP protocol headers and the contained data. The database size of one measurement is typically a few megabytes. The database contains one table for sockets, each with a unique socket identifier as primary key, ordered by the time the sockets were opened. This table also contains the number of frames and the complete amount of data transferred within this socket. A second table contains all the frames tagged with a serial number (in the order of their appearance) and a foreign key to the

[4]Only the repeater may introduce a not recognizeable latency in the network traffic.

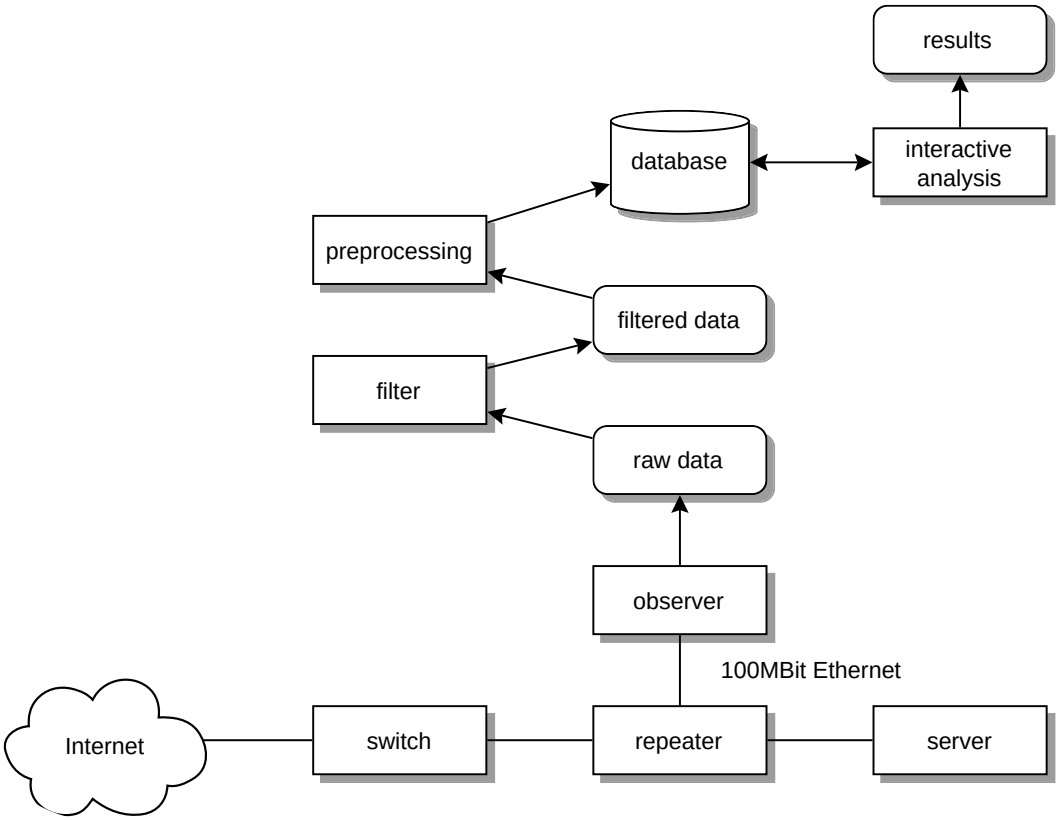


Figure 5.1: Quantitative analysis measurement environment.

socket to which the frame belongs, ordered by the socket identifier first and then by the frame number within each socket. Each socket contains at least three frames for the TCP connection setup at the beginning, then the data and finally typically two frames for the TCP connection closedown. The frames for Java and HTML can be differentiated by their port. Identification of parallel sessions is done with session identifiers.

To compare Java and HTML applications with each other we have to face the problem, that HTML opens a socket for each HTTP connection, whereas Java applications typically open a small number of sockets for the connection oriented JDBC connections. What we finally require to compare is the number of transferred data over the number of user interactions for the same application implemented both with Java and HTML. With HTML each user interaction leads to a HTTP connection and hence the number of sockets roughly[5] equals the number of user interactions. Apparently, this is not true for Java.

Therefore typical use cases observed from user profiles of pilot users are taken and synchronization points are defined within them, such that the number of user interactions between two synchronization points is well defined (and equal for both the Java and HTML implementation) and the synchronization point itself can easily be detected from

[5]Additional sockets are opened for each image contained in the HTML pages. With the applications discussed here, almost all images are downloaded at the beginning and stored in the browser cache. If this does not hold for another application, the number of sockets has to be divided by the average number of images per page to receive the number of user interactions.

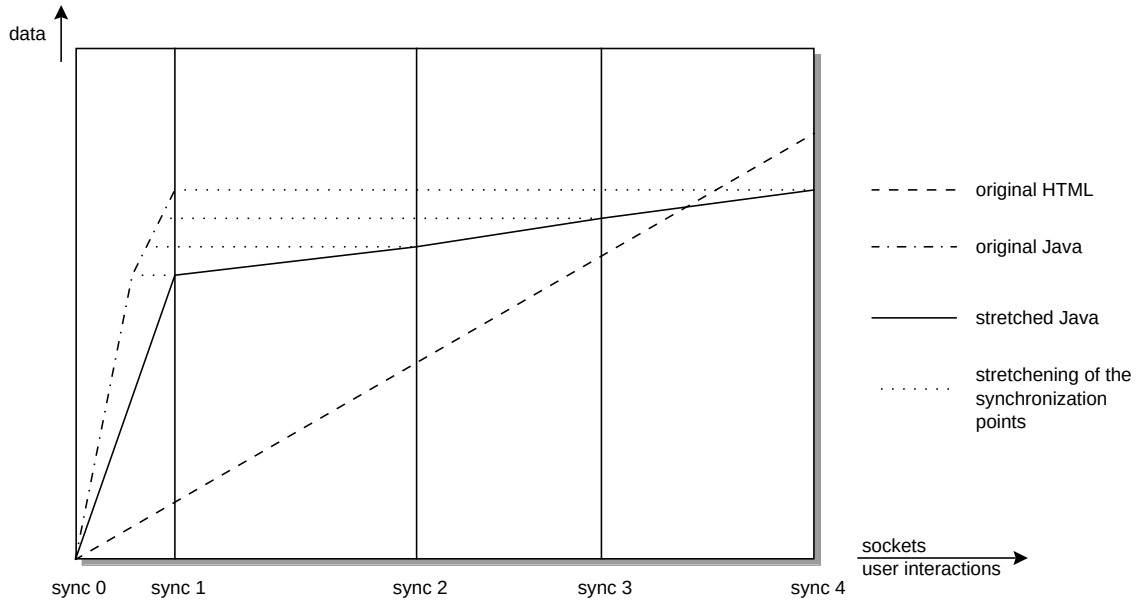


Figure 5.2: Stretching of the Java curve.

data transferred over the network (e. g. a typical SQL statement). These synchronization strings are also stored in the database, as pure ASCII text for Java and with encoded entity references for HTML. The intent is, to use the synchronization points of the HTML curve (data over sockets respectively user interactions) to stretch the original Java curve (data over sockets), such that the Java synchronization points meet the HTML synchronization points. Hence the strechted Java curve finally shows data over user interactions. Figure 5.2 shows this method: The original HTML curve shows data over sockets. As previously mentioned, this equals roughly to data over user interactions. The original Java curve data over sockets apparently uses far less sockets. Finally the Java curve is stretchened with respect to the synchronization points, receiving a Java data over user interaction curve (interpolation).

A second run of the preprocessor searches the frame data for the predefined synchronization points and marks the frames where a synchronization point has been detected. Also with the second run the cumulative number of transferred bytes so far is stored for each frame. Interactive analysis tools can now operate on this database, or the database can be exported into an MS Excel spreadsheet.

Figure 5.3 shows the result measurement: Java needed 20 sockets, HTML 369. Hence the Java curve is stretched with the help of eight synchronization points. The Java applet is loaded up until the first synchronization point as is some data from the database. With HTML, due to the image loading, more data is transferred before the first synchronization point than afterwards too. From the first synchronization point both curves are almost linear. HTML shows a higher gradient, because a complete HTML page (approximately 1.8kB) has to be loaded for each user interaction, whereas with Java only a small amount of data from the database has to be transferred (approximately 0.6kB). Figure 5.4 shows the difference between the two curves. For more than 200-300 user interactions Java typi-

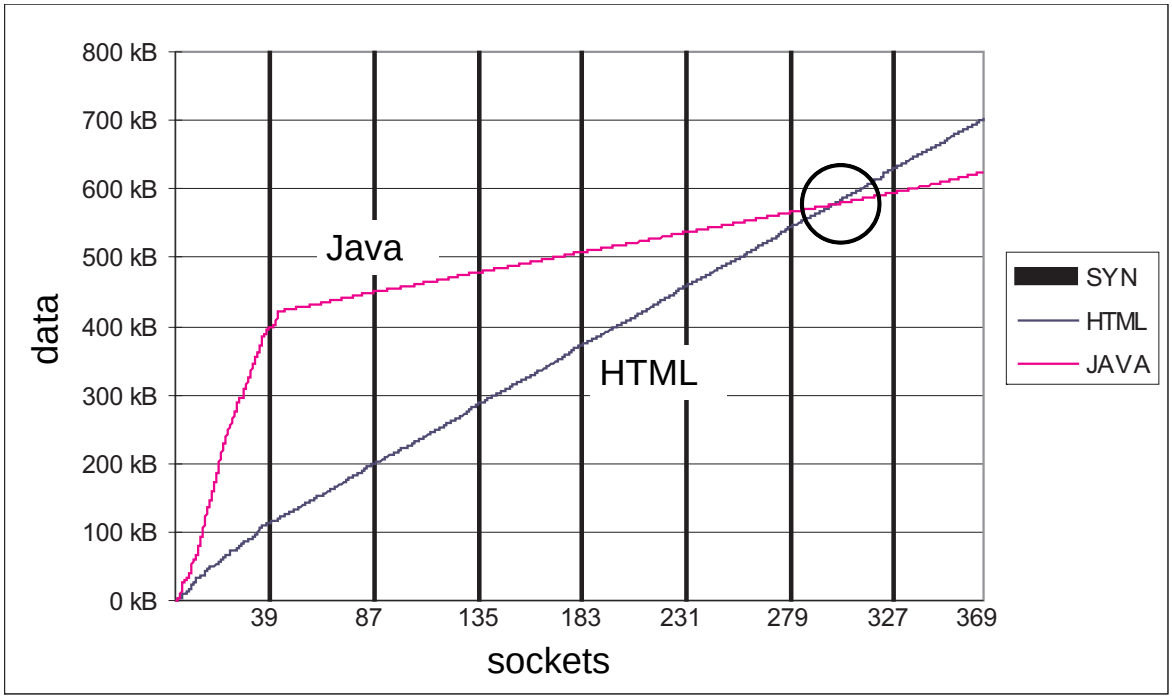


Figure 5.3: Java versus HTML: Transferred data over user interactions.

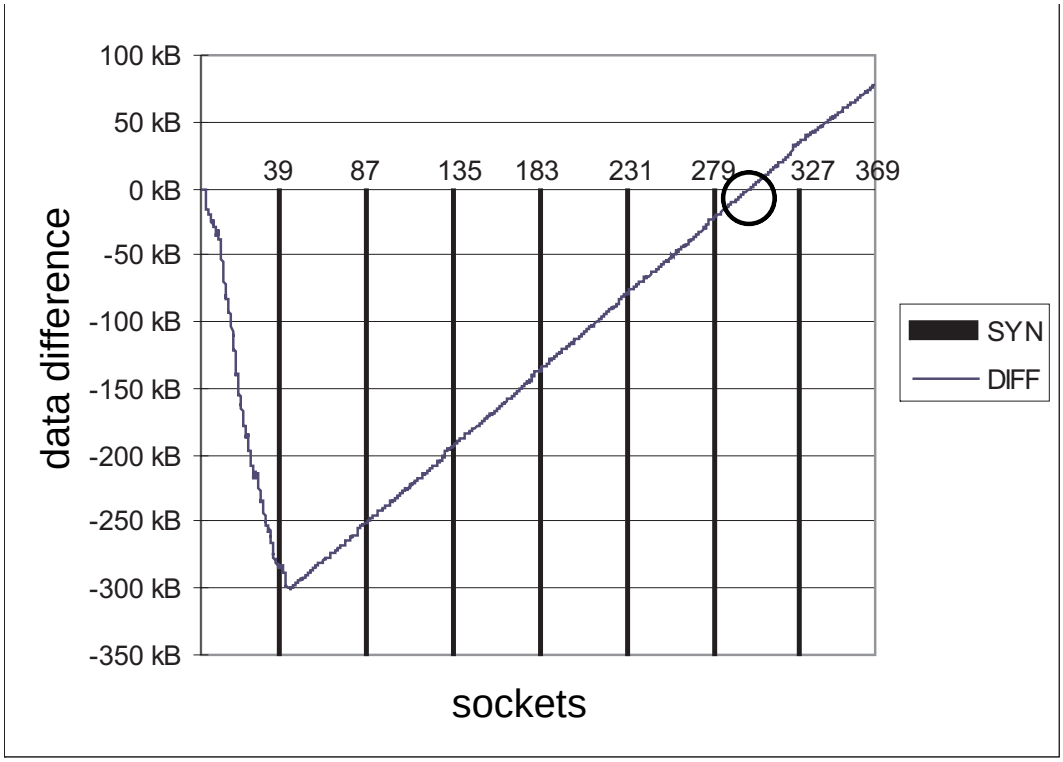


Figure 5.4: Java versus HTML: Difference of transferred data over user interactions.

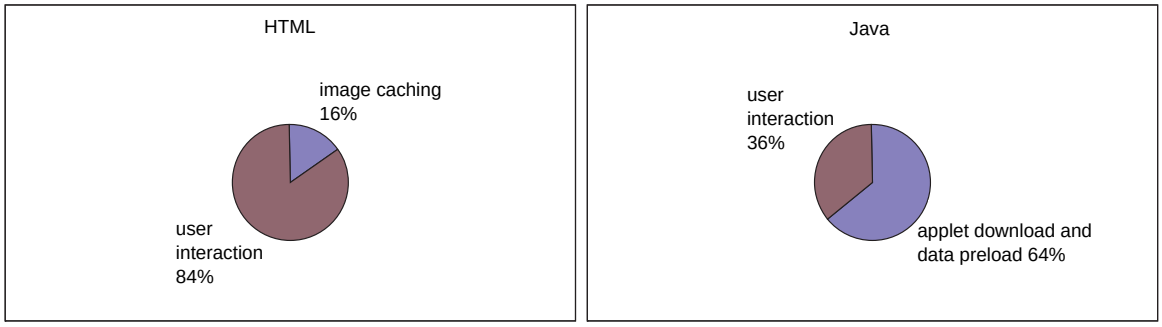


Figure 5.5: Java versus HTML: Download at the beginning.

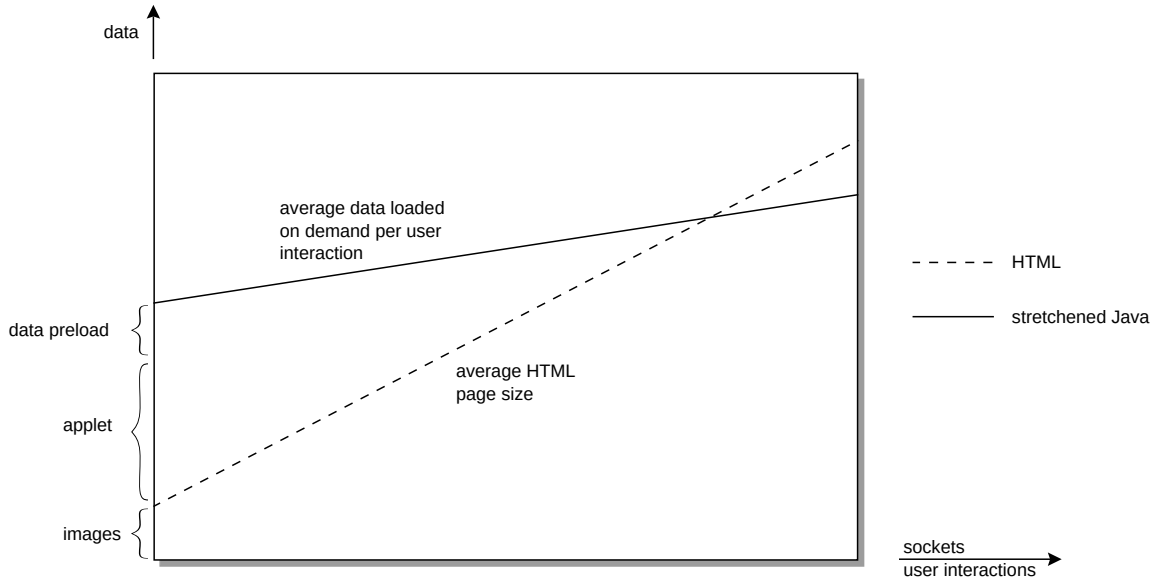


Figure 5.6: Rule whether Java or HTML performs better.

cally performs better than HTML in terms of network traffic. Finally, figure 5.5 shows the different percentage of data downloaded at the beginning (until the first synchronization point) and during the user interactions.

Based on the previous measurement, the following rule of thumb gives a rough criteria to determine whether Java or HTML is better in terms of network traffic for a particular application. Given an applet size D_{Applet} and a data amount $D_{preload}$ to be preloaded, both in bytes, and an average data amount d_{Java} per user interaction loaded on demand from the Java applet compared to an average size d_{HTML} of the dynamically generated HTML page, both in bytes per user interaction, the following formula gives the number of user interactions U, where Java outperforms HTML (see figure 5.6):

$$U = \frac{D_{Applet} + D_{preload}}{d_{HTML} - d_{Java}} \tag{5.1}$$

The size of the images does not count because both implementations load them once and afterwards hold them in the cache. However, to determine more exactly which

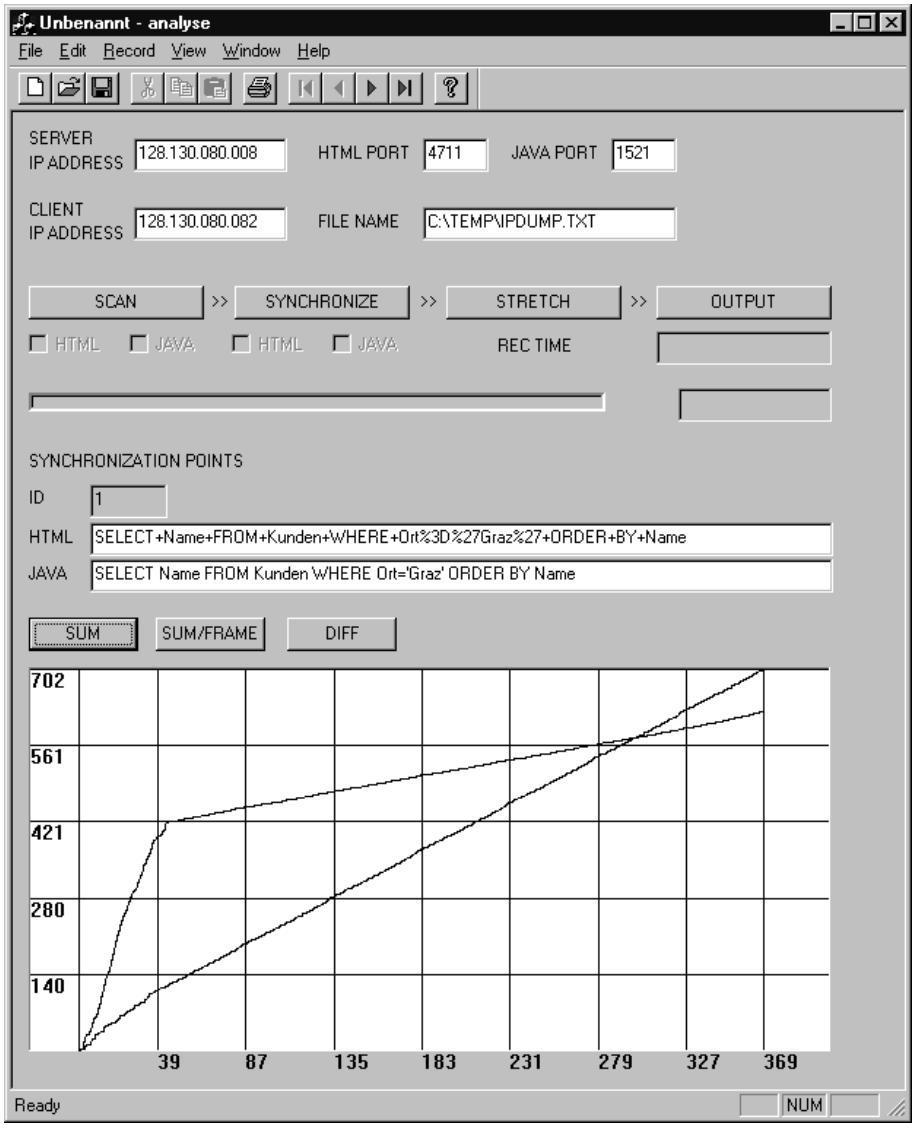


Figure 5.7: User interface of the analysis tool.

implementation is better, the presented measurement method has to be applied to the particular application implemented in both Java and pure HTML observed with typical user behaviour. The analysis tools developed for this purpose are described in [Kam98] and figure 5.7 shows an impression of the user interface.

5.4 Related Work

The large software companies adopt their powerful design and implementation tools to generate large scale Web applications, e. g. Oracle Designer/2000, Oracle Developer/2000 [⇒Oracle], Web Objects Framework [⇒WOF], SAP@Web [⇒SAP], IBM AS400 [Hub98]. These work well, but they are expensive, not very easy to use, and mostly oriented towards the needs of the Intranet, not the Internet. Moreover, they have

a long history in the area of classical programming paradigms and are thus not perfectly suited for designing Web applications. Some of them lack functionality and have unnecessary restrictions. However, they can be the best method anyway, if a company is already running one of these systems.

On the other hand, many freeware solutions exist but they have performance problems with large scale applications. Most of them have a short live-span, no technical support, and are not suitable for commercial products. Perhaps some solutions based on the Linux OS will become interesting in the next time, for example PHP3, a scripting language embedded into HTML with some features for database integration of many different database types [⇒PHP].

A lot of research work is done on static hypertext documents as automatons [SFC98] and about enhancements of HTML with additional tags and meta-tags to generate large static document trees from a concise functional description embedded into HTML code [BS98]. These work well – as long as there is no need to integrate a database.

Research on the integration of databases and HTML mainly focuses on forms [⇒heitml], whereas this thesis proposes mainly the use of links for interaction with the database. So far no research work could be found that uses a systematic model and a design language for application generation of link based interaction between databases and the Web.

5.5 Summary

The aim of this thesis was to make the design of database backed Web applications easier and the implementation less prone to error. In the opinion of the author this goal has been reached, as the two real-life applications both of which were implemented successfully with pure HTML and Java show. The feedback from the first users was positive. The inherent difficulties of the integration of relational databases and the Web could be overcome. This also allows the connection of legacy relational database systems to the Web with similar functionality as in classic implementations.

The finite state machine model, well known from static hypertext documents, could be remodelled to suit dynamically generated hypertext. Groups of links are mainly used instead of forms to interact with the database. An object based client-server model was presented to design the user interface layout, the middleware functionality and the database transactions in a homogenous way. The actual implementation of such a system was then composed from several parts with potentially different techniques, e. g. pure HTML, Java or VRML. A toolset was implemented to generate the different applications automatically from the PHC/DL and PHC/LL descriptions. The toolset itself is a Web application fostering unlimited Web collaboration. The tools were designed from scratch, closing the current gap between research theory and the market. They are especially suited to the needs of the Web without overhead caused by classic programming paradigms.

By using this technique it was shown how to build complete information systems, including logical page flow and bidirectional crosslinks, multi language support and frames with

different window modes. The design methodology further guarantees stable and robust applications with sophisticated user interactions, compatible with almost any browser, and flexible layout design separated from the application functionality. The approach is Internet–oriented and deals with the problems of wide area distribution with small bandwidth connections. It relies on the proven technology of relational databases.

The usability of Java was compared with the pure HTML approach, investigating different client-server tradeoffs and persistence frameworks. Object oriented design and implementation has some drawbacks with typical information systems – it was shown how Java can supplement the HTML approach but not supplant it. A quantitative analysis showed that pure HTML typically outperforms Java in terms of network traffic for less than 200-300 user interactions.

As different as they are in design and implementation, relational databases and the Web also differ considerably in the ways they can be searched. Universal relations and natural language interfaces known from database theory have been combined with keyword searches known from the Web to define a metadata model for searching a database backed Web application. The key idea here was the separation of structural and content meaning of words. Based on this idea generic interfaces for both Intelligent Software Agents and robots from search engines were implemented.

Despite the limits of pure HTML, a sophisticated user interface could be designed, which mainly uses links for interaction but no JavaScript, Plug–Ins or Active–X. With the framework tools, a novel design and implementation technique for Web based distributed information systems was implemented. It is now possible to *generate* the different implementations automatically from the design language. With this development framework, the design and implementation of various database powered Web applications is easier, faster and less prone to error.

5.6 Future Work

The following suggestions for future work are summarized from the previous chapters:

- A completely object oriented graphical version of the design language PHC/DL including tool support [ISB95].
- Object oriented notation, also for the PHC/LL, possibly using the Extensible Markup Language XML or another subset of SGML. Object oriented approaches like [BS98, ⇒heitml] will be evaluated with respect to their usability as PHC/LL replacement.
- Enhancement of the generator tools to provide Java (see section 3.3) and VRML as alternative user interface implementations. The object relational successors of the current relational systems will provide better integration possibilities for Java.

- Enhancement of the pure HTML interface with a scripting language as soon as it becomes clear which one will be the major client side scripting language on the Web (Netscape's JavaScript or Microsoft's VBscript).
- Integration of a security framework into the PHC approach.
- Usage of the connection oriented successors of the HTTP as far as they become widely available.
- With different generator backends, it will be possible to generate PHC applications for different database management systems because the whole design methodology is not dependent on a particular product, as long as the database understands standard SQL.
- Enhancements of the search interface with better linguistic analysis as in [TB90] with more powerful assignment of the application entry points. Usage of associative and related terms including a probability estimation. Context grouping for long sentences.
- Semantic description of the application's entry point to enable a more flexible detection of the right entry point from a search result.
- Learning abilities for the database guardian agent to extend the search repository during runtime.

Appendix A

More PHC Examples

The following examples show more sophisticated features of both the design and the layout language. The syntax definitions can be found in appendix B. Figure A.1 shows the PHC/DL description of a more sophisticated version of the geographical selection. The enhancements can be found in the three lists where the cursor loops are now explicitly defined. Within these cursor loops various `IF` statements define different `PRINT`, `IMAGE` and `GOTO` clauses depending on the boolean value of the respective condition. This example also shows the definition of conditions with the `CONDITION` clause. Figure A.2 shows the usage of these conditions in PHC/LL in detecting, where a sublist has to be placed in a list resulting in a browser view similar to a file explorer as shown in figure A.3.

With slight modifications (as shown in figure A.4) the first layout description remains valid and now generates the same output from the enhanced version of the PHC/DL. Alternatively, if `SelList` is kept within the enhanced version of the PHC/DL, the first PHC/LL description remains valid even in an unchanged form. The generated code becomes slightly larger in this case, but the performance of the final application is the same.

To show the possible variations which PHCI enables a third layout is presented for the same PHC/DL. Figure A.5 shows the PHC/LL description and figure A.6 shows an example of a browser view in state S3.

The PHC 'Selection' in figure A.7 shows the possibility of more complex local variables: In this case two temporary tables are built and used for the output of the element 'Matches'. This PHC is also an example for a PHC with just one state and thus declared as 'stateless' with the `NOSTATE` clause in the interface part. Thus the programmer *can* use states, but he does not have to.

The `GOTO` clause in the element 'Show' gives an example of a state manipulation message sent to *another* PHC: The PHC 'Result' is set to the state S1 and eight public session state parameters of this PHC are set to the given values. The `PAGE` clause defines the page `Result` as the new active page.

The PHC 'Result' in figure A.10 shows a list containing more than one element within its query. The elements refer to the attributes of the query by the cursor name defined

```
PHC Geo

// geographical selection
// version 2

INTERFACE

PARAM
    Province PUBLIC ROWID
        PRINT tProvinz.aBezeichnung[aProvId];
    Region PUBLIC ROWID
        PRINT tRegion.aBezeichnung[aRegId];
    District PUBLIC ROWID
        PRINT {SELECT aBezeichnung FROM tGemeinde WHERE aGemeindId = PARAM;};

ELEMENT
    Overview, OneUp, Province, Region, District;
LIST
    ProvList, RegList, DisList;
STATE
    S0, //nothing selected, the first state is the start state
    S1, //province selected
    S2, //region selected
    S3; //didtrict selected

IMPLEMENTATION

BEGIN
VAR
    Anz INT;

ELEMENT Overview (
    ALWAYS { PRINT 'overview'; IMAGE fover2; GOTO S0; };
)

ELEMENT OneUp (
    ALWAYS { PRINT 'higher region level'; IMAGE fup1;};
    STATE S0,S1 { GOTO S0; };
    STATE S2 { GOTO S1; };
    STATE S3 { GOTO S2; };
)

ELEMENT Province (
    ALWAYS { GOTO S1; };
    STATE S0 { PRINT NULL; };
    STATE OTHER { PRINT PARAM Province; };
)

ELEMENT Region (
    ALWAYS { GOTO S2; };
    STATE S0,S1 { PRINT NULL; };
    STATE OTHER { PRINT PARAM Region; };
)

ELEMENT District (
    ALWAYS { GOTO NULL; };
    STATE S0,S1,S2 { PRINT NULL; };
    STATE OTHER { PRINT PARAM District; };
)

LIST ProvList (
    ALWAYS
        QUERY ProvQuery IS
            SELECT aBezeichnung { PRINT; },
                aProvId { SET PARAM Province;}
            FROM tProvinz;
    (
        ELEMENT //implicit name is "ProvList"
        {
            SELECT COUNT(*) INTO Anz
            FROM tRegion
            WHERE aProvId = RegQuery.aProvId;
        IF ((ProvQuery.aProvId == PARAM Province) AND (STATE != S0)) {
            CONDITION selected;
            GOTO S0;
            IF (Anz == 0)
                IMAGE fopen1;
            ELSE // Anz != 0
                IMAGE fclosed1;
        }
        ELSE {
            GOTO S1;
            IF (Anz == 0)
                IMAGE fclosed1;
            ELSE // Anz != 0
                IMAGE fclosed3;
        };
    }
)

LIST RegList (
    STATE S0 { NULL; };
    STATE S1,S2,S3
        QUERY RegQuery IS
            SELECT aBezeichnung AS Name { PRINT; },
                aRegId AS Id { SET PARAM Region; }
            FROM tRegion
            WHERE tRegion.aProvId = PARAM Province;
    (
        ELEMENT //implicit name is "RegList"
        {
            SELECT COUNT(*) INTO Anz
            FROM tGemeinde
            WHERE aRegId = RegQuery.Id;
        IF ((RegQuery.Id == PARAM Region)
            AND (STATE != S0) AND (STATE != S1)) {
            CONDITION selected;
            GOTO S1;
            IF (Anz == 0)
                IMAGE fopen1;
            ELSE // Anz != 0
                IF (STATE == S2)
                    IMAGE fopen2;
                ELSE // District selected
                    IMAGE fclosed2;
        } ELSE {
            GOTO S2;
            IF (Anz == 0)
                IMAGE fclosed1;
            ELSE // Anz!= 0
                IMAGE fclosed3;
        };
    }
)

LIST DisList (
    STATE S0,S1 { NULL; };
    STATE S2,S3
        QUERY DisQuery IS
            SELECT aBezeichnung AS Name, aGemeindId AS Id
            FROM tGemeinde
            WHERE aRegId = PARAM Region;
    (
        ELEMENT //implicit name is "RegList"
        {
            PRINT DisQuery.Name;
        IF ((DisQuery.Id == PARAM District) AND (STATE == S3)) {
            CONDITION selected;
            IMAGE fopen1;
            GOTO S2;
        } ELSE {
            IMAGE fclosed1;
            GOTO S3 SET PARAM District := DisQuery.Id;
        }
    }
)

END //Geo
```

Figure A.1: Alternative PHC/DL description for PHC Geo with more complex features.

```
<TABLE  WIDTH=360 BORDER=1>
.....

<TR><TD WIDTH=150 ALIGN=center><CENTER><B>Region selection</B></CENTER>
<!--$PHC IMAGE(Geo.OneUp); -->
<FONT COLOR="GREEN"><!--$PHC PRINT(Geo.OneUp); --></FONT><BR></TD></TR>

<TR><TD ALIGN=left VALIGN=top><FONT SIZE=2><!--$PHC IMAGE(Geo.Overview);-->
<FONT COLOR="GREEN"><!--$PHC PRINT(Geo.Overview); --></FONT><BR>


<!--$PHC FOR prov IN Geo.ProvList LOOP
        IMAGE(prov); PRINT(prov); -->
        <BR>
<!--$PHC IF prov.selected BEGIN
                FOR reg IN Geo.RegList LOOP -->
                        <IMG SRC="show_image?pname=fline1" VSPACE=0 WIDTH=17>
<!--$PHC                IMAGE(reg); PRINT(reg); -->
                        <BR>
<!--$PHC                IF reg.selected BEGIN
                                FOR dis IN Geo.DisList LOOP -->
                                <IMG SRC="show_image?pname=fline1" VSPACE=0 WIDTH=17>
                                <IMG SRC="show_image?pname=fline1" VSPACE=0>
<!--$PHC                        IMAGE(dis); PRINT(dis); -->
                                <BR>
<!--$PHC                        END LOOP;
                        END;
                END LOOP;
        END;
END LOOP; -->
</FONT></TD></TR>

.....
</TABLE>
```

Figure A.2: PHC/LL description to PHC/DL from figure A.1.

after the keyword `QUERY`. The element `Caddy` of this list shows the usage of a functional message sent to another PHC. Figure A.13 shows the respective `METHOD` declarations as part of the PHC description of 'Basket'.

Also the `PAGE` attribute determines that the active page will remain the same but the `REFRESH` clause determines that the ouput of the page `Shopping_Basket` will eventually change. Thus this page has to be reloaded, if currently shown in one frame. Figures A.11 and A.12 show the PHC/LL description and the resulting browser view belonging to this PHC.

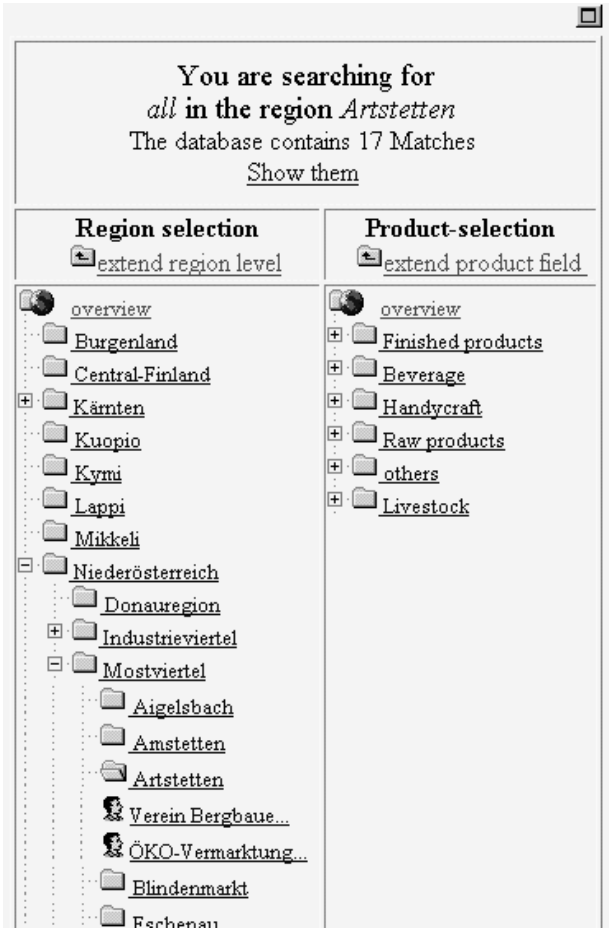


Figure A.3: Browser view of PHC Geo version 2 in state S3.

```
<TABLE  ALIGN="right" WIDTH=115 BORDER=0>
    <TR><TD ALIGN="center" VALIGN=top><B>Region selection</B></TD></TR>
    <TR><TD><FONT COLOR="green"><!--$PHC PRINT(Geo.Overview); --></FONT></TD></TR>
    <TR><TD><FONT COLOR="green"><!--$PHC PRINT(Geo.OneUp); --></FONT></TD></TR>
    <TR><TD><!--$PHC PRINT(Geo.Province); --></TD></TR>
    <TR><TD><!--$PHC PRINT(Geo.Region); --></TD></TR>
    <TR><TD><!--$PHC PRINT(Geo.District); --></TD></TR>
    <TR><TD ALIGN=center><HR></TD></TR>
<!--$PHC IF (Geo.STATE == S1) BEGIN FOR item IN Geo.ProvList LOOP -->
    <TR><TD ALIGN="right"><!--$PHC PRINT(item); --></TD></TR>
<!--$PHC END LOOP; END; -->
<!--$PHC IF (Geo.STATE == S2) BEGIN FOR item IN Geo.RegList LOOP -->
    <TR><TD ALIGN="right"><!--$PHC PRINT(item); --></TD></TR>
<!--$PHC END LOOP; END; -->
<!--$PHC IF (Geo.STATE == S3) BEGIN FOR item IN Geo.DisList LOOP -->
    <TR><TD ALIGN="right"><!--$PHC PRINT(item); --></TD></TR>
<!--$PHC END LOOP; END; -->
</TABLE>
```

Figure A.4: PHL/LL for PHC Geo version 2: Old layout from new PHC/DL.

```
<TABLE  ALIGN="left" WIDTH=100% BORDER=3>
<TR><TD COLSPAN=3 ALIGN="center"><B>Region selection</B><BR><BR>
        <FONT COLOR="green"><!--$PHC PRINT(Geo.Overview); --></FONT><BR>
        <FONT COLOR="green"><!--$PHC PRINT(Geo.OneUp); --></FONT></TD></TR>
<TR><TD WIDTH=33% ALIGN=left VALIGN=top>
<!--$PHC FOR item IN Geo.ProvList LOOP -->
<!--$PHC        IF item.selected BEGIN -->
<!--$PHC                IF (Geo.STATE == S1) BEGIN -->
                                <FONT COLOR="red"><B>
<!--$PHC                END; -->
<!--$PHC                PRINT(item); -->
<!--$PHC                IF (Geo.STATE == S1) BEGIN -->
                                </B></FONT>
<!--$PHC                END; -->
                        <IMG SRC="yel_arrow.gif" ALIGN=absmiddle>
<!--$PHC        END ELSE BEGIN -->
<!--$PHC        PRINT(item); -->
<!--$PHC        END; -->
                <BR>
<!--$PHC END LOOP; -->
</TD><TD WIDTH=33% ALIGN=left VALIGN=top>
<!--$PHC FOR item IN Geo.RegList LOOP -->
<!--$PHC        IF item.selected BEGIN -->
<!--$PHC                IF (Geo.STATE == S2) BEGIN -->
                                <FONT COLOR="red"><B>
<!--$PHC                END; -->
<!--$PHC                PRINT(item); -->
<!--$PHC                IF (Geo.STATE == S2) BEGIN -->
                                </B></FONT>
<!--$PHC                END; -->
                        <IMG SRC="yel_arrow.gif" ALIGN=absmiddle>
<!--$PHC        END ELSE BEGIN -->
<!--$PHC        PRINT(item); -->
<!--$PHC        END; -->
                <BR>
<!--$PHC END LOOP; -->
</TD><TD ALIGN=left VALIGN=top>
<!--$PHC FOR item IN Geo.DisList LOOP -->
<!--$PHC        IF item.selected BEGIN -->
                        <FONT COLOR="red"><B>
<!--$PHC                PRINT(item); -->
                        </B></FONT>
<!--$PHC        END ELSE BEGIN -->
<!--$PHC        PRINT(item); -->
<!--$PHC        END; -->
                <BR>
<!--$PHC END LOOP; -->
</TD></TR>
</TABLE>
```

Figure A.5: Yet another PHC/LL to the same PHC/DL for Geo.

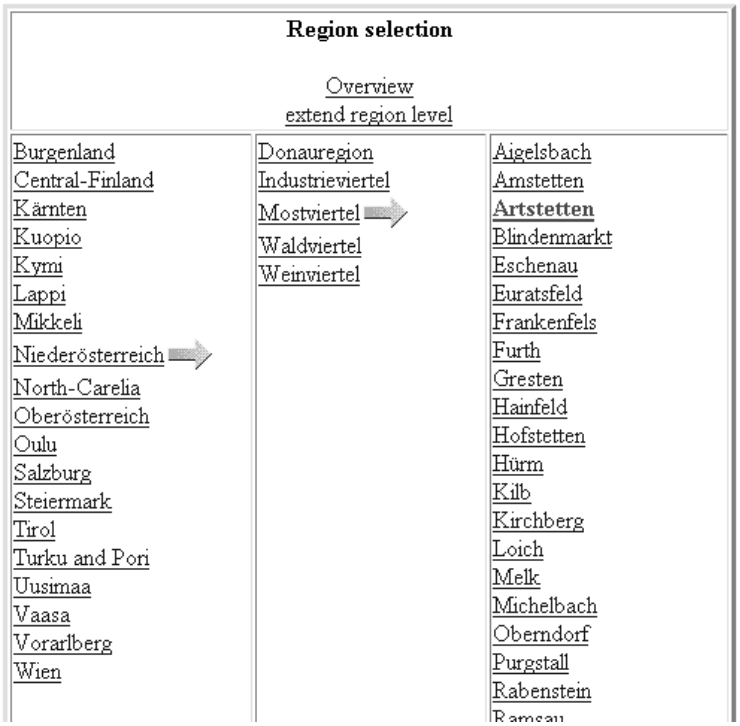


Figure A.6: Browser view of PHC Geo with alternative layout from figure A.5 in state S3.

```
PHC Selection

INTERFACE

ELEMENT
    SearchDis, SearchProd, Matches, Show;

NOSTATE;

IMPLEMENTATION

BEGIN
VAR
    TempDis BAG { Distrıct ROWID }; // optional PRINT SQL statement
    TempProd BAG { Product ROWID };
    Anzahl INT;

{ // 2 temporary tables or 16 different queries (switch/case)

    SWITCH (Geo.STATE) {
        CASE S0 { // nothing selected
            INSERT INTO VAR TempDis (District)
            SELECT aGemeinId FROM tGemeinde;
            };
        CASE S1 { // province selected
            INSERT INTO VAR TempDis (District)
            SELECT tGemeinde.aGemeinId FROM tGemeinde, tRegion
            WHERE tGemeinde.aRegId = tRegion.aRegId
            AND tRegion.aProvId = PARAM Geo.Province;
            };
        CASE S2 { // region selected
            INSERT INTO VAR TempDis (District)
            SELECT tGemeinde.aGemeinId FROM tGemeinde
            WHERE aRegId = PARAM Geo.Region;
            };
        CASE S3 { // district selected
            INSERT INTO VAR TempDis (District)
            VALUES (PARAM Geo.District);
            };
    };

    SWITCH (Prod.STATE) {
        CASE S0 { // nothing selected
            INSERT INTO VAR TempProd (Product)
            SELECT aProdId FROM tProdukt;
            };
        CASE S1 { // category selected
            INSERT INTO VAR TempProd (Product)
            SELECT tProdukt.aProdId FROM tProdukt, tSubkategorie
            WHERE tProdukt.aSubkatId = tSubkategorie.aSubkatId
            AND tSubkategorie.aKatId = PARAM Prod.Category;
            };
        CASE S2 { // subcategory selected
            INSERT INTO VAR TempProd (Product)
            SELECT tProdukt.aProdId FROM tProdukt
            WHERE tProdukt.aSubkatId = PARAM Prod.SubCat;
            };
        CASE S3 { // product selected
            INSERT INTO VAR TempProd (Product)
            VALUES (PARAM Prod.Product);
            };
    };
}

// Show matches

ELEMENT SearchDis
    {
        GOTO NULL;
        SWITCH (Geo.STATE) {
            CASE S0 { PRINT 'any region'; };
            CASE S1 { PRINT PARAM Geo.Province; };
            CASE S2 { PRINT PARAM Geo.Region; };
            CASE S3 { PRINT PARAM Geo.District; };
        };
    }

ELEMENT SearchProd
    {
        GOTO NULL;
        SWITCH (Prod.STATE) {
            CASE S0 { PRINT 'all'; };
            CASE S1 { PRINT PARAM Prod.Category; };
            CASE S2 { PRINT PARAM Prod.SubCat; };
            CASE S3 { PRINT PARAM Prod.Product; };
        };
    };

ELEMENT Matches
    {
        SELECT COUNT(*) INTO Anzahl
            FROM tAnbieter a,tAnbot ab, VAR TempDis hg, VAR TempProd hp
            WHERE a.aGemeinId=hg.aGemeinId AND ab.aPersId=a.aPersId
              AND ab.aProdId=hp.aProdId AND ab.aVerfuegbarkeit=1;
        PRINT Anzahl;
        GOTO NULL;
    };

ELEMENT Show
    {
        PRINT 'Show them';
        GOTO S1 IN Result
            SET PARAM GeoState := Geo.STATE,
            SET PARAM District := PARAM Geo.District,
            SET PARAM Region := PARAM Geo.Region,
            SET PARAM Province := PARAM Geo.Province,
            SET PARAM ProdState := Prod.STATE,
            SET PARAM Product := PARAM Prod.Product,
            SET PARAM SubCat := PARAM Prod.SubCat,
            SET PARAM Category := PARAM Prod.Category
            PAGE Result;
        };

END //Selection
```

Figure A.7: PHC/DL description of PHC Selection.

```
<TR><TD COLSPAN=2 HEIGHT=100 ALIGN=center>
<FONT SIZE=4>You are searching for<BR>
<EM><!--$PHC PRINT(Selection.SearchProd); --></EM>
in the region
<EM><!--$PHC PRINT(Selection.SearchDis); --></EM></FONT><BR>
The database contains <!--$PHC PRINT(Selection.Matches); --> Matches
<!--$PHC PRINT(Selection.Show); -->
<IMG SRC="show_image?pname=yel_arrow" ALT="Show them" HEIGHT=20 WIDTH=30
BORDER=0 ALIGN=MIDDLE>
</TD></TR>
```

Figure A.8: PHC/LL description to PHC/DL from figure A.7.

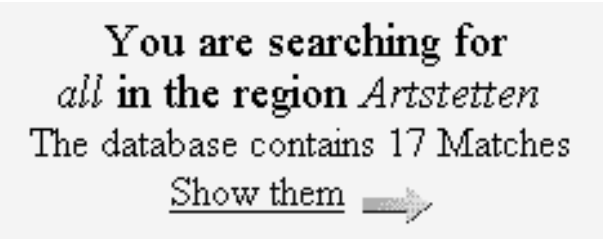


Figure A.9: Browser view of PHC Selection.

```
PHC Result

INTERFACE

PARAM
    GeoState STATE;
    District ROWID;
    Region ROWID;
    Province ROWID;
    ProdState STATE;
    Product ROWID;
    SubCat ROWID;
    Category ROWID;
ELEMENT
    Home, Offer, Address, Counter, Caddy;
LIST
    Result_list;

STATE S0, S1;

IMPLEMENTATION

BEGIN
VAR
    TempDis BAG { District ROWID };
    TempProd BAG { Product ROWID };
    pick INT;
    count INT;

STATE S1
{ // 2 temporary tables or 16 different queries (switch/case)

    SWITCH (GeoState) {
        CASE S0 { // nothing selected
            INSERT INTO VAR TempDis (District)
            SELECT aGemeinId FROM tGemeinde;
            };
        CASE S1 { // province selected
            INSERT INTO VAR TempDis (District)
            SELECT tGemeinde.aGemeinId FROM tGemeinde, tRegion
            WHERE tGemeinde.aRegId = tRegion.aRegId
            AND tRegion.aProvId = PARAM Province;
            };
        CASE S2 { // region selected
            INSERT INTO VAR TempDis (District)
            SELECT tGemeinde.aGemeinId FROM tGemeinde
            WHERE aRegId = PARAM Region;
            };
        CASE S3 { // district selected
            INSERT INTO VAR TempDis (District)
            VALUES (PARAM District);
            };
    };

    SWITCH (ProdState) {
        CASE S0 { // nothing selected
            INSERT INTO VAR TempProd (Product)
            SELECT aProdId FROM tProdukt;
            };
        CASE S1 { // category selected
            INSERT INTO VAR TempProd (Product)
            SELECT tProdukt.aProdId FROM tProdukt, tSubkategorie
            WHERE tProdukt.aSubkatId = tSubkategorie.aSubkatId
            AND tSubkategorie.aKatId = PARAM Category;
            };
        CASE S2 { // subcategory selected
            INSERT INTO VAR TempProd (Product)
            SELECT tProdukt.aProdId FROM tProdukt
            WHERE tProdukt.aSubkatId = PARAM SubCat;
            };
    };
}

LIST Result_List ( // from which to select entries for the shopping basket
    STATE S0 { NULL; };
    STATE S1
        { count:=0; }
        QUERY Result_Query IS
        SELECT a.aPersId, a.aVorname, a.aNachname, a.aPLZ, a.aStrasse,
            g.aBezeichnung AS Ort, ab.aProdId ,ab.aAnbotId, ab.aBezeichnung,
            ab.aVerfuegbarkeit
        FROM wAnbieter a, tGemeinde g, tAnbot ab,
            VAR TempDis hg, VAR TempProd hp
        WHERE a.aGemeinId=hg.aGemeinId AND ab.aPersId=a.aPersId
            AND ab.aProdId=hp.aProdId AND a.aGemeinId=g.aGemeinId
            AND ab.aVerfuegbarkeit=1 ORDER BY a.aNachname;
        (
        ELEMENT Counter {
            count:=count+1;
            PRINT count;
            GOTO NULL;
        }
        ELEMENT Home {
            PRINT Result_Query.aVorname||' '||Result_Query.aNachname;
            GOTO IN Show_Home
                SET PARAM PersId := Result_Query.aPersId
                PAGE Home;
        }
        ELEMENT Offer {
            PRINT Result_Query.aBezeichnung;
            GOTO IN Show_Info
                SET PARAM AnbotId := Result_Query.aAnbotId
                PAGE Info;
        }
        ELEMENT Address {
            PRINT Result_Query.Ort;
            GOTO NULL;
            // GOTO S0 IN VillageInfo SET PARAM Village := Result_Query.aPLZ;
        }
        ELEMENT Caddy {
            pick:=0;
            SELECT Anzahl INTO pick FROM PARAM Basket.Offers
                WHERE AnbotId = Result_Query.aAnbotId;
            IF (pick == 0) {
                IMAGE einkauf;
                MESSAGE Basket.fill (pAnbotId => Result_Query.aAnbotId)
                    PAGE THIS
                    REFRESH Shopping_Basket;
            }
            ELSE {
                IMAGE einkauf2;
                MESSAGE Basket.remove (pAnbotId => Result_Query.aAnbotId)
                    PAGE THIS
                    REFRESH Shopping_Basket;
            }
            PRINT NULL;
        }
    )
)

END //Result
```

Figure A.10: PHC/DL description of PHC Result.

```
<TABLE   Border=0>
<!--$PHC FOR row IN Result.Result_List LOOP -->
<TR ALIGN="left">
<TD VALIGN=top >
       <!--$PHC PRINT(row.Counter); -->
       <img src="show_image?pname=left" WIDTH=20 HEIGHT=11 BORDER=0>
</TD>
<TD>
       <!--$PHC PRINT(row.Offer); -->
       <BR>
       <!--$PHC PRINT(row.Home); -->, <!--$PHC PRINT(row.Address); -->
       <BR>
       <IMG SRC="show_image?pname=cbar-short" ALIGN="center" WIDTH=300 HEIGHT=2 BORDER=0>
</TD>
<TD>
       <!--$PHC PRINT(row.Caddy); -->
</TD>
</TR>
<!--$PHC END LOOP; -->
</TABLE>
```

Figure A.11: PHC/LL description to PHC/DL from figure A.10.

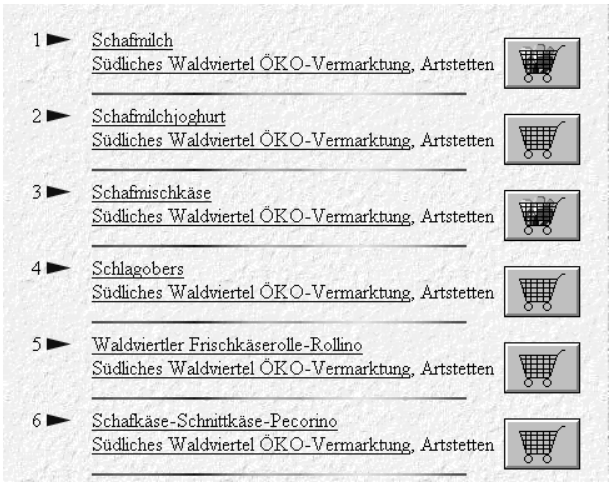


Figure A.12: Browser view of PHC Result.

```
PHC Basket

INTERFACE

PARAM
    Waren BAG {aAnbotId ROWID, aAnzahl INT};
...

IMPLEMENTATION

METHOD fill(pAnbotId ROWID DEFAULT NULL)
VAR
    check_anz INTEGER;
BEGIN
    SELECT COUNT(*) INTO check_anz FROM PARAM Waren WHERE Waren.aAnbotId=pAnbotId;
    IF (check_anz == 0) then // not yet contained
        INSERT INTO PARAM Waren (aAnbotId,aAnzahl)
            VALUES(pAnbotId,1);
END;

METHOD remove(pAnbotId ROWID DEFAULT NULL)
BEGIN
    DELETE FROM PARAM Waren WHERE Waren.aAnbotId=pAnbotId;
END;

METHOD change(pAnbotId ROWID DEFAULT NULL, pAnzahl INT DEFAULT 1)
BEGIN
    UPDATE PARAM Waren hw SET aAnzahl=pAnzahl WHERE hw.aAnbotId=pAnbotId;
END;

BEGIN
...
END //Basket
```

Figure A.13: PHC/DL description of PCH Basket, method part.

Appendix B

PHC Language Syntax Definition

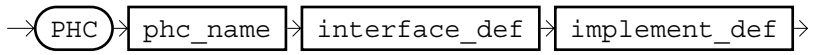
This appendix summarizes the grammar rules of PHC/DL and PHC/LL. A detailed description of both languages and their implementation can be found in [Fal98]. Information concerning language design, Backus Naur Form (BNF) and compiler implementation can be found in [Sch85, ASU88]. As a convention the non-terminal symbols consist of lower-case characters and the keywords of upper-case characters. Identifiers have the suffix `_name` and constants have the suffix `_const`. The initial symbol is the symbol `start` in rule D1.

B.1 Design Language

The following rules show the syntax of PHC/DL:

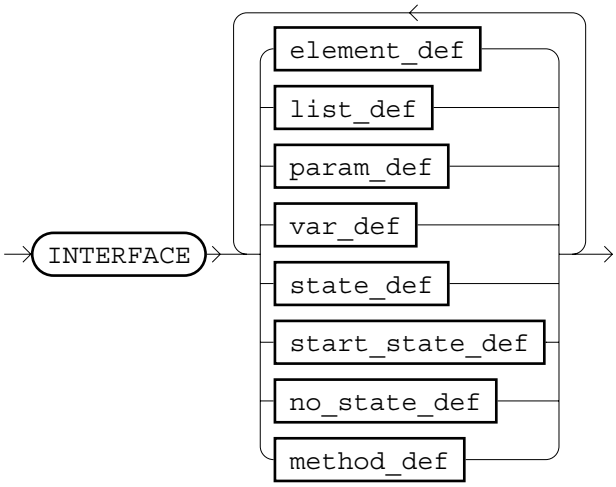
D1

`start`



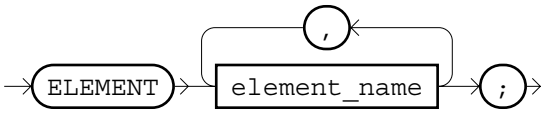
D2

interface_def



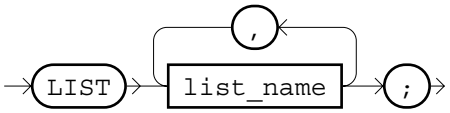
D3

element_def



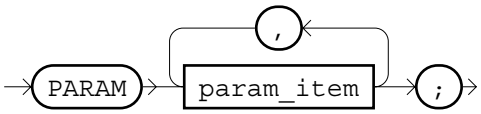
D4

list_def



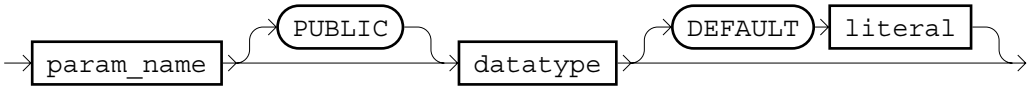
D5

param_def



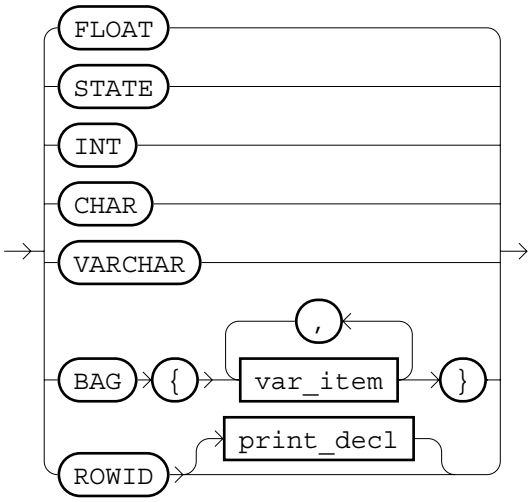
D6

param_item



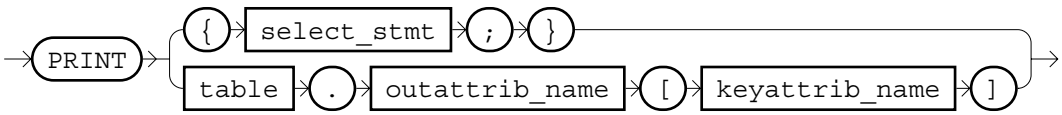
D7

datatype



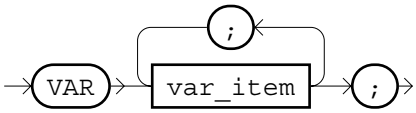
D8

print_decl



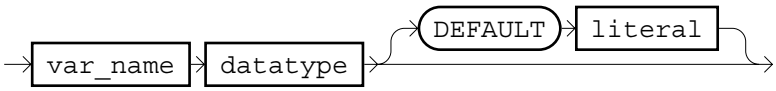
D9

var_def



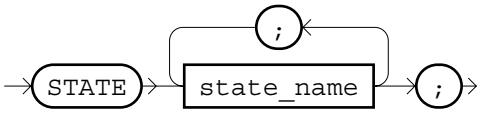
D10

var_item



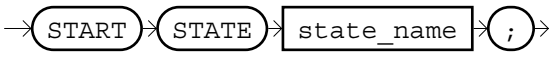
D11

state_def



D12

start_state_def

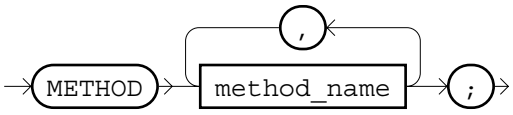


D13

no_state_def

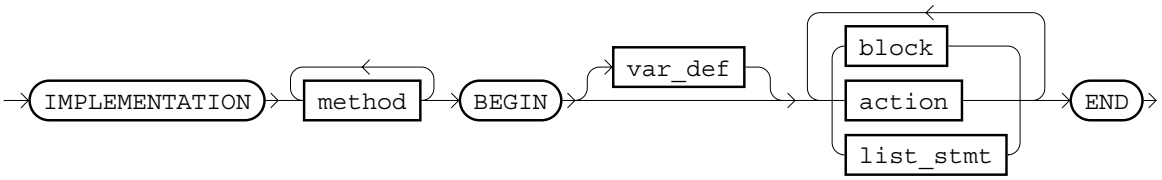
D14

method_def



D15

implement_def

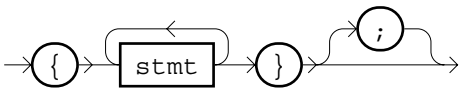


D16

method

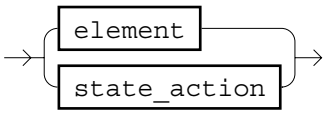
D17

block



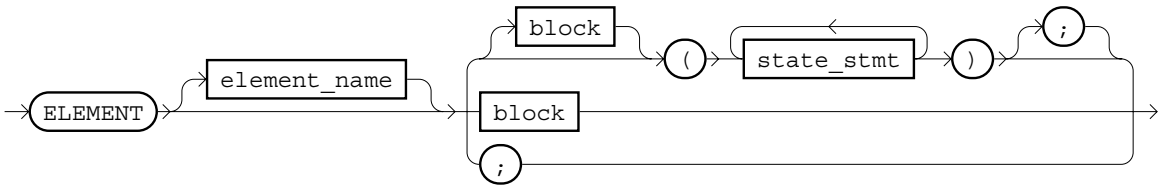
D18

action



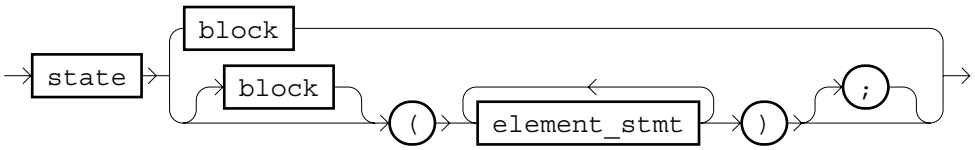
D19

element



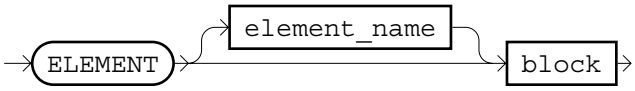
D20

state_action



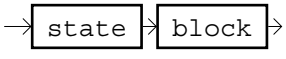
D21

element_stmt



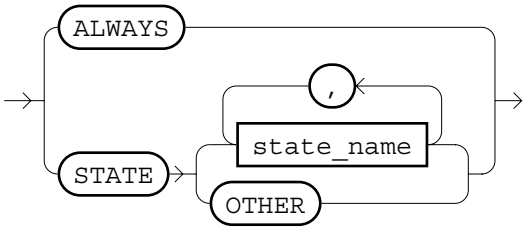
D22

state_stmt



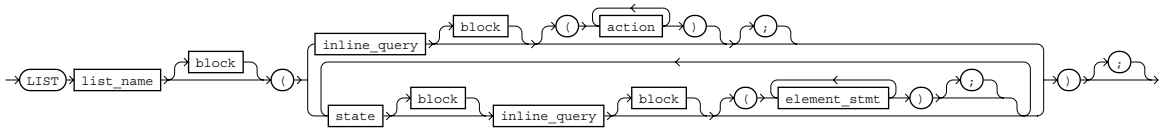
D23

state



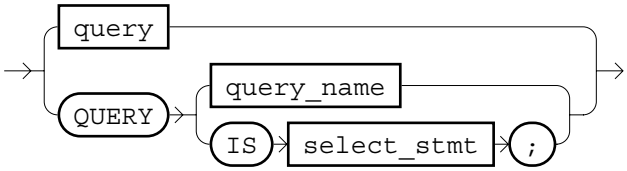
D24

list_stmt



D25

inline_query

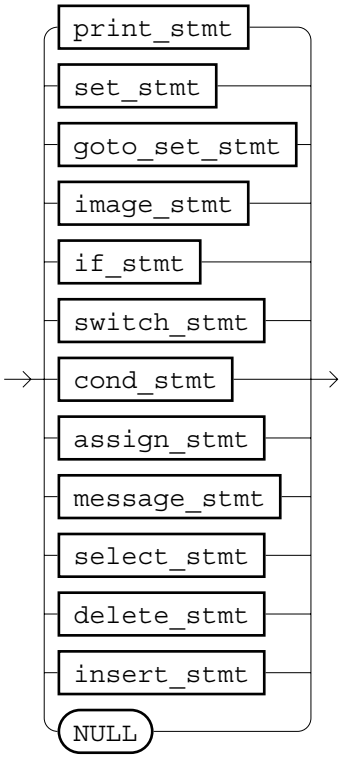


D26

query

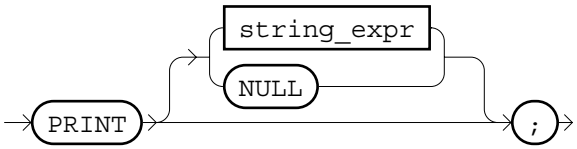
D27

stmt



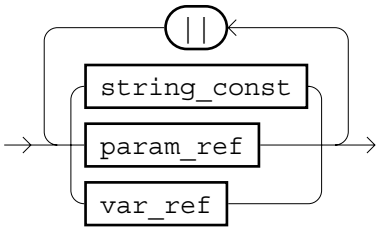
D28

print_stmt



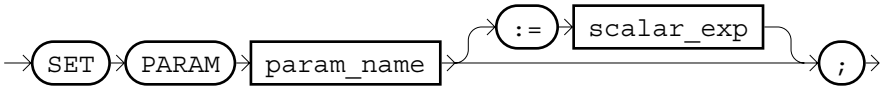
D29

string_expr



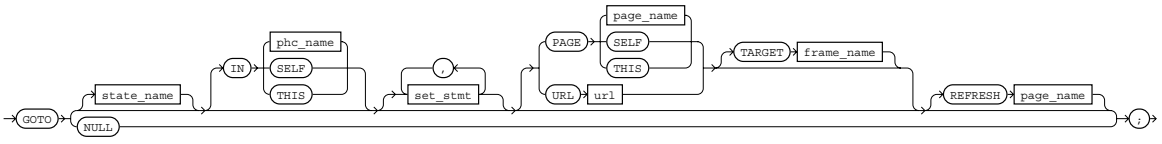
D30

set_stmt



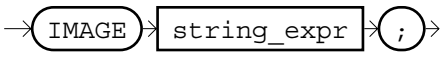
D31

goto_set_stmt



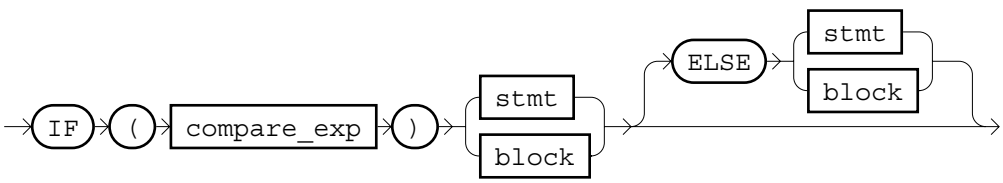
D32

image_stmt



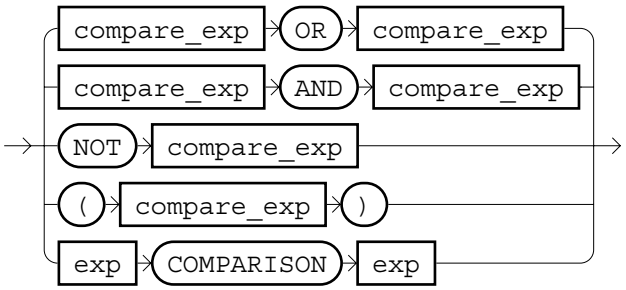
D33

if_stmt



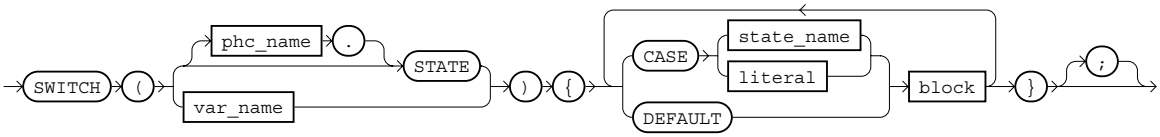
D34

compare_exp



D35

switch_stmt

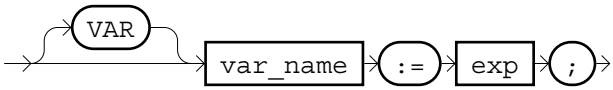


D36

cond_stmt

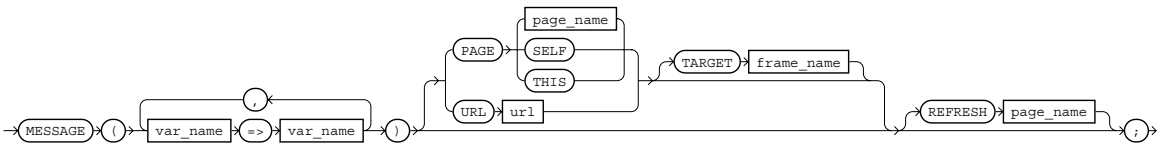
D37

assign_stmt



D38

message_stmt

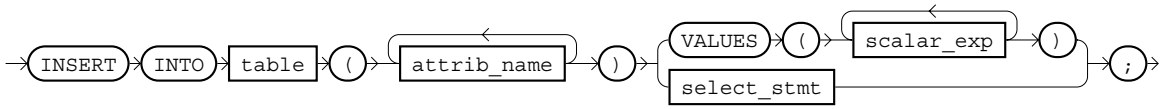


D39

delete_stmt

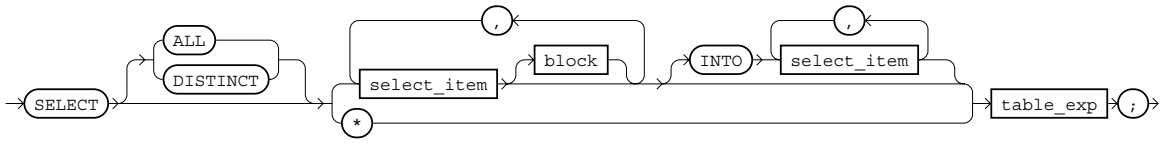
D40

insert_stmt



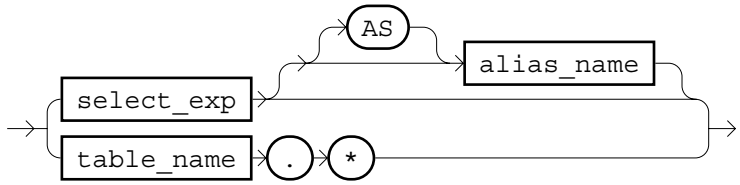
D41

select_stmt



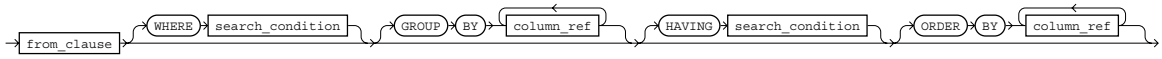
D42

select_item



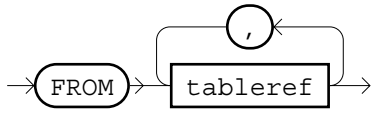
D43

table_exp



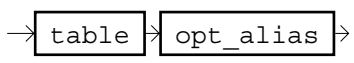
D44

from_clause



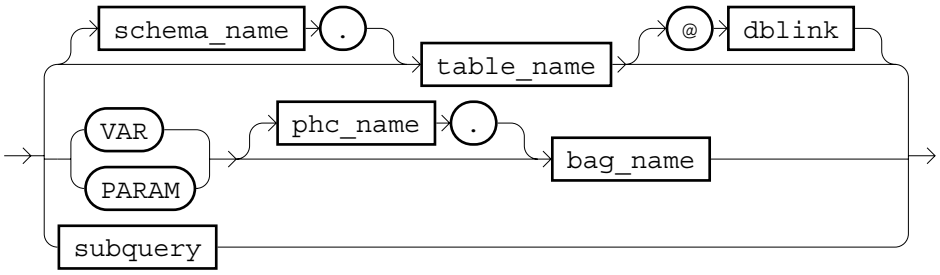
D45

tableref



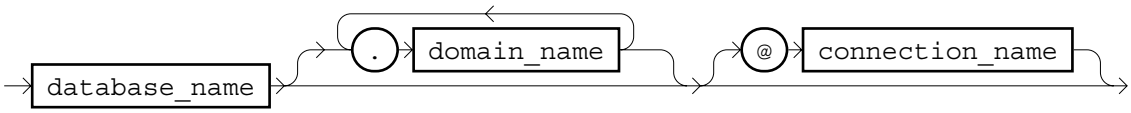
D46

table



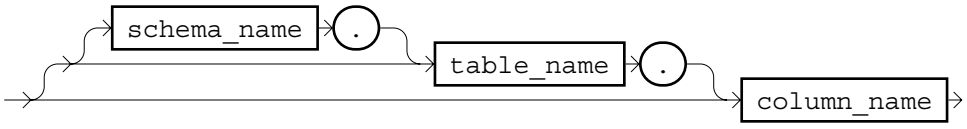
D47

dblink



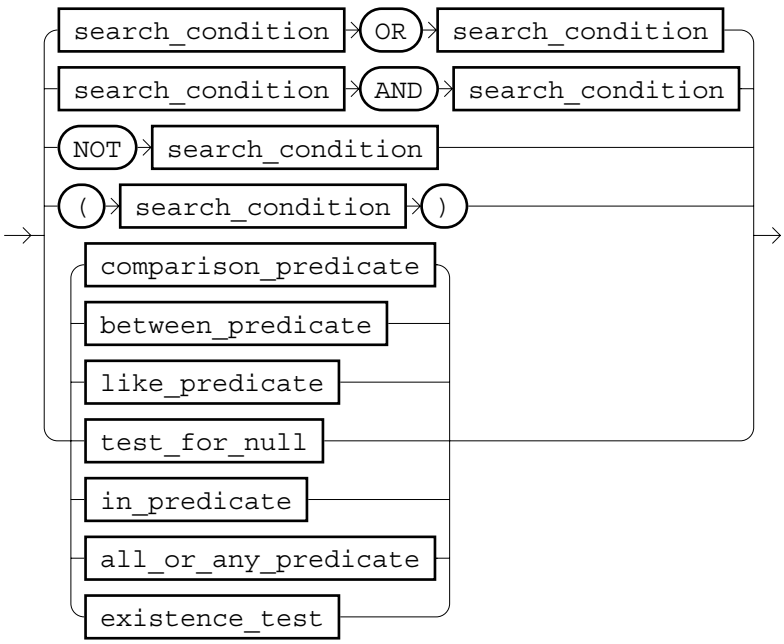
D48

column_ref



D49

search_condition

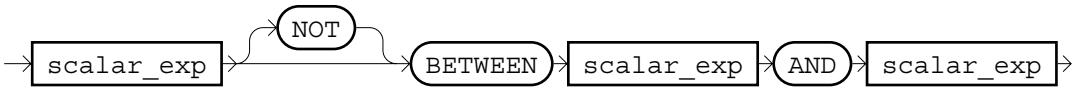


D50

comparison_predicate

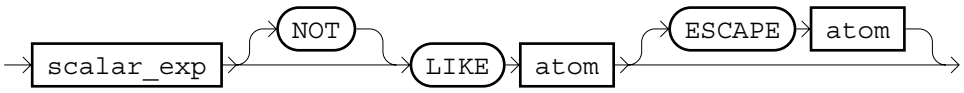
D51

between_predicate



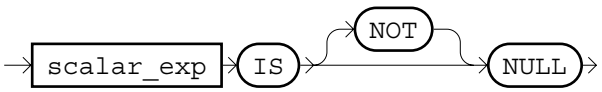
D52

like_predicate



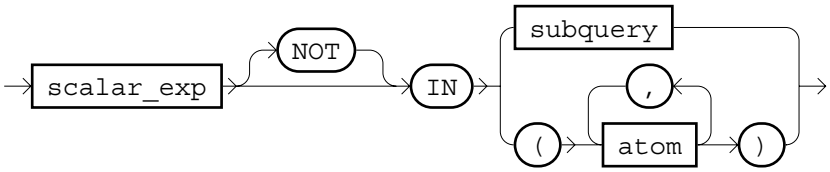
D53

test_for_null



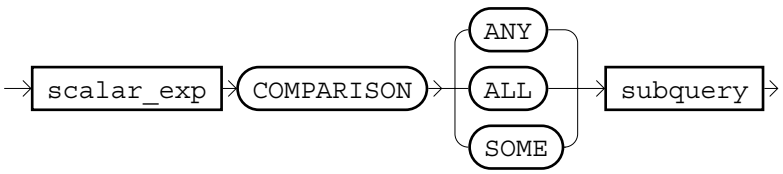
D54

in_predicate



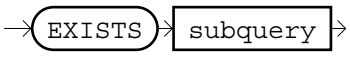
D55

all_or_any_predicate



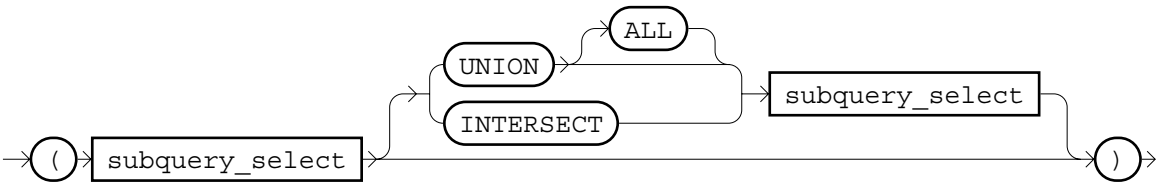
D56

existence_test



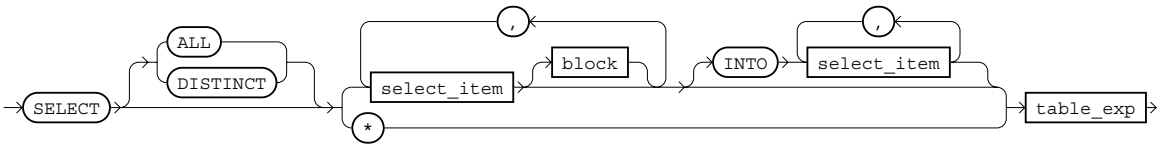
D57

subquery



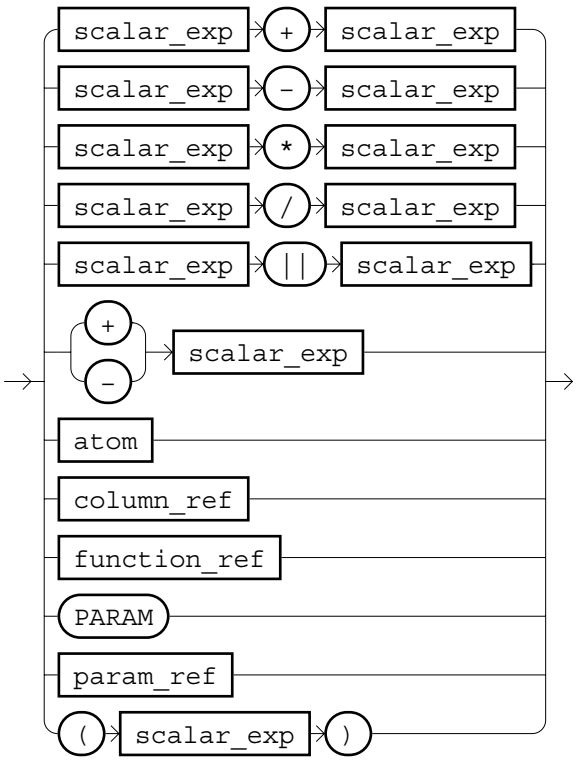
D58

subquery_select



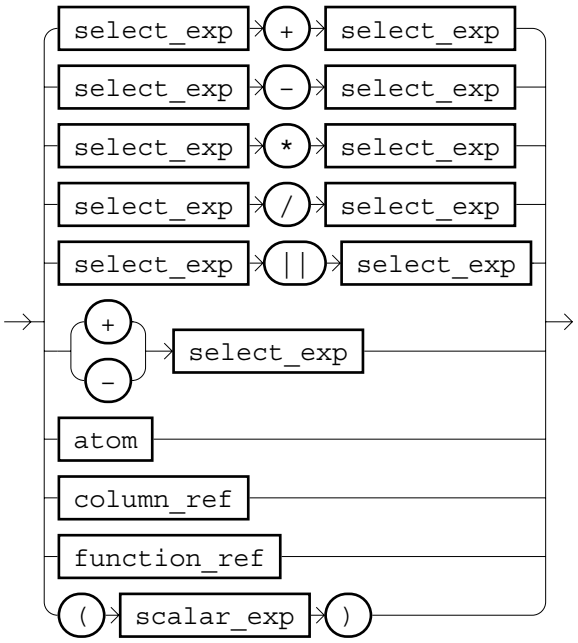
D59

scalar_exp



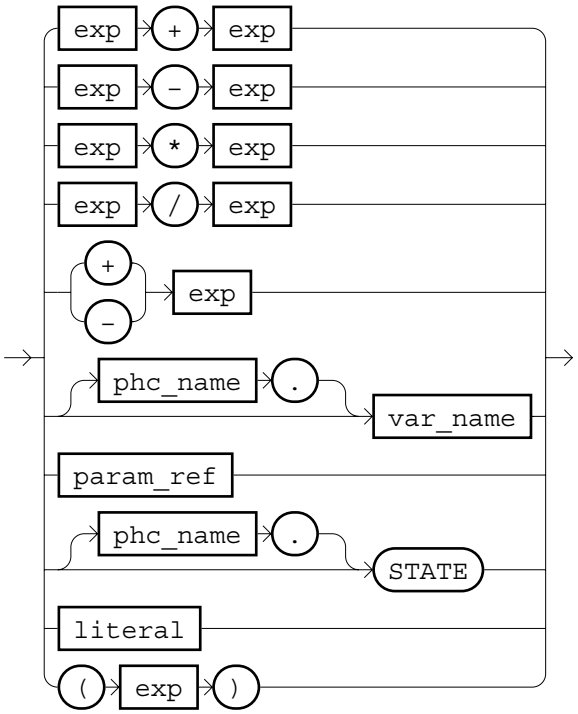
D60

select_exp



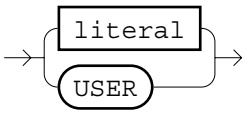
D61

exp



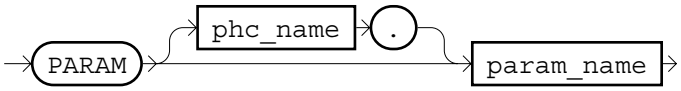
D62

atom



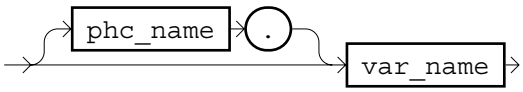
D63

param_ref



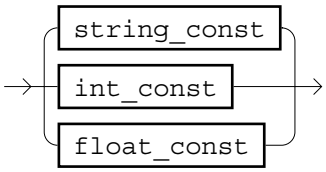
D64

var_ref



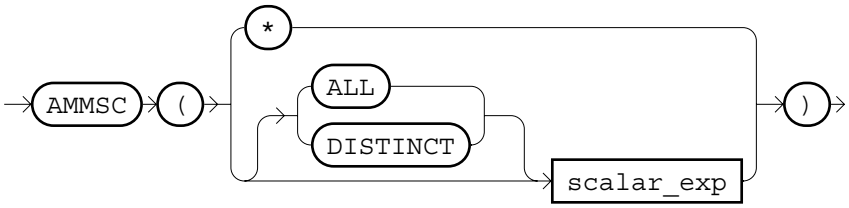
D65

literal



D66

function_ref



D67

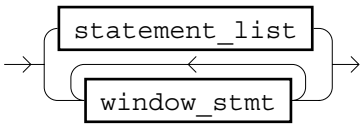
url

B.2 Layout Language

The following rules show the syntax diagrams of PHC/LL:

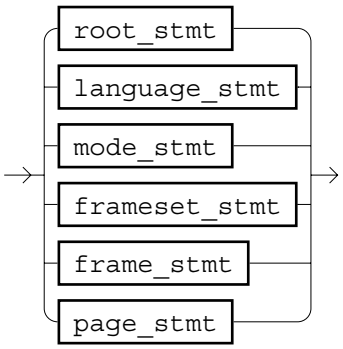
L1

start



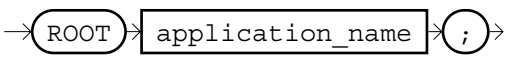
L2

window_stmt



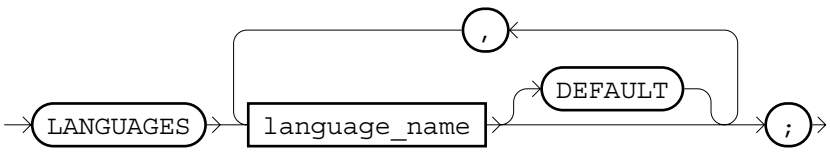
L3

root_stmt



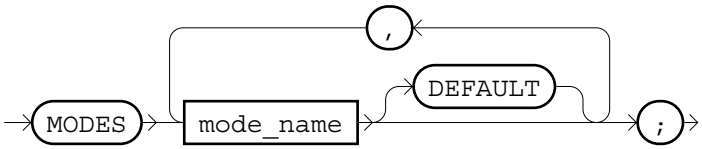
L4

language_stmt



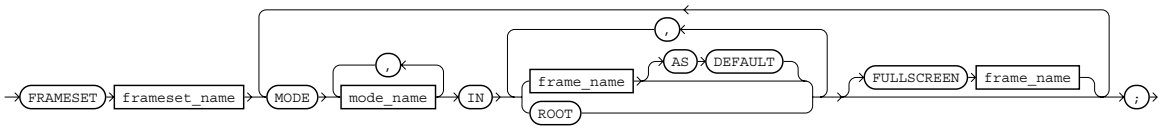
L5

mode_stmt



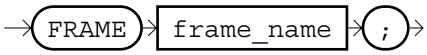
L6

frameset_stmt



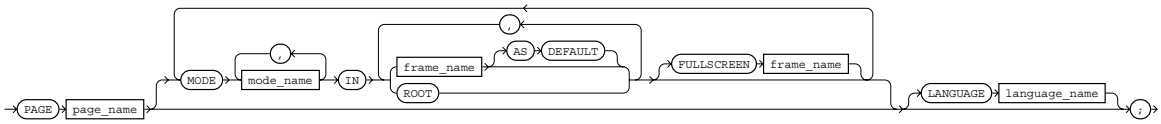
L7

frame_stmt



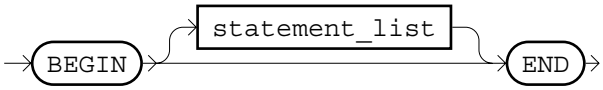
L8

page_stmt



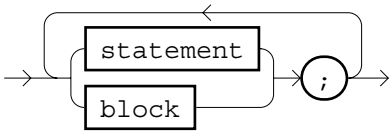
L9

block



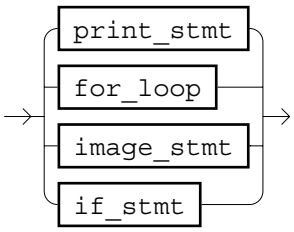
L10

statement_list



L11

statement



L12

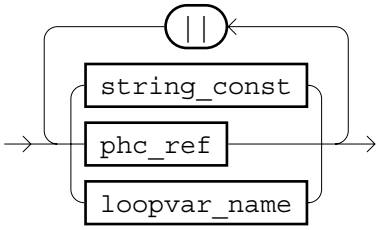
print_stmt

L13

image_stmt

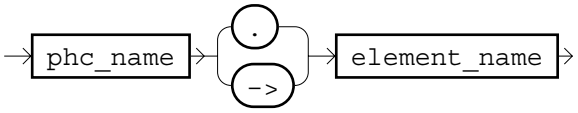
L14

string_exp



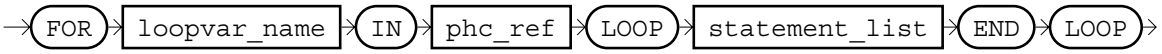
L15

phc_ref



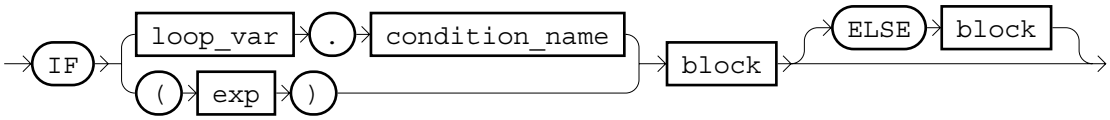
L16

for_loop



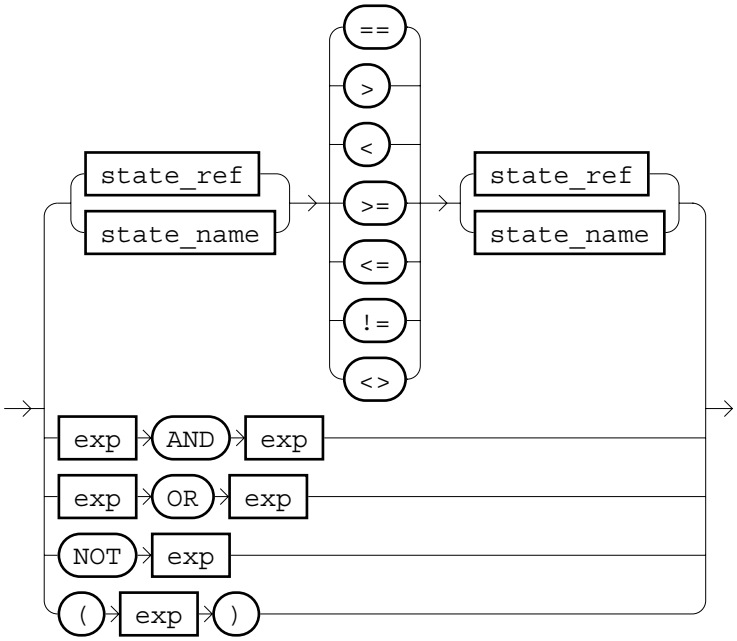
L17

if_stmt



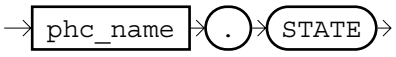
L18

exp



L19

state_ref



List of Figures

List of Figures

Bibliography

[AB87] Atkinson, M.; Bunemann, O.: *Types and persistence in database programming languages.* ACM Computing Surveys, Volume 19, Nr. 2, pp. 105-190, Juni 1987.

[Adi97] Adida, B.: *It all starts at the server.* IEEE Internet Computing, Vol. 1, Nr. 1, pp. 75-77, 1997.

[Adi97a] Adida, B.: *Taking Web clients to the next level.* IEEE Internet Computing, Vol. 1, Nr. 2, pp. 65-67, 1997.

[Adi97c] Adida, B.: *Database-backed Web sites.* IEEE Internet Computing, Vol. 1, Nr. 6, pp. 78-80, 1997.

[AM95] Atkinson, M.; Morrison, R.: *Orthogonally persistent object systems.* The VLDB Journal, Volume 4, Nr. 3, pp. 319-401, Juli 1995.

[AM98] Arocena, G.; Mendelzon, A.: *Viewing Web Information Systems as Database Applications.* Communications of the ACM, Vol.41, No.7, July 1998.

[ASU88] Aho, A.; Sethi, R.; Ullman, J.: *Compilerbau.* Addison-Wesley, 1988.

[BBB+95] Bergner, K.; Bartsch, W.; Braun, P.; Molterer, S.; Teubner, G.: *Pflegeleicht: Einbindung relationaler Datenbanken ins Web.* iX Mutliuser-Multitasking Magazin, November 1995.

[BDK92] Bancilhon, F.; Delobel, C.; Kanellakis, P. (Editor): *Building an Object-Oriented Database System - The Story of O_2.* Morgan Kaufmann, San Mateo, CA, 1992.

[Beh98] Behme, H.: *Hilfe für die Verwaltung von WWW–Inhalten.* iX Mutliuser-Multitasking Magazin, Juni 1998.

[Boo94] Booch, G.: *Object-oriented Analysis and Design. With Applications.* Benjamin/Cummings Publishing, 2nd edition, 1994.

[Bor98] Born, A.: *Chancenlos: Kein Durchbruch für NCs.* iX Mutliuser-Multitasking Magazin, August 1998.

[BPS94] Blaha, M.; Premerlani, W.; Shen, H.: *Converting OO Models into RDBMS Schema.* IEEE Software, pp. 28-39, May 1994.

[Bra97] Bradley, N.: *The concise SGML companion.* Addison-Wesley Longman, 1997.

[Bru98] Brutzman, D.: *The Virtual Reality Modeling Language and Java.* Communications of the ACM, Vol.41, No.6, June 1998.

[BS98] Barta, R.A.; Schranz, M.W.: *JESSICA: an object-oriented hypermedia publishing processor.* 'Proceedings of the 7th International World Wide Web Conference' in Computer Networks and ISDN Systems, Volume 30, Numbers 1-7, pp. 281-290, Elsevier, 1998.

[BSW98] Beutelsbacher, A.; Schwenk, J.; Wolfenstetter, K.: *Moderne Verfahren der Kryptographie.* vieweg, Braunschweig, 2. Auflage, 1998.

[BW98] Behbehani, A.; Wartala, R.: *VRML–Welten dynamisch generieren.* iX Mutliuser-Multitasking Magazin, August 1998.

[Cat94] Cattell, R. (Editor): *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, CA, 1994.

[Cat96] Cattell, R.G.G., Editor: *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[Cat97a] Cattell, R.G.G., Editor: *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[Cat97] Catarci, T.: *Interacting with Databases in the Global Information Infrastructure.* IEEE Communications Magazine, Vol. 35, Nr. 5, pp. 72-76, 1997.

[CDK94] Coulouris, G.; Dollimore, J.; Kindberg, T.: *Distributed Systems.* Addison-Wesley, 1994.

[CH98] Caglayan, A.; Harrison, C.: *Intelligente Software-Agenten.* Carl Hanser Verlag München Wien, 1998.

[Che76] Chen, P.: *The Entity-Relationship Model - Toward a Unified View of Data.* ACM Transactions on Database Systems, Volume 1, Nr. 1, pp. 9-36, 1976.

[Cho98] Chorafas, D.N.: *Agent Technology Handbook.* McGraw-Hill, 1998.

[Cla97] Clark, D.: *CISCO connect online: It's Good for Business.* IEEE Internet Computing, Vol. 1, Nr. 6, pp. 55-58, 1997.

[Cod70] Codd, E.F.: *A relational model for large shared data banks.* Communications of the ACM, Volume 13, Nr. 6, pp.377-387, 1970.

[Con98] Connolly, D. (Interview): *Architecture of the Web.* IEEE Internet Computing, Vol.2, No.2, March/April 1998.

[DaD97] Date, C.J.; Darwen, H.: *A Guide to the SQL Standard.* Addison-Wesley, Reading, MA, 4th ed, 1997.

[Dat87] Date, C.: *A Guide to INGRES.* Addison-Wesley, Reading, MA, 1987.

[Dav95] Davis, N.: *Telematics in Education: The UK Case.* In: Veen, W.; Collis, B.; et al.: Telematics in Education: The European Case, Academic Book Centre, ABC, de Lier, the Netherlands, 1995.

[DD95] Demuth, F.; Dierks, J.: *Entscheidungskriterien zur Auswahl von 4GL-Systemen.* iX Mutliuser-Multitasking Magazin, S. 38-51, Jänner 1995.

[Dic97] Dicken, H.: *Formel SQL: Performance von Datenbankabfragen aus Java.* iX Mutliuser-Multitasking Magazin, Dezember 1997.

[Die98] Diercks, J.: *Am Anfang war das BLOB: Datenbanksysteme mit neuen Fähigkeiten.* iX Mutliuser-Multitasking Magazin, August 1998.

[EGH+92] Engels, G.; Gogolla, M.; Hohenstein, U.; Hülsmann, K.; Löhr-Richter, P.; Saake, G.; Ehrich, H.-D.: *Conceptual modelling of database applications using an extended ER model.* Data & Knowledge Engineering, North-Holland, Volume 9, Nr. 2, pp. 157-204, 1992.

[EN94] Elmasri, R.; Navathe, S.: *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, CA, 2.Auflage, 1994.

[ES97] Ensor, D.; Stevenson, I.: *Oracle8 Design Tips.* O'Reilly 1997.

[Eva98] Evans, D.: *The OQL Standard Emerges.* Byte Magazine, March 1998.

[Fal98] Falb, J.: *A State Machine Based Generator for Database Powered Web Applications.* Diploma Thesis at the Institute of Computer Technology of the Vienna University of Technology, Vienna 1998.

[Fla97] Flanagan, D.: *JavaScript: The Definitive Guide.* O'Reilly, 2nd edition, 1997.

[FW94] Field, M.; Weedon, R.: *Professional training in computing: The UK Open University·s Computing for Commerce and Industry Programme.* In: Open and Distance Learning - Critical Success Factors, International Conference, Geneva, 10-12 Oct. 1994, Proceedings, Berne 1995.

[Gas95] Gastkemper, F.: *Pedagogy.* In: Open and Distance Learning - Critical Success Factors, International Conference, Geneva, 10-12 Oct.1994, Proceedings, Berne 1995.

[GJM91] Ghezzi, C.; Jazayeri, M.; Mandrioli, D.: *Software Engineering.* Prentice-Hall, 1991.

[GMM98] Guttman, R.; Moukas, G.; Maes, P..: *Agent-mediated Electronic Commerce: A Survey.* Knowledge Engineering Review [⇒MIT], June 1998.

[Gog94] Gogolla, M..: *An extended Entity Relationship Model. Fundamentals and Pragmatics.* Lecture Notes in Computer Science, Band 767. Springer-Verlag, Berlin, 1994.

Bibliography

[Gra96] Graham, I.: *The HTML Sourcebook.* Wiley & Sons, Inc., New York, 2nd edition 1996.

[Gra97] Graham, I.: *The HTML Sourcebook.* Wiley & Sons, Inc., New York, 3rd edition 1997.

[Gre97] Greenspun, P.: *Database Backed Web Sites.* Ziff-Davies Press, Macmillan Computer Publishing, Emeryville, CA, USA, 1997.

[Gre97a] Greenwald, R.: *Using Oracle Web Application Server 3.* QUE, 1997.

[Gre98] Grehan, R.: *Object Marries Relational.* Byte Magazine, March 1998.

[GR95] Gabriel, R.; Röhrs, H.: *Datenbanksysteme: Konzeptionelle Datenmodellierung und Datenbankarchitekturen.* Springer-Verlag, Berlin, 1995.

[GSS94] Gottlob, G.; Schrefl, M.; Stumptner, M.: *Datenbanksysteme: Skriptum zur Vorlesung.* Institut für Informationssysteme, Abteilung für Datenbanken und Expertensysteme, Technische Universität Wien, 1994.

[Hal98] Halter, C.: *Durchsuchen Datenbank-basierter Web-Applikationen mit Robots und Agents.* Diploma Thesis at the Institute of Computer Technology of the Vienna University of Technology, Vienna 1998.

[Hal97] Hall, B.: *Web-Based Training Cookbook.* John Wiley & Sons, 1997.

[Har74] Harary, F.: *Graphentheorie.* R.Oldenburg Verlag München Wien, 1974.

[HE92] Hohenstein, U.; Engels, G.: *SQL/EER: Syntax and Semantics of an Entity-Relationship-Based Query Language.* Information Systems, Volume 17, Nr. 3, pp. 209-242, 1992.

[Heu97] Heuer, A.: *Objektorientierte Datenbanken - Konzepte, Modelle, Standards und Systeme.* Addison-Wesley, 2., aktualisierte und erweiterte Auflage, 1997.

[HeuS95] Heuer, A.; Saake, G.: *Datenbanken: Konzepte und Sprachen.* International Thomson Publishing, Bonn, 1995.

[HeuS97] Heuer, A.; Saake, G.: *Datenbanken: Konzepte und Sprachen.* International Thomson Publishing, Bonn, 1.korrigierter Nachdruck, 1997.

[HFPH98] Heuer, A.; Flach, G.; Post, K.; Hein, O.: *Jasmine: OO-Datenbank für multimediale Anwendungen.* iX Mutliuser-Multitasking Magazin, August 1998.

[Hir95] Hirohido, H.: *Experimental Analysis of User's Behaviour in Hypermedia CAI Systems.* In: Liberating the Learner, WCCE 95. Proceedings, Birmingham 1995.

[HNS86] Hohenstein, U.; Neugebauer, L.; Saake, G.: *An Extended Entity Relationship Model for Non-Standard Databases.* In: Proc. Workshop „Relationale Datenbanken“, Bericht Nr. 3-86, S. 185-211. Lessach, 1986.

[HNSE87] Hohenstein, U.; Neugebauer, L.; Saake, G.; Ehrich, H.-D.: *Three-Level Specification of Databases Using an Extended Entity Relationship Model.* In: Proc. GI-Fachtagung „Informationsermittlung und -analyse für den Entwurf von Informationssystemen“, Informatik-Fachberichte, Band 143, S. 58-88. Springer-Verlag, Berlin, 1991.

[Hoh93] Hohenstein, U.: *Formale Semantik eines erweiterten Entity-Relationship-Modells.* Teubner-Verlag, Stuttgart, Leipzig, 1993.

[Hol98] Holzner, S.: *XML complete.* McGraw-Hill, New York, 1998.

[HU96] Hopcroft, J.; Ullman, J.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley, 3., korrigierte Auflage 1994 / 1., korrigierter Nachdruck, 1996.

[Hub98] Hubertz, J.: *Ins Allerheiligste: AS400 sicher im Internet.* iX Mutliuser-Multitasking Magazin, Jänner 1998.

[IBV98] Isakowitz, T.; Bieber, M.; Vitali, F. (guest editors): *Web Information Systems.* Special Section in Communications of the ACM, Vol.41, No.7, July 1998.

[Int89] International Organization for Standardization (ISO): *Database Language SQL.* Document ISO/IEC 9075:1989, 1989.

[Int92] International Organization for Standardization (ISO): *Database Language SQL.* Document ISO/IEC 9075:1992, 1992.

[ISB95] Isakowitz, T.; Stohr, E.; Balasubramanian, P.: *RMM: A Methodology for Structured Hypermedia Design.* Communications of the ACM, Vol. 38, Nr.8, pp. 34-44 1995.

[ISO95] International Organization for Standardization (ISO) - American National Standards Institute (ANSI): *Working Draft Database Language SQL (SQL/Foundation SQL3).* Part 2, X3H2-94-080 and SOU-003, 1995.

[Iye97] Iyengar, A.: *Dynamic Argument Embedding: Preserving State on the World Wide Web.* IEEE Internet Computing, Vol. 1, Nr.2, pp. 50-56 1997.

[Jel98] Jelliffe, R.: *The XML and SGML Cookbook.* Prentice-Hall 1998.

[JH98] Jones Jr., I.R.; Heuring, V.P..: *Modeling and simulating optical computing architectures.* Systems Implementation 2000, Chapman & Hall, IFIP 1998.

[Joh95] Johnson, R.: *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley 1995.

[Kam98] Kampichler, W.: *Performance Analysis of Database Backed Web Applications.* Diploma Thesis at the Institute of Computer Technology of the Vienna University of Technology, Vienna 1998.

[KDM98] Khoshafian, S.; Dasananda, S.; Minassian, N.: *The Jasmine Object Database.* Morgan Kaufmann Publishers, 1998.

[KM97] Kristol, D.; Montulli, L.: *HTTP State Management Mechanism.* draft-ietf-http-state-man-mec-02.ps on [⇒W3C], 1997.

[KZ98] Kiniry, J.; Zimmerman, D.: *A Hands-on Look at Java Mobile Agents.* IEEE Internet Computing, Vol. 1, Nr. 4, pp. 21-30, 1998.

[LLOW91] Lamb, C.; Landis, G.; Orenstein, J.; Winreb, D.: *The ObjectStore Database Systems.* Communications of the ACM, Volume 34, Nr. 10, pp. 50-63, 1991.

[LV96] Lausen, G.; Vossen, G.: *Objekt-orientierte Datenbanken: Modelle und Sprachen.* Oldenburg, München, 1996.

[Mer97] Merkle, B.: *In die Ferne schweifen: Verteilte Java-Objekte mit RMI.* iX Mutliuser-Multitasking Magazin, Dezember 1997.

[Mai83] Maier, D.: *The Theory of Relational Databases.* Computer Science Press, Rockville, MD, 1983.

[MD92] McGoveran, D.; Date, C.J..: *A Guide to Sybase and SQL Server.* Addison-Wesley, Reading, MA, 1992.

[Mis95] Misgeld, W.: *SQL: Einstieg und Anwendung.* Carl Hanser Verlag, München, 1995.

[MP97] Musella, D.; Padula, M.: *Step by Step Toward the Glovbal Internet Library.* IEEE Communications Magazine, Vol. 35, Nr. 5, pp. 64-70, 1997.

[MS98] Manninger, M.; Schischka, R.: *Adapting an Electronic Purse for Internet Payments.* Information Security and Privacy, Lecture Notes in Computer Science, Vol. 1438, pp. 205-214, Springer, Berlin, 1998.

[Mus97] Musciano, C. and Kennedey, B.: *HTML: Das umfassende Referenzwerk.* O'Reilly, Köln 1997.

[Obj93] Object Design Inc.: *ObjectStore User Guide, Release 3.1.* December 1993.

[Oin98] Oinas–Kukkonen, H.: *What is Inside a Link?* Communications of the ACM, Vol.41, No.7, July 1998.

[Ove97] Overbeck, J.: *Objektorientierte Systeme und relationale Datenbanken.* Lecture notes at the Vienna University of Technology, Institute of Information Systems, 1997.

[Pet91] Petković, D.: *INFORMIX. Das relationale Datenbanksystem mit INFORMIX OnLine.* Addison-Wesley, Bonn, 1991.

[Pet92] Petković, D.: *INGRES. Das relationale Datenbanksystem mit Knowledge-Base und Object-Base.* Addison-Wesley, Bonn, 1992.

[Rad98] Radinger, W.: *Object Oriented Approaches for Database backed Web Applications Implemented in Java.* Diploma Thesis at the Institute of Computer Technology of the Vienna University of Technology, Vienna 1998.

[RCK98] Rada, R.; Cargill, C.; Klensin, J.: *Consensus and the Web.* Communications of the ACM, Vol.41, No.7, July 1998.

[Ren95] Renwick, W. L.: *Organisational Strategies.* Open and Distance Learning - Critical Success Factors, International Conference, Geneva, 10-12 Oct.1994, Proceedings, Berne 1995.

[Ros98] Rosenberg, D.: *Bringing Java to the Enterprise: Oracle on Its Java Server Strategy.* IEEE Internet Computing, Vol.2, No.2, March/April 1998.

[Rut98] Rutkowski.: *Dimensioning the Internet.* IEEE Internet Computing, Vol.2, No.2, March/April 1998.

[Saa93] Saake, G.: *Objektorientierte Spezifikation von Informationssystemen.* Habilitationsschrift, Teubner-Verlag, Stuttgart/Leipzig, 1993.

[Sau92] Sauer, H.: *Relationale Datenbanken: Theorie und Praxis inklusive SQL-2.* Band 2. Addison-Wesley, 1992.

[Say97] Sayegh, M.: *Corba: Standard, Spezifikation, Entwicklung.* O'Reilly, Köln, 1997.

[Sch97] Schader, M.: *Objektorientierte Datenbanken. Die C++-Anbindung des ODMG-Standards* Springer-Verlag, Berlin, 1997.

[Sch85] Schreiner, A.; Friedman, G.: *Compiler bauen mit UNIX – Eine Einführung.* Carl Hanser Verlag München Wien, 1985

[SFC98] Stotts, D.P.; Furuta, R.; Cabarrus, C.R.: *Hyperdocuments as Automata: Verification of Traces-Based Browsing Properties by Model Checking.* ACM Transactions on Information Systems, Vol. 16, No. 1, pp. 1-30, January 1998.

[Smi98] Smith, R.: *Internet-Kryptographie* Addison-Wesley Longman, 1998.

[SST97] Saake, G.; Schmitt, I.; Türker, C.: *Objektdatenbanken* International Thomson Publishing, Bonn, 1997.

[Stü93] Stürner, G.: *Oracle 7. Die verteilte semantische Datenbank.* dbms publishing, Weissach, 2.Auflage, 1993.

[Stü96] Stürner, G.: *Oracle 7. Die verteilte semantische Datenbank. Release 7.3.* dbms publishing, Weissach, 4.Auflage, 1996.

[SW74] Steinbuch, K.; Weber, W.: *Taschenbuch der Informatik – Band I.* Springer Verlag, 3.Auflage, 1974.

[Tak98] Takahashi, K.: *Metalevel Links: More Power to Your Links.* Communications of the ACM, Vol.41, No.7, July 1998.

[Tan96] Tanenbaum, A.S.: *Computer Networks.* Prentice-Hall, 3rd edition, 1996.

[Tha95] Thalheim, B.: *Fundamentals of Entity-Relationship Modeling.* Springer-Verlag, Berlin, 1995.

[TB90] Trost, H.; Buchegger, E.: *Datanbak-DIALO: how to communicate with your database in German (and enjoy it).* Austrian Research Institute for Artificial Intelligence, Butterworth-Heinemann, 1990.

[Tol96] Tolksdorf, R.: *Die Sprache des Web: HTML3.* dpunkt Verlag, Heidelberg, 2.Auflage 1996.

[Ull88] Ullman, J.: *Principles of Database and Knowledge-Base Systems,* Volume 1. Computer Science Press, Rockville, MD, 1988.

[Ull89] Ullman, J.: *Principles of Database and Knowledge-Base Systems,* Volume 2. Computer Science Press, Rockville, MD, 1989.

[Vio96] Viola, J.: *Extrovertierte Objekte.* Datenbank Fokus, pp. 43-47, 7/1996.

[Wal81] Walter, I.: *Adalbert Stifter – Werke.*, Verlagsgruppe Kiesel, Salzburg, 1981.

[WE96] Washburn, K.; Evans, J.: *TCP/IP - running a successful network.* Addison-Wesley, 1996.

[WM97] Williamson, A.; Moran, C.: *Java Database Programming: Servlets & JDBC.* Prentice Hall Europe, 1997.

[WSCP96] Wall, L.; Schwartz, R.; Christiansen, T.; Potter, S.: *Programming Perl (Nutshell Handbook).* O'Reilly, 2nd edition, 1996.

[Zen98] Zeng, C.: *Using Keyword Separation to Improve Searching on the Web.* Proceedings of the ISCA 'International Conference on Computers and Their Applications', pp.430-435, Honolulu, Hawaii, USA, Mar 25-27, 1998.

[Zöl97] Zöller, M. (Hrsg.): *Informationsgesellschaft – Von der organisierten Geborgenheit zur unerwarteten Selbständigkeit.* VI. Kongreß Junge Kulturwissenschaft und Praxis, Essen, 21.-23.Mai, Hanns Martin Schleyer-Stiftung 1997.

References on the Web (URLs)

References about the Web – with its high momentums and rapid changes – cannot solely be based on paper material. The following references thus point to the Web itself. However, the URLs are not too detailed and show instead the most important entry points. The author's homepage [⇒Goeschka] provides more detailed and up-to-date links to Web references.

[⇒Alta] `http://www.altavista.com/` Alta Vista search engine.

[⇒Any] http://www.anybrowser.org/campaign/ Campaign for a Non-Browser Specific WWW.

[⇒ECMA] http://www.ecma.ch/ ECMA – European association for standardizing.

[⇒FCGI] http://www.fastcgi.com Open Market, FastCGI.

[⇒GemStone] http://www.gemstone.com GemStone.

[⇒Goeschka] http://www.ict.tuwien.ac.at/goeschka/, Homepage of Karl M. Goeschka with references to online ressources.

[⇒heitml] http://www.h-e-i.de/, h*ei*tml extends and anhances the functionality of HTML by defineable tags and full programming features.

[⇒IBEX] http://www.ibex.ch/, ITASCA database system.

[⇒IBIS] http://nestroy.wi-inf.uni-essen.de/Lv/, Internetbasierte Informationssysteme.

[⇒IBM] http://www.trl.ibm.co.jp/aglets/, IBM Aglets Software Development Kit.

[⇒ICT] http://www.ict.tuwien.ac.at/, Vienna University of Technology, Institute of Computer Technology.

[⇒Informix] http://www.informix.com/, Illustra.

[⇒ISAPI] http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/internet/src/isapimrg.html Microsoft Internet Server API.

[⇒Java] http://java.sun.com/, The original source for information about Java.

[⇒JS] http://developer.netscape.com/library/documentation/javascript.html JavaScript.

[⇒MIT] http://agents.www.media.mit.edu/groups/agents/ Massachusetts Institute of Technology, Software Agents Group, Media Laboratory.

[⇒MS] http://www.microsoft.com/ Microsoft.

[⇒News] news:comp.infosystems.www.*, N ewsgroups dealing with themes related to the Web.

[⇒NS] http://www.netscape.com/ Netscape.

[⇒NSAPI] http://www.netscape.com/newsref/std/server_api.html Netscape Server API.

[⇒NW] http://www.nw.com/ Network Wizards.

[⇒O2] http://www.o2tech.fr or http://www.o2tech.com O2 Technology: Java Relational Binding.

[⇒ODI] http://www.odi.com Object Design Inc.: ObjectStore PSE/PSE Pro for Java Tutorial Release 2.0 from April 1998.

[⇒ODMG] http://www.odmg.org Object Data Management Group: The Standard for Storing Objects (ODMG 2.0).

[⇒OMG] http://www.omg.org Object Management Group: CORBA2.2 (Feb-1-1998) and UML1.1 (Nov-17-1997).

[⇒Oracle] http://www.oracle.com Oracle Web Application Server and Web request Broker.

[⇒OUC] http://www.ouc.bc.ca/libr/connect96/search.htm Okanagan University College, Library, Workshop about Search Engines on the Web.

[⇒Persistence] http://www.persistence.com Persistence Software Inc., Architecting OO applications for high performance with relational databases (White Paper).

[⇒PHP] http://www.php.net/ PHP3.

[⇒PJama] http://www.dcs.gla.ac.uk/ University of Glasgow, Department of Computing Science: The PJama Project.

[⇒POET] http://www.poet.com POET.

[⇒Postgres] http://www.postgresql.org Postgres95.

[⇒Rational] http://www.rational.com Rational Inc., The Rational Objectory Process.

[⇒RIPE] http://www.ripe.net Réseaux IP Européens - Network Coordination Centre RIPE NCC: European Hostcount.

[⇒RFC] ftp://ftp.univie.ac.at/netinfo/rfc/, Requests for Comments (RFCs).

[⇒SAP] http://www.sap.com/ SAP: SAP@Web.

[⇒SoS] http://www.sci.ouc.bc.ca/libr/connect96/search.htm Sink or Swim: Internet Search Tools and Techniques.

[⇒SunLabs] http://www.sunlabs.com/research/forest/ SunLabs: Project Forest.

[⇒UMBC] http://www.cs.umbc.edu/kqml/ University of Maryland, Baltimore County: Knowledge Sharing Effort.

[⇒W3C] http://www.w3.org/, World Wide Web Consortium, *the* basic starting point for everything concering the Web.

[⇒WOF] http://www.apple.com/webobjects/, WebObjects Framework.

Publications

[Goe93] Göschka, K.: *Microcompiler Design Language.* Diploma Thesis at the Vienna University of Technology, Vienna, Austria, 1993.

[Goe95a] Göschka, K.: *Generation of firmwarecompilers.* Presentation at the '21st Euromicro Conference – Design of Hardware/Software Systems', Como, Italy, Sept 4-7, 1995.

[Goe95b] Göschka, K.: *MDL – Microcompiler Design Language. Eine Methode zur hardwaregesteuerten Generierung von Firmwarecompilern.* Elektrotechnik und Informationstechnik (ISSN 0932-383X EIEIEE 112), Volume 112, Number 12/95, pp.659-661, published by Springer Verlag, Vienna, Austria, 1995.

[Goe95c] Göschka, K.: *Firmware Compiler Generation.* Diploma Thesis at the Vienna University of Technology, Vienna, Austria, 1995.

[DMG96] Dietrich, D.; Manninger, M.; Göschka, K.: *Internet und Smart Card.* Proceedings of the 'ASA Konferenz 1996', Vienna, Austria, Sep 24, 1996.

[Goe97] Göschka, K.: *Generation of firmwarecompilers.* Journal of Systems Architecture (ISSN 1383-7621/0165-6074), Volume 43, Numbers 1-5, pp.99-109, published by Elsevier Science, North-Holland, 1997.

[RGR97] Riedling, E.; Göschka, K.; Ramharter, R.: *DEMENET - The DEMETER Project.* Proceedings (ISBN 87 7432 465 9) of the 'First European Conference for Information Technology in Agriculture', pp.167-170, Copenhagen, Denmark, June 15-18, 1997.

[MGD97] Manninger, M.; Göschka, K.; Dietrich, D.: *Die Smart Card im Internet.* Praxis der Informationsverarbeitung und Kommunikation (ISSN 0930-5157), Volume 20, Number 3/97, pp.148-154, published by K.G.Saur Verlag, Munich, Germany, 1997.

[RGM97] Riedling, E.; Göschka, K.; Manninger, M.: *Education at the Vienna University of Technology: Traditional Lecture Based Education vs. Telematics Based Education.* Proceedings (ISBN 1-885189-03-6) of the 'International Conference on Engineering Education: Progress Through Partnerships', pp.717-725, Chicago, Illinois, USA, August 13-15, 1997.

Publications

[GR97] Göschka, K.; Riedling, E.: *Development of an Object Oriented Framework for Design and Implementation of Database Powered Distributed Web Applications with the DEMETER Project as a Real-Life Example.* Short Contributions Proceedings (ISBN 0-8186-8215-9) of the '23rd Euromicro Conference – New Frontiers of Information Technology', pp.132-137, Budapest, Hungary, Sep 1-4, IEEE Computer Society, 1997.

[Goe98a] Göschka, K.: *Design and implementation of database powered web systems – experiences from the DEMETER project.* Proceedings (ISBN 0-412-83530-4) of the IFIP WG2.4 'Systems Implementation 2000 Conference', pp.333-344, Berlin, Germany, Feb 23-26, Chapman & Hall, 1998.

[Goe98b] Göschka, K.: *Internet Software Engineering: Design and Implementation of Interactive Web Applications.* Proceedings (ISBN 1-880843-23-4) of the ISCA 'International Conference on Computers and Their Applications', pp.5-8, Honolulu, Hawaii, USA, Mar 25-27, 1998.

[GF98] Göschka, K.; Falb, J.: *Experiences from Design and Implementation of Real-Life Database Backed Web Applications.* Proceedings (ISBN 4-931474-00-4) of the 'IEEE 20th International Conference on Software Engineering', pp.122-126, Kyoto, Japan, Apr 19-25, IEEE Computer Society, 1998.

[GRRF98] Göschka, K.; Riedling, E.; Radinger, W.; Falb, J.: *Using Database Backed Web Applications for the Implementation of Interactive Tutorials on WWW.* Proceedings of the 'International Conference on Engineering Education', Rio de Janeiro, Brazil, Aug 17-20, 1998.

[GFR98] Göschka, K.; Falb, J.; Radinger, W.: *Database Access with HTML and Java – A Comparison Based on Practical Experiences.* Proceedings of the 'IEEE 22nd International Computer Software and Applications Conference', Vienna, Austria, Aug 18-21, IEEE Computer Society, 1998.

[GR98] Göschka, K.; Riedling, E.: *Web Access to Interactive Database Training: New Approaches to Distance Laboratory Work at the Vienna University of Technology.* Proceedings of the 'Teleteaching 98' as part of the '15th IFIP World Computer Congress', Vienna, Austria and Budapest, Hungary, Aug 29-Sep 6, 1998.

[GM98] Göschka, K.; Manninger, M.: *Database Powered Web Applications for Internet Marketing and Commerce.* Full day workshop at the '15th IFIP World Computer Congress', Vienna, Austria, Aug 30, and Budapest, Hungary, Sep 5, 1998.

[GF98a] Göschka, K.; Falb, J.: *New Architectures for Database Backed Web Applications.* Proceedings of the 'AACE WebNet98 World Conference', Orlando, Florida, USA, Nov 7-12, 1998.

Curriculum Vitae

Karl M. Göschka
Vienna University of Technology
Institute of Computer Technology
Gusshausstrasse 27-29/384
A-1040 Vienna, Austria
Email: goeschka@ict.tuwien.ac.at

Jan 17, 1967	Born in Vienna, Austria
9/73 – 6/77	Primary School
9/77 – 5/85	High School
June 7, 1984	1st prize at the 15th Austrian Mathematical Olympiad
May 29, 1985	Graduation from High School with 'distinction'
June 14, 1985	1st prize at the 16th Austrian Mathematical Olympiad
July 10, 1985	3rd prize at the 26th International Mathematical Olympiad
9/85	Software Engineer, Bank Austria
10/85 – 9/86	Military Service, Austria
10/86 – 6/94	Diplom-Ingenieur (M.Sc.) in Electrical Engineering with 'distinction'. Thesis: 'Microcompiler Design Language', awarded by the annual 'GIT-Förderpreis' of the 'Österreichischer Verein für Elektrotechnik' on Nov 23, 1994
10/87 – 4/95	Diplom-Ingenieur (M.Sc.) in Computer Science with 'distinction'. Thesis: 'Firmwarecompiler Generation'
'89 – '94	Software Engineer (during vacations), Siemens Austria: Development of firmware, computer architecture, compiler implementation.
7/94 – present	Research Assistant, Institute of Computer Technology, Vienna University of Technology, Austria. Main research area: Design and implementation of database backed Web applications, information engineering and multi media protocols, security concerns. Lecturing: Graduate course on 'Web databases', Supervision of 23 Masters students in thesis research. Industry projects: Coordinator of a cooperation with Ericsson Austria about wireless Intranet communication services. Development of database powered Web applications for the project DEMETER funded by the European Commission. Several small projects with Siemens Austria. Administration: Design and maintenance of the institute's LAN, Oracle database and Web-Server.