Stefan Jablonski
Ilia Petrov
Christian Meiler
Udo Mayer

Guide to Web Application and Platform Architectures

Springer

Stefan Jablonski
Ilia Petrov
Christian Meiler
Udo Mayer

Guide to Web Application and Platform Architectures

Stefan Jablonski
Ilia Petrov
Christian Meiler
Udo Mayer

Guide to Web Application and Platform Architectures

With 149 Figures

Springer

Stefan Jablonski
e-mail: stefan.jablonski@informatik.uni-erlangen.de

Ilia Petrov
e-mail: ilia.petrov@informatik.uni-erlangen.de

Christian Meiler
e-mail: christian.meiler@informatik.uni-erlangen.de

Udo Mayer
e-mail: udo.mayer@informatik.uni-erlangen.de

Institute for Computer Science
Dept. of Computer Science 6 (Database Systems)
University of Erlangen-Nuremberg
Martenstrasse 3
91058 Erlangen, Germany

Library of Congress Control Number: 2004109496

ACM Computing Classification (1998):
H.3.5, D.2.11, D.2.12, C.2.4, K.6.3, H.5.3, D.2.10, K.8.1

ISBN 978-3-642-05668-0 ISBN 978-3-662-07631-6 (eBook)
DOI 10.1007/978-3-662-07631-6

Originally published by Springer-Verlag Berlin Heidelberg New York in 2004

Typesetting by the Authors
Cover design: KünkelLopka, Heidelberg
Printed on acid-free paper 33/3142/YL 5 4 3 2 1 0

For Renate, Lisa, and Johannes
S.J.

For my mother Maria, and my father Peter
I.P.

For Marion
C.M.

For Andrea
U.M.

Preface

The Web, as we see it today, is full of various standards and technologies. If you want to solve a certain problem, you can find plenty of literature describing solutions using a single or a group of technologies. But this literature often tends to focus on just few special aspects of the Web, concealing the overall big picture. The usages and application areas of different implementation approaches are often not well illustrated. Furthermore, the range of products and standards described often reflects only current trends in the market. Often, a conceptual comparison of the various approaches is missing.

We recognized that there is a necessity to structure and classify the rank growth of standards and technologies for the Web. Consequently, this book is intended to give a comprehensive overview of the field of Web application development. The two central elements of this book are the Web application architecture and the architecture of the underlying platform, the Web application architecture, which together build a framework for Web application development.

This book is useful as a source of information for teaching purposes. We believe that a good structural overview is a prerequisite for good teaching. Secondly, the book can be a precious guideline for application designers. Readers benefit from the experience the authors have gained in projects developing Web applications.

The Web application development framework introduced in this book allows for iterative refinement and documentation of design decisions. The iterative approach has proven to be useful for Web application development, as complex architectures running in an open Web environment are hard to be developed in a one-shot manner. Often alternative design decisions have to be considered and assessed. Thus, partial solutions have to be reengineered and subsequently improved which results in a progressive refinement process leading to the final solution.

Contents

Part I:
Introducing the Web Application Design Methodology

In this first part of the book we introduce an approach to designing Web applications. Web applications have three dimensions. We can distinguish between the architecture of platforms, the architecture of applications, and a set of Internet standards and technologies. In this part we motivate the benefit of this separation and describe the specifics of each dimension.

Our design methodology for Web applications relies on a separation of concerns. We start with a technology-independent architecture of the Web application. This is then mapped onto a set of Internet standards and technologies. For this purpose we define a general classification of technologies. Finally platform components are chosen.

The Web application design methodology is a step-by-step procedure which shows an iterative character like other methodologies such as RUP. Our procedure is based on a number of iterations through the different design phases, which leads to iterative improvement of the architecture. An additional advantage of this approach is that it helps to validate the architecture during the design time. We make design recommendations in terms of "best practices" at each design phase.

Chapter 1 provides an introduction to the book. It motivates our ideas, characterizes the intended audience of the book, and gives an overview of its structure. Chapter 2 introduces our framework architecture, featuring the three dimensions discussed above. The Web application design methodology is the subject of Chap. 3. Chapter 4 presents a classification of Internet standards and technologies which forms one of the dimensions of our design approach.

At the end of this first part of the book the reader will be familiar with our Web application design methodology. How to fill this approach with concrete Internet standards and technologies is the theme of the second part of this book. In Part III we introduce concepts that facilitate the practical implementation of our Web application design methodology.

1 Introduction

The World Wide Web (WWW or the Web) has developed from an information source to a full-fledged platform for complex applications. Thus the Web has turned into a kind of melting pot for new technologies. New concepts and technologies are constantly being developed which makes it extremely difficult to find the right means when a Web application must be built. On the one hand, it is difficult to select an approach that ensures high compatibility with other approaches and is a strategic, future-oriented choice. The former is important since a Web application often has to communicate with other Web applications which might have been implemented differently. The latter is an important issue since many of these new techniques exhibit an extremely short lifetime. This results from the popularity of the Web and thus depends on subjective factors such as social trends. On the other hand, it is hard to assess the adequate granularity of a Web application. Among other things, this is caused by the fact that a group of Web application users is difficult to identify and assess. When the Web application becomes accepted its number of users grows. With increasing numbers of new users new requirements are introduced. This requires the implementation of new functionality and, for example, makes personalization necessary. All in all, Web applications can therefore be considered as highly dynamic programs.

Without going into detail now, we can define a Web application as a software system that is accessible over the Web [KPRR03]. It uses Web technologies and strives to use standard technologies wherever feasible. The development of Web applications is the main focus of our book. However, we do not want to contribute yet another book on Web application development techniques. There are many books out there and recommended for the detailed study of these standards and technologies [KPRR03] [Wild99]. We will discuss these standards and techniques here, but just to convey an insight and provide a high-level overview. Instead, we aim to provide a more global view of Web application development and present an architectural framework for the development of Web applications. The first part of this book is dedicated to this architectural framework. We will see that the framework consists of three dimensions. The first dimension covers the Web platform; the second defines an architecture for Web applications, while the third deals with Web standards and technologies. We state firstly that for each Web application development this architectural framework must be set. Then it might be appropriate to choose implementation techniques to enact this architecture. According to this methodology, in the second part of the book implementation techniques and standards for Web applications are presented. However, it is not the goal of this book to provide a kind of programming handbook for the various approaches. Instead, we concentrate on the most essential features of these approaches and focus on how to use and deploy them. According to this preference, the second part is more a guideline that deals intensively with recommended application scenarios for these approaches.

In the third part of the book we shift from a singular examination of a Web application to a more global one. We state that due to the omnipresence of the Web it is not appropriate to merely look at single Web applications. Instead, the whole landscape of Web applications must be taken into consideration which aims to cover an entire application area. In this third part of the book we present approaches such as registry management, process management, and organizational management that all pursue a global perspec-

tive. It is shown how this global perspective fosters a systematic approach for Web application development.

The mission of the book is to provide a path through the opaque jungle of Web subjects. The overall aim is to support developers in finding the right and adequate implementation strategy for their Web applications. This book is based on the experiences we gained in many years of Web application development in both industrial projects and academic research. Since it is an experience-based book that aims to convey guidelines and recommendations, its value is only appreciated when the ideas of the book are contemplated thoroughly and the essence is compiled into a personal new attitude towards Web applications. The reader who looks for sharp implementation concepts to enact concrete applications quickly will not be satisfied. Such a reader is better off with the many text books on implementation concepts for Web applications. We serve those readers who are looking for a conceptual approach to cope with the comprehensive and challenging character of this new technological frontier.

1.1 Who Should Read the Book?

Due to its conceptual character this book is useful for various reader groups. Business-oriented people with an interest in Internet-related technologies and the way they can be combined together to produce Web-based solutions can learn about the variety of approaches for Web applications. The discussions and recommendations here can help them to identify the critical issues of an application. Only if these issues are formulated precisely by the domain experts will the generated solutions meet the real application requirements. Chief information officers who are in charge of assessing an organization's technological landscape and its integration needs can extract a conceptual architecture from the book. The architectural framework presented in the first part of the book will help them to deliver a well-structured architecture for their own IT systems landscape. This book will also assist consulting professionals seeking to answer clients' problems and to find readily available and comprehensive literature without undertaking a laborious search. The discussion in this book is sufficient to gain a feeling of how to use internet standards and technologies; however, it cannot replace textbooks that meticulously explain these technologies, especially for implementation purposes.

System architects and software developers are supported since the book provides a conceptual view of Web applications. This abstract and higher level perspective is necessary to find a clear architectural structure for software projects. Looking merely at concrete technologies distracts from this essential global architecture.

The book can be used as a guide, reference and textbook in Web application engineering or information system courses at universities. It is relevant to students who are interested in the architectural and technological foundations of the Web.

1.2 Structure of the Book

This book comprises three parts (Fig. 1.1). The first part introduces the framework architecture, which is the basis for the whole book. It presents a fundamental approach to Web application development. As a result the reader will know how to structure and organize Web applications in a comprehensive manner.

The second part of the book provides an insight into the most popular and important standards and technologies for Web applications. They are introduced and assessed with respect to their roles in and the contribution they make to the architectural framework given in the first part of the book. Having read this part the user will be able to select standards and technologies suitable for a Web application development.

Part I	**Introducing the Web Application Design Methodology**	
	1	Introduction
	2	Framework Architecture
	3	Developing WAA and
	4	Classification of Internet Standards and Technologies
Part II	**Internet Standards and Technologies**	
	5	Basic Programming Concepts for Web Applications
	6	Component-Oriented Software Development
	7	Web Services and Web Applications
	8	Web Site Engineering and Web Content Management
Part III	**Complementary Technologies for Web Application Development**	
	9	Why Technologies and Standards Are Not Enough
	10	Registries
	11	Organizations and Organizational Structures
	12	Process Technology
	13	Repositories
	14	Putting It All Together

Fig. 1.1. Structure of the book

The third part of the book completes the architectural framework. It introduces practical concepts that support the development of comprehensive Web applications. These concepts facilitate the control, administration and maintenance of complex Web applications.

Both Part I and Part III contain new contributions which represent the provision of a conceptual framework for Web application development. Part II merely classifies well-known Internet standards and techniques in this framework. This part is not intended to provide a complete introduction into these mechanisms; rather, it assesses their usefulness and applicability in the context of our architectural framework. Readers who are very familiar with these concepts can skip the second part, although they will then miss the assessment of these concepts with respect to the framework architecture.

2 Framework Architecture

In this chapter we introduce a conceptual framework architecture for Web applications. The goal of such an architecture is twofold: to serve as a guide and to serve as a utility. One of the aims of any framework architecture is to support the process of Web application design by having some kind of "guide". Why do we actually need to consider a design "process" in the field of Web applications? Is it not as simple as installing a Web server, writing some PHP script, and storing the data in some database? A typical 30 minute task some readers may think. In this chapter we will make an attempt to show that the process of creation of Web applications can be much more complex, requiring much more planning and architecture as may appear at first glance. The technological diversity, all those competing approaches, and the ubiquity of the Web are certainly contributing factors.

As for the other goal, as a utility, just like traditional architectures such as the layered approach or modular approach it can serve not only for design purposes but also for validating existing designs and as a basis for comparison.

This chapter is organized as follows. The next section motivates the concept and the approach to designing Web applications. Section 2.2 provides the big picture by introducing the concept of framework architecture and describing its constituents. Section 2.3 contains a historical detour, investigating the origins and describing some of the key terms in the context of the Web. Sections 2.4 and 2.5 define the Web platform architecture and the Web application architecture respectively, two of the constituents of the framework architecture. Last but not least, Sect. 2.6 discusses the requirements that the framework architecture for Web applications should fulfill.

2.1 Motivation

We will start by motivating the three principles on which the Web is based (Sect. 2.1.1). They will serve as a reference point for the rest of the chapter; we will refer to them when discussing platform modules and architectural principles. Section 2.1.2 contains an introductory discussion on the wide variety of technologies and standards involved in today's Web application programming.

2.1.1 Principles of the Web

The Web as an environment has become attractive to solution providers and regular users because of the principles of openness, simplicity, and ubiquity on which it is found. The principle of openness has to do with the fact that the Web is based on a set of open standards, certified by a general standardization body such as the World Wide Web Consortium (http://www.w3.org/). The standards and technologies deployed on the Web should be designed to cover a broad range of hardware and software systems, should operate on top of other technologies, and should attract the attention of the broad and diverse software engineering, developer, and user communities. This set of certified standards serves as a common basis for building and integrating Web applications. The principle of openness means: free access to open standards, the possibility to propose changes or completely new standards, royalty-free use of the Web and application deployment (i.e. any

application developed for the Web can be deployed free of patent fees or other charges, which is not the case with other environments).

The principle of simplicity has two aspects: on the one hand, simplicity of use, and on the other hand, simplicity of programming. In the Web environment everyone is a potential user. With very little or almost no prior training each of us should be in a position to make use of its full potential. The creation of HTML (HyperText Markup Language) as document format is not a coincidence in this context. It is simple, easy to create (generate) and work with (read/navigate). Thus, an HTML page is a first and very simple Web application that can easily be called using HTTP (HyperText Transfer Protocol).

Both HTML and HTTP are often characterized as "very simple" [Mogu02], but the trick is that their use does not require serious infrastructure. Therefore one can assume that the required software is relatively easy to create and is readily available for any platform. The principle of simplicity influences the required infrastructure and the way we work with Web applications and their design.

The principle of ubiquity makes the Internet challenging from a technical point of view. It has a lot to do with interoperability, but also with open and widely accepted standards and technical simplicity. The Web and the technologies related to it aim at the largest possible scope, which eventually is every computer system. The principle of ubiquity is especially hard to apply and follow, because it carries many potential problems: it has to span heterogeneous systems; a universal transport and communication protocol has to be used; interoperability problems with respect to Web applications and requirements on the Web platform need to be resolved, etc. Scalability is also a relevant issue: the Web was conceived as medium which can easily scale to billions of servers.

2.1.2 Wide Variety of Technologies

In the last 10 years the Web has turned into a very dynamic playground for new technologies. Similar technologies are normally combined in technological groups with respect to their field of application (Fig. 2.1). For example, server side logic may be implemented as components (e.g. EJB or CCM – CORBA Component Model) or as Web server plug-ins (e.g. ISAPI plug-ins or NSAPI plug-ins).

Why is there such a wide variety of technologies? We can distinguish at least two reasons for the existence of competing technologies: namely, evolutionary and cross-company reasons. Evolutionary because, as the computer industry evolves, some technologies are quite naturally superseded by others. Efficiency, maintainability, extensibility, security, etc, are just some of the factors with which the new technologies are labeled. Typical examples are CGI (Common Gateway Interface) and Java servlets (Sect. 6.5.3). They belong to the same technological group – server side (Web tier logic), but they are at different technological levels. In 1997 Sun Microsystems introduced Java servlets as a technological successor of CGI scripts, helping to avoid server performance problems (processor time overhead, high memory consumption, and overall system performance overhead) posed by CGI, and introduced additional possibilities.

The cross-company reasons – market competition – is the main reason why rival software companies come up with competing technologies. Therefore we can observe similar technologies in the same technological group. Typical examples are ODBC (Microsoft) and JDBC (Sun), ISAPI (Microsoft) and NSAPI (Netscape), ASP (Microsoft) and Java servlets (Sun).

A combination of these two factors can also be observed – competing technologies from the same company regardless of the technological group. Interestingly enough,

these cannot be necessarily viewed as "successive versions" of the same technology. Within the Microsoft realm, for instance, we have ASP vs. ASP.NET; COM vs. DCOM vs. COM+; .NET Assemblies vs. DLLs (Dynamic Link Libraries).

To recapitulate, there are so many "terms" because of the natural evolution, because the big companies sell competing technologies, and mainly because the Web is one of the fastest evolving environments.

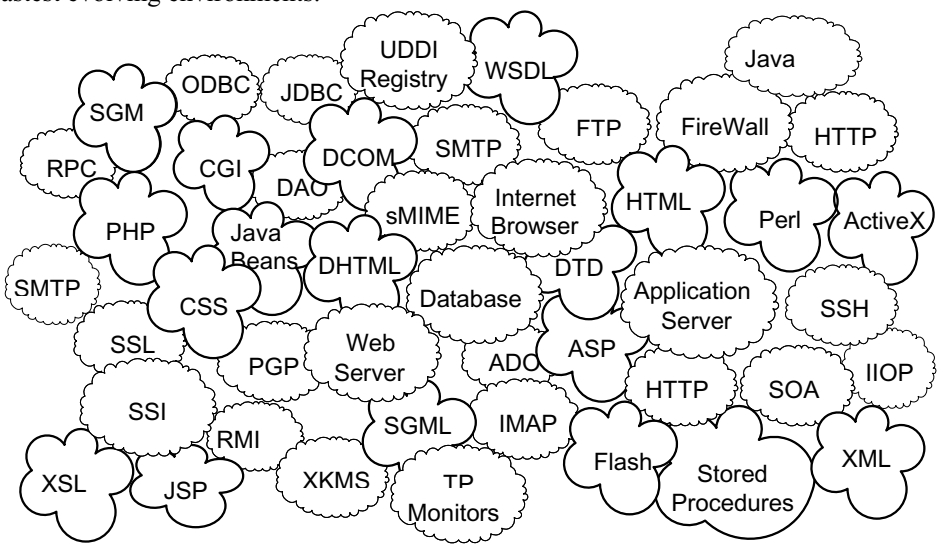


Fig. 2.1. Technological landscape of the Internet

2.2 Framework Architecture for Web Applications

In a Web environment, there is a many-to-many relationship between the architecture of an application and how it can be implemented; the variety of technologies depicted in Fig. 2.1 supports this observation. In other words, a component of an application can be implemented using different technologies hosted by different platform modules. Conversely, one platform module can execute components from different applications.

Consider the following example: an order entry application must be modified to display the order entries sorted in ascending order of company name; the bubble sort algorithm must be used for sorting the entries. No platform and technological considerations are made at the conceptual level. The algorithm is specified in pseudo code in the UML model. At a later stage code for either a standalone C/C++ application or a Java applet can be derived from the UML model and generated. In this case the bubble-sort order entry application will be just one of many running on a certain platform (either directly on the RTL and OS or in a virtual machine); however, there may be other C++ applications running on the same OS using the RTL, for instance.

The flexible relationship between applications, implementation technologies, and platforms fosters the following conceptual approach: the Web is the environment for Web applications. We distinguish two architectural aspects for each Web application: the architecture of a platform for Web applications (Web platform architecture – WPA) and

the architecture for a Web application itself (Web application architecture – WAA). This distinction is the core of all upcoming discussions. The differentiation between the architecture of the platform and the architecture of the application is discussed in [HoNS00] and [Stoe00].

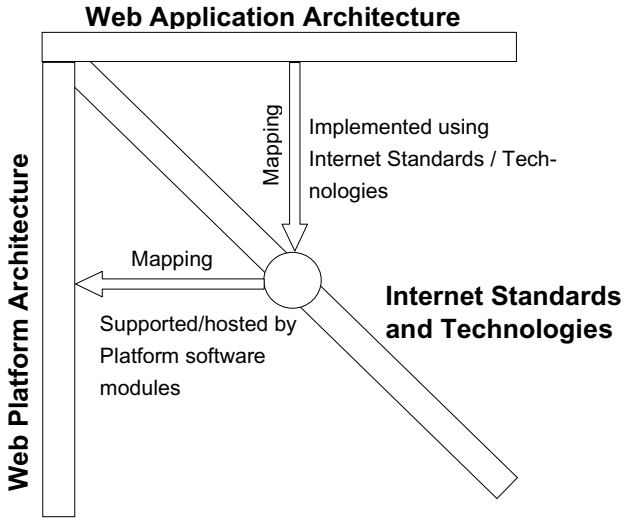


Fig. 2.2. Parts of the framework architecture for Web applications

We will make an attempt to define a classification of Internet standards and technologies and to combine it with the architectural framework. The result is a framework for the structured and systematic design and development of architectures for Web platforms and Web applications, consisting of three orthogonal dimensions (Fig. 2.2). All technologies mentioned in Part I of this book are explained throughout the chapters of the second part, which discuss different approaches to programming Web applications. Doing so enables us to provide a much more focused and functionally oriented discussion on how they can be utilized.

We call the result framework architecture due to its character. It is used to develop other architectures. However, it may also be used to describe architectures of Web applications and compare them. The development model that this architecture provides serves as a framework for creating other architectures. To underline this fact the term "framework architecture" is used.

By making the distinction between WPA, WAA, and Internet standards and technologies as in Fig. 2.2 we can easily structure the problem domain. The key principle behind this is separation of concerns. By considering application architectures separately we can employ pure software modeling techniques, and use standard modeling languages like UML sequence diagrams. We are free to leave out the restrictions a concrete technology can impose.

As a next step we map our application architecture onto a set of technologies. The specifics of each technology impose restrictions on the application design. We again regard the selection of technologies as being independent of the selection of platform mod-

ules. This introduces certain degrees of freedom, which will be taken into account in the mapping process.

In the next step the designer must choose the modules of the platform, i.e. the software on which certain application components will run. The choice is mainly based on the technology chosen in the previous phase. This is another mapping step.

To sum up, the goal we achieve by this multiphase procedure is to reduce the complexity by addressing one problem at a time. Such an approach is not genuinely new. It is central to the field of database design [ElNa02], where the development of a database scheme is divided into two phases: a DBMS-independent and a DBMS-specific phase. While the DBMS-independent phase focuses on conceptual development and meeting clients' requirements, the DBMS-specific phase maps the conceptual model onto a DBMS-specific model.

As we have already seen, a large part of the problem is reduced to mappings. The fist step is to map the conceptual architecture of the application onto technologies. The subsequent step is to map the result, i.e. application architecture and technologies, onto software platform modules. To illustrate this by a simple example let us consider the following case. In a conceptual architecture a designer decided to implement the server side logic as components. After the first mapping the best choice turned out to be Enterprise Java Beans (EJB) for portability reasons. The next step is to choose a platform module to execute the EJB and thus to map the EJB components onto a platform architecture. Due to the three factors of portability, low cost, and small-scale solution, the designer chooses JBoss as an EJB server/container.

In the following sections we will provide a detailed motivation for the three components of our architecture: platform architecture, application architecture, and technologies. These three components will then be discussed in subsequent sections.

2.2.1 Why Platform Architecture?

Any application, regardless of how simple or how complex it is, requires the presence of certain software modules in order to run. It runs on top of them by using the functionality they provide. The combination of these software modules is typically called a platform. Many applications can run on a single platform. A platform may be loosely defined as: any set of technologies or software modules on top of which other technologies and other software execute.

```
#include <stdio.h>
int main( void ){
    char *strOrdrEntry = "Order Entry Example.";
    printf( "%s", strOrdrEntry );
    return 0;
}
```

Fig. 2.3. Sample application program

In order to get a practical insight into this matter, let us consider a simple order entry application written in ANSI C/C++. The program calls the standard function printf() to display a desired string (Fig. 2.3). Printf() is an ANSI C function provided by the Run-

Time Library (RTL). Loosely, the RTL contains an implementation of a set of standard functions, such as I/O, memory management, process management, etc, that any C/C++ program may use to abstract from the specifics of the underlying operation system. In this sense the RTL is part of the platform for ANSI C/C++ applications. Most of these functions have been standardized by an ANSI body within the C/C++ language standardization effort. The goal was to achieve portability at source code level.

After the compilation the generated object file is linked to the RTL, which contains the implementation of printf() for the current OS using the OS Native API (Fig. 2.4). Therefore the call we make in the program eventually translates to a set of calls to functions provided by the OS. The printf() implementation wraps them, thus building up a layer of abstraction, i.e. prinf() can be implemented on different OSs using different native functions; the functionality it provides to the application program remains the same. The Native OS functions interact with the OS, which in term interacts with the driver, which eventually instructs the hardware device to depict the string (Fig. 2.4). The RTL and the OS are part of the platform on top of which the order entry application runs.

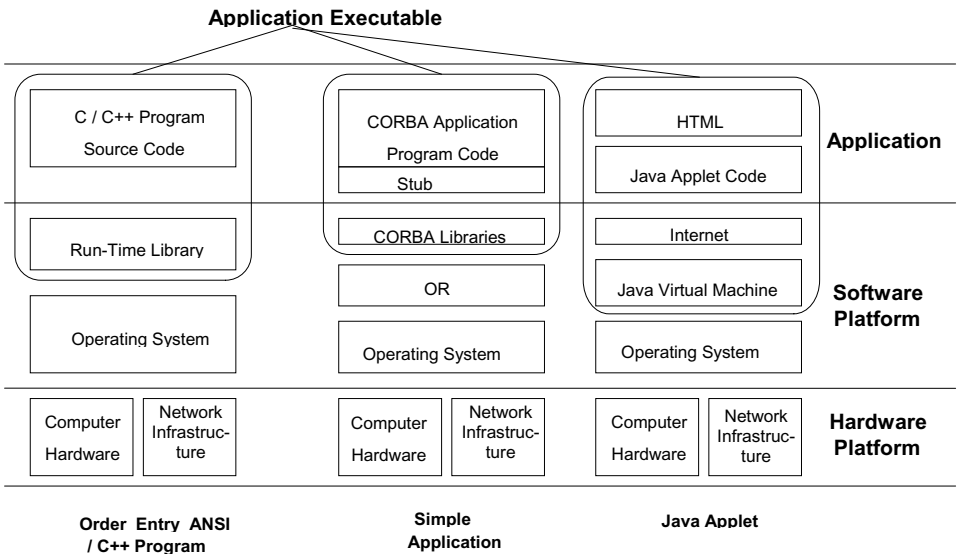


Fig. 2.4. Sample software and hardware platforms

From Fig. 2.4 we observe the fact that even the simplest application needs a platform, which in the simplest case is the OS. Another interesting issue is that we can also clearly distinguish two kinds of platforms, hardware and software platforms. The software platform comprises the OS, the respective libraries, the execution environment, etc. As we can see (Fig. 2.4) a CORBA application will be linked to the respective CORBA libraries, handling the interaction with the ORB. The ORB abstracts from the specifics of each OS and the heterogeneity and complexity of the distributed environment. An ORB implementation is available for every OS. Using similar reasoning we can describe the software platform for a Java applet. The reader can find a good introduction to the Java and CORBA basics in [OrHa98], [CaWH00].

A software platform provides services (graphics, networking, logic, I/O) to the application and typically involves several layers of abstraction. The goal is to achieve among other things portability, added-value services (the services offered in the libraries offer value-added functionality on top of the "raw" OS functions), etc. The notion of (value-added) services such as transactional support or security offered by the platform is discussed briefly in Sect. 2.3.5, and in detail in Chap. 6.

The notion of hardware platform is closely associated with computer hardware and network infrastructure. By computer hardware we mean all hardware components of a computer like the processor, system board, memory, etc. The network infrastructure comprises all hardware equipment required for reliable networking. It involves network controllers, the cables, network switches, routers, wireless LAN access points, and so on. We prefer the term "network infrastructure" since it implies the physical distribution. A detailed discussion of computer hardware is beyond the scope of this section.

The importance of the platform in the application design has often been downplayed. This can be easily explained by the nature of most of the small or middle-size applications today – designed and coded for a single OS using a small number of technologies. In Web applications we have the converse phenomenon. The infrastructure is complex, heterogeneous, requires a lot of consideration to support and configure. The actual application is "concealed" behind the software platform, i.e. behind terms like Web server, application server, etc. The complexity and variety of the infrastructure requires a thorough design of the underlying platform, i.e. architecture of this platform is demanded.

2.2.2 Why Application Architecture?

The application architecture is to be understood in the context of conceptual architecture [HoNS00]. It is closely associated with modularization – how the application functionality is decomposed into modules (module view). Execution models (execution views [HoNS00]) expressed for example as sequence charts reflect the execution flow of the application. In contrast to the platform architecture, the application architecture focuses mainly on the design of application functionality. Its necessity is discussed in all standard literature on software engineering [Somm00].

Typical application architectures are created using UML models. Design patterns are normally applied on application architectures [GHJV97], [SiSJ02], [BMR+96], [AlCM03]. We discuss in Sect. 2.5.2 how the Model-View-Controller design pattern can be applied to Web application architectures. The idea is to develop the architecture application in a technology-neutral and platform-neutral manner. The importance of this idea lies in the fact that the same architecture can be implemented using different technologies or different alternative technologies from a technological group, which makes the design robust against technology changes.

The factor technology- and platform-independent application design gains specific importance in the field of Web applications. One of its characteristics is the myriad of technologies and competing approaches. A well-designed and technology-independent architecture also guarantees a controllable way to map it onto a preferred set of implementation technologies.

Let us also consider the fact that a large part of the technologies actually involves languages. Different modules of the Web application are coded in different programming languages: HTML, PHP, CGI, C/C++, Java, SQL are just a small part. This not only explains the diversity of the code base, but also justifies the mapping phases (Fig. 2.2), or

in other words, the concept of technology-independent WAA, mapped onto a set of target technologies.

2.2.3 Platform and Application Architectures Combined – Why We Have to Consider Technologies

To summarize, consider the following: applications always run on platforms. There are architectures for the platform and architectures for the application. Each platform offers specific services (e.g. memory management, I/O, visualization, transaction processing, and security) to the applications. Platform components together serve as execution environments for the application logic.

Taking into account the existence of either a platform architecture or an application architecture, we should be able to identify them in Fig. 2.4. Figure 2.5 shows such an attempt whereby just two implementations are shown, the C/C++- and the HTML-based implementations. It is quite obvious that we associate the software and hardware platforms with the WPA. But where is the conceptual architecture of the application? We might consider the upper layer of the implementation stacks as architecture; however, two architectures would then result, one for each implementation variant. However, we are looking for a single conceptual architecture which is the starting point for the two (or more) implementations. Therefore, we interpret the upper layer of the implementation stack in Fig. 2.5 as the result of the merger of a WAA and a set of selected Internet technologies and standards, i.e. it shows two possible implementations of the application.

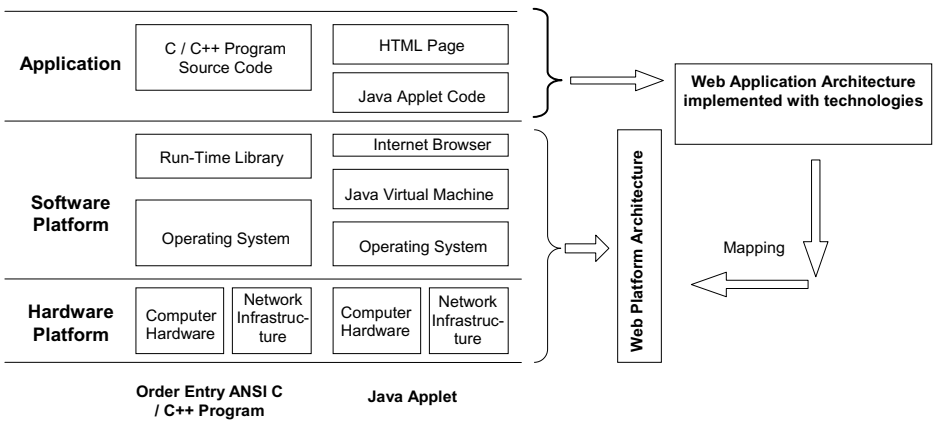


Fig. 2.5. Distinction between platform and application architecture

One of the conclusions we can draw from the previous discussion is that there is a missing element in Fig. 2.5 which provides the glue between the platform and the application architecture. We mentioned the existence of technologies, but we did not consider them. Figure 2.6 demonstrates how Fig. 2.5 has to be extended in order to reflect the previous discussion. We add a layer "Conceptual Architecture" which holds the general architecture of the application. This layer presents the WAA. Through Internet standards and technologies the WAA is associated with the WPA. The chosen Internet standards

and technologies directly point to – possibly different – implementations of the described application.

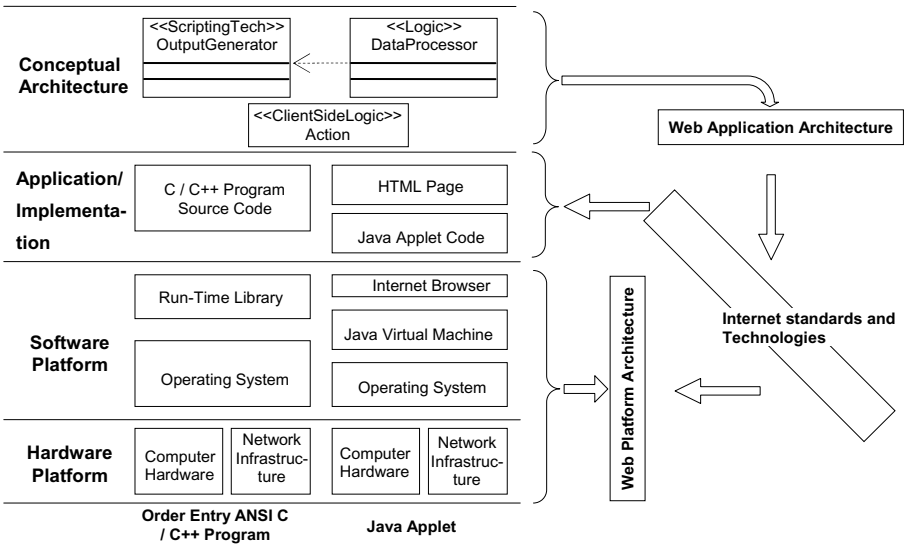


Fig. 2.6. Application architecture, platform architecture and technologies

The Web application functionality requires the capabilities of the platform in order to execute. The distinction between these is a key design concept. It allows the application designer to focus on the design task, but still refer to generic types of platform modules.

The relationship between capabilities and functionality is bidirectional. The WAA influences the choice of capabilities and, vice versa, the selection of capabilities affects the implementation of functionality. Working with generic capabilities seems to allow designers to work freely with whatever capabilities they require. On a conceptual level it is desirable to have such an underconstrained model. In practice, however, constraints exist. They result from the characteristics of the technologies and from the precise capabilities of each platform module. Therefore the proper choice of technologies is affected by both the application functionality and the platform capabilities.

Introducing technologies means that one or more matching steps need to be considered, where requirements are reflected. These requirements may involve the technologies to be used, e.g. Java, CGI or ASP, some of the platform modules such as IIS or Apache Tomcat. The following example illustrates the correlation of technologies for the platform and application architecture. A client needs a simple report generator for its order entry application that must be implemented with interoperable technologies (Java and CGI) and must be able to handle incoming requests at a rate of hundred requests per second. The WAA consists of separate components handling incoming requests, the HTML generation, and report generation logic. A number of alternatives are possible when matching platform components and technologies. Choosing CGI to implement request handling and the HTML generation component is the first alternative. Such a choice would entail a change in the implementation of the Web application because the request handler and HTML generating components must be combined. Moreover, CGI shows

low performance and scales poorly with a high number of requests. The second alternative is choosing a Microsoft specific solution: ASP for the combined entity of HTML generation and request handling and MS IIS (Microsoft Internet Information Server) as the platform. Such a solution would also require a change in the application architecture, i.e. a combined request handler and HTML generator. Although such an approach would meet the performance requirement, it is too vendor specific, which is an obvious disadvantage because it fails to meet the interoperability requirement. Choosing a J2EE solution is the third and last alternative. It would meet both the interoperability and the performance requirements. Nevertheless the implementation would undergo some changes. The request handler would be merged with the report generator and the combined request handler and report generator would be implemented as Java servlets, while the HTML generation is made as JSP. Apache Tomcat, which is an integrated JSP servlet execution engine (Catalina) and an HTTP server (Coyote), may be chosen as a platform component.

In summary, let us once again stress the role of the principle of openness. We introduce a set of Internet standards and technologies as the third dimension of our Web applications framework architecture. Application architectures are mapped to (i.e. implemented with) standardized technologies, and, further, deployed and executed on a set of platform modules created for these standardized technologies. The goal is to have a single application architecture running on many platforms and one platform module running different parts of different applications. This ideology is most concisely expressed by the phrase “Configure! Do not invent!”

2.3 From Client/Server to WWW

In this section we will take a deeper look at the evolution of the Internet, the Web, and notions like Web server or application server, which are an inseparable part of the today’s Web landscape. Such retrospection will provide us with an insight for the introduction of the WPA (Sect. 2.4).

We will put the discussions in the following section into historical perspective, considering to a certain degree the natural evolution of the technologies. This perspective will help us to define the reasons for the emergence of a certain technology and the goals pursued.

2.3.1 General Introduction Client/Server

The Web builds on top of the Internet. It is actually one of the services the Internet offers. A significant part of Web applications and Internet applications are in fact client/server applications. We call them conventional Web applications. There are, however, whole classes of applications like peer-to-peer applications and asynchronous communication applications (e.g. electronic collaboration, MOM, document based applications), which run on the Internet and are not client/server applications.

The term client/server is overused. It stands for: hardware architecture, software architecture, and a communication model. The use of client/server in the context of hardware architecture is more or less obsolete. The term client stood for a less powerful PC connected to a high-performance mainframe computer (called server).

The term client/server is predominantly used in the context of software architecture nowadays. The “server” is, in this case, the part of the application (software) which is

common to all “clients”. The “server” part comprises the bulk of business logic and external resources; it requires a lot of computing power and is therefore typically installed on a server machine. The “client” part of the application is relatively lightweight, contains client side functionality, and is therefore typically installed on client machines. The communication between the client and the server side application is typically called networking and is done over TCP/IP. Real-world examples for this type of software architecture are the modern database systems.

The use of the term in the context of a communication model is associated with terms such as roles and request/response. When regarded as roles, client and server are closely related to the fact that a server side application can actually invoke another application, and thus act as client for it. So the client and server are actually treated as relative roles to distinguish the two parties, namely the “invoker” and the one “invoked”. The client is a request sender, the server processes the request, and as a result returns a response.

2.3.2 History

At the very dawn of the computer industry computer systems were primitive machines running single user monolithic applications. Only one user could interact with a system at a time and only one application could be executed at a time. There was no multiprocessing, no multitasking; also the systems had no multiuser mode of operation.

As the technology evolved the computing facilities became larger and the support of multiple users working concurrently with the system became a necessity. Terminals were designed to handle the input and the output of large computers (Fig. 2.7). IBM 3270 terminals represent a very good example. The terminals were text based and supported the concept of forms. Some of the more advanced models could even establish a dial-up connection to the main computer. Terminals had almost no processing power; they could handle and buffer user input and display the output, but they could not really execute pieces of application logic. This is why they were called “dumb” terminals. The concepts of having remote presentation and forms-based input as we have them today in the Web were first implemented in terminal-based systems.

The next phase in the evolution is closely associated with mainframes (central computers) and minicomputers. The main concepts that were brought about were the concept of having many applications running concurrently on a single platform, and the concept of having applications sharing the services a platform provides, e.g. communication or data management capabilities (Fig. 2.7).

The next evolutionary step can be attributed to the emergence of the PC and computer networks. They have made it possible to connect computers with a stable and relatively high-throughput network connection. The physical distance back then was an issue but it was gradually resolved. Notions like LAN and WAN, like Ethernet and Token Ring, hub switches, and routers emerged. What networks gave was the possibility to transfer not only text but also large volumes of binary data, and to perform remote procedure calls etc. This influenced both the application and the platform architecture. Different application modules were supported on different servers. The infrastructure required to ensure communication became part of the platform.

The PC was created by IBM. The idea was to have a cheap and low-performance computer capable of running its own applications. In contrast to the “dumb” terminals the PCs were autonomous and self-sufficient. They had their own OS and were capable to ofhandling not only simple input/output but also application logic pieces (typically UI

but also some business logic). Centralized systems with "dumb" terminals were gradually replaced by server machines having a number of PCs as clients instead of terminals.

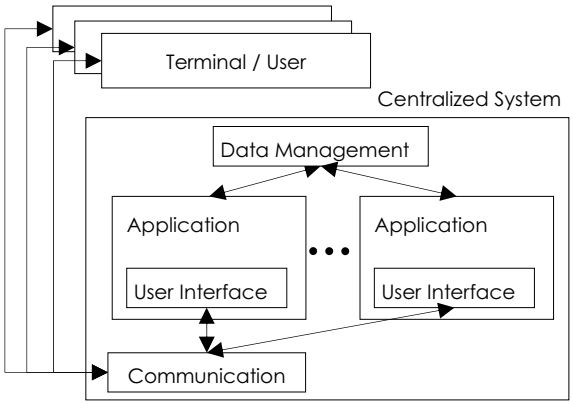


Fig. 2.7. Centralized system architecture

Technically, this introduced the RPC (Remote Procedure Call) computing model. The idea was to invoke functions of an application physically located on a remote computer. This is implemented by packaging the call to some kind of request, sending the request to the remote computer over a TCP/IP connection, unpackaging the request, and carrying out the code locally on the remote computer. The return result is sent back the same way only it is packaged as a response.

The first conclusion which we can draw by considering this change from a more conceptual point of view is that tiers appear. We can distinguish a client tier and a server tier (Fig. 2.8). From an application design point of view, handling just the input and the output (presentation) on a remote computer (client tier) was a huge step forward, because applications were no longer located on a single computer. User events and data are transmitted (serialized) from the client and over the network to the server computer. After processing the client request, the results and data are returned to the client machine and displayed.

In this respect there are two relevant notions – thin and thick (rich) client (Fig. 2.9). Thin client stands for a client part of the application which handles just the user interaction (i.e. visualizes the presentation, handles, and transmits the user events) and has no client side business logic. Thin clients are lightweight and simple, posing a few requirements on both the software and the hardware platform. Web applications typically have thin-client architecture. Technically thin Web application clients require no expensive and time-consuming distribution and the deployment effort is reduced to a minimum. Changes in the Web applications take effect and are available to the user immediately.

Thick or rich clients, on the other hand, contain client side business logic, and have typically richer presentation capabilities. Rich-client architectures typically pose more serious requirements on both the hardware and the software platform. Java Web Start applications are an example of rich clients. They typically have a powerful GUI and client side business logic requiring visualization, communication, and data storage from the software platform. Rich clients require more processing power and memory from the hardware platform compared to thin clients.

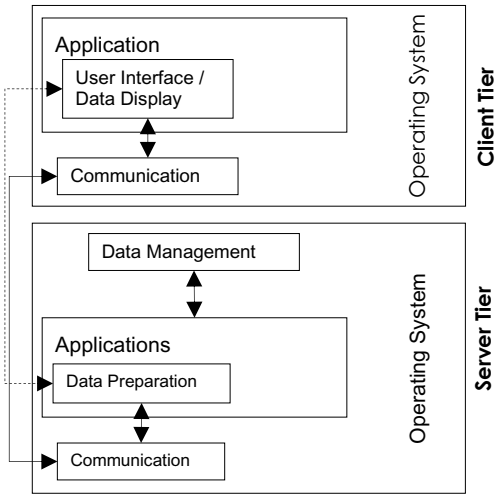


Fig. 2.8. Two tier client/server architecture

Pure client/server architectures are only of historical importance. The disadvantages include poor scalability and very high maintenance costs. Client/server architectures exhibit serious performance lags with more than 1000 concurrent clients. The communication and efficient resource management are serious bottlenecks. All these may result in higher downtimes.

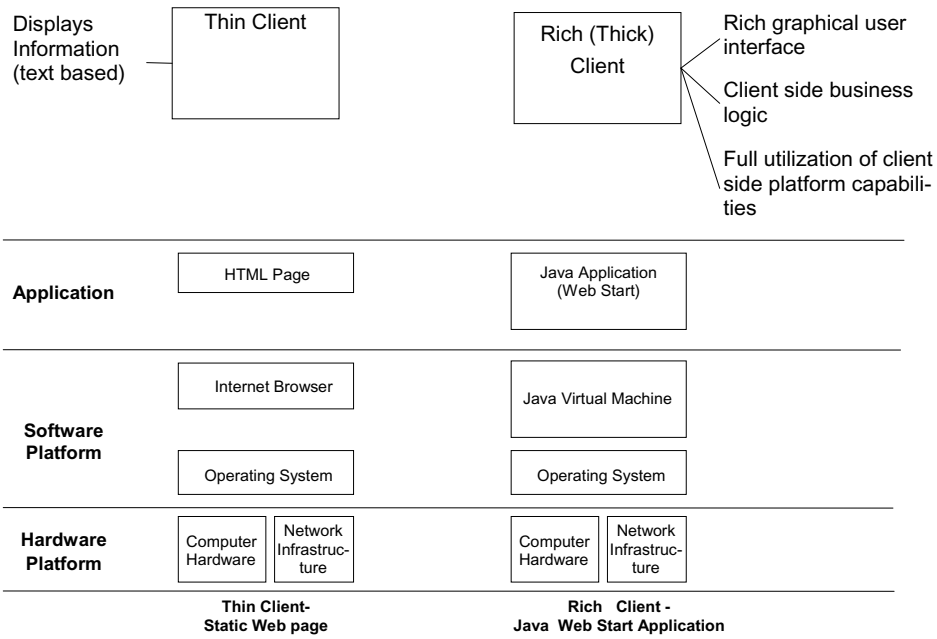


Fig. 2.9. Thin and thick clients in a two-tier architecture

We discuss the evolution of the Internet and the Web further in the next section, both of which are deeply rooted in the client/server architecture. Before we do this let us briefly discuss another relevant matter. Client/server as software architecture continued to evolve and several branches (middleware, distributed computing, component-oriented computing, databases, and transaction processing) originated from it. These are commonly described by the term enterprise computing [BhRa00], [FKNT02]. The Web followed a separate trend of development. For quite some time these trends (Web and enterprise computing) have been evolving in parallel, independent of each other. However, roughly since 1998 we observe a merging trend. Enterprise computing became a natural part of the Web, covering the server side tiers. In the forthcoming sections we will see that enterprise computing technologies are gradually finding their place also in the Web tier. Historically there were no primary and secondary trends. To simplify the motivation in this book we will assume that the Web trend was the dominant one.

2.3.3 The Web

The Web is just one of the services the Internet provides – probably the most popular one. Before we go into a more detailed discussion let us briefly say a few words about the Internet.

2.3.3.1 The Internet

The Internet is "the" global network. It is the largest network having the widest possible physical span. The Internet can, perhaps imprecisely, be associated with the phrase "the network of all networks".

Under the notion of network we actually mean a TCP/IP network, comprising the physical network infrastructure (LAN infrastructure) and the logical infrastructure, (routers, operating systems, DNS servers). Briefly, a network is everything required to execute a command like "ping w3c.org" (the involved protocols are Ethernet, IP, ICMP/TCP, DNS).

The Internet is often said to provide different services, which is indeed a misnomer. It is actually the proper implementation of the TCP/IP networking software which provides these services. This implies that one can have many of those services in a simple LAN, without having to be "connected to the Internet", which in turn yields the definition of the term intranet. The Internet is in this respect simply the cables connecting the different LANs, the set of software TCP/IP modules, and the computers on which they run.

Among some of the services which the Internet provides are (Fig. 2.10):

- e-mail – (SMTP, POP3 / IMAP, MIME)
- DNS – Domain Name Service
- FTP – File Transfer Protocol
- Telnet
- Web
- and others.

Any of these services is available on specific TCP/IP ports. Therefore any program with a TCP/IP connection to a host on a specific port can use the service offered on that port. Every computer with an Internet connection is called an Internet host. Every Internet host has a unique network address called the IP-address. Any services provided by the respective host can be addressed by a combination of IP address and a port number.

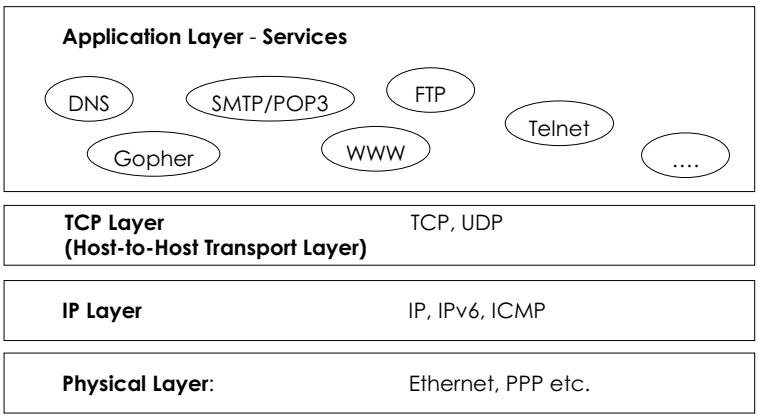


Fig. 2.10. Some Internet services

2.3.3.2 *The Web*

The Web introduces three completely new concepts:

- A new addressing (identification) mechanism [IRFC94]: URL (Uniform Resource Locator) and its more powerful derivatives URN (Uniform Resource Name) and URI (Universal Resource Identifier).
- A transport protocol: HTTP (HyperText Transport Protocol), is a on top of TCP/IP as a communication protocol.
- A language for formatting Web documents: HTML (HyperText Markup Language).

Let us now take a quick technical detour. The Web consists of a set of computers connected to the Internet (Internet hosts), running a piece of software called Web server (Sect. 2.3.4). For example, behind any URL of the type http://www.orderentry-example.de there is Web server software. Any Web server software listens by default on TCP/IP port 80 for incoming requests. For example, if you open an Internet browser and type www.orderentry-example.de it will first establish a connection on port 80 to the Internet host behind the URL and then send an HTTP GET request to receive the standard file index.html.

In the simplest case there will be multiple static HMTL documents on the file system of the Web server, which form a static Web site. The HTML documents consist roughly of contents (text, references to pictures, etc.) and links. The links contain URLs pointing to other documents, some of which may be on different servers. This is how the information on the Web is linked, which led to the name of the environment – the Web (in the literature one also finds the expression “Information Web”).

The goal of the URL is to represent a universal addressing mechanism for the Web. We assumed that a URL points to HTML documents. In the large majority of cases this is true; however, we can also address files (resources) or even include server side program calls as part of the URL.

HTTP is the transport protocol over which communication in the Web is carried out, i.e. the client side platform (the Internet browser) talks to the Web server. HTTP is a

document-based protocol, i.e. the user clicks on a hyperlink which is translated to either HTTP GET or POST commands. An HTTP request is generated, which is sent to the Web server. After processing it the Web server sends a response back, containing an HTML document. HTTP is a relatively simple protocol – session support was, for example, not included in the original versions and was introduced subsequently in terms of cookies. HTTP is not a binary protocol which contributes to the higher interoperability. HTTP is document-based [Wild99] in contrast to the RPC communication pattern in typical enterprise computing applications. There was no support for sessions in the original HTTP version; it was added later with the introduction of cookies [Wild99].

2.3.4 Web Server

The term Web server stands for a set of software modules, which must be installed onto an Internet host computer so that it can participate in the Web; the Web server handles HTTP requests, and retrieves and delivers HTML documents as a response.

The term is ambiguous; it refers to both the software package and the dedicated computer (the Internet host) on which the software is installed, and sometimes to the combination of both. In the coming discussion we use the term Web server in the former context (as a software package).

Web server is a composite term, designating a collection of software modules. A Web server typically includes an HTTP server, a CGI "environment", and it may also include an FTP server module. Some Web servers also include an API for writing server side modules (plug-ins, extensions), e.g. ISAPI or NSAPI, which goes into the direction of an application server (Sect. 2.3.5). For reasons of simplicity we assume the following nomenclature: a Web server consists of an HTTP server and a standard application server, handing the server side business logic (Fig. 2.11). The standard application server must not exist physically as a software module. We assume that it is a logical module of the Web server, which handles management of executed process, resource management, etc. The notion of a standard application server significantly simplifies the rest of the discussion.

On every Internet host running a Web server there will be exactly one HTTP server listening for incoming requests on port 80. Of course the HTTP server can be configured to listen to another port. We will refer to port 80 as the WWW port. It processes the request, and in the normal case it instructs the standard application server to fetch a static HTML document or to execute a CGI program. The standard application server returns in either case an HTML document, which the HTTP server wraps in an HTTP response and sends to the client.

The structure of an HTTP server is shown in Fig. 2.11 (not all of the modules discussed here are mandatory). The connection manager handles incoming connections on the HTTP server port. It is also partly responsible for managing sessions (opened explicitly by the client).

Let us assume for our example that the actual entry point to the order entry system is located at http://www.orderentry-example.de/input.html. Once the user types this URL the browser on the client site will contact the DNS server and resolve the name www.orderentry-example.de, i.e. it will get the IP address (e.g. 131.0.0.3) for this name [Wild99]. It will request a connection to the internet host 131.0.0.3 on the WWW port (since it is an HTTP request). If the Web server is not overloaded the connection manager will open a connection. Once this is done requests may start arriving.

All incoming requests are then passed to the request manager. For example, right after the connection has been opened the client browser formulates a "GET http://www.orderentry-example.de/input.html" HTTP request. The request manager parses and analyzes the request. After the analysis the request manager determines that the client actually requests the resource "input.html".

Before the actual resource processing is done control is passed to the security manager. It checks the permissions assigned to the client. For reasons of simplicity we prefer to talk about "clients in general"; in practice, however, every client within a session can be assigned certain security privileges.

The resource manager is configured to process different resource types. It instructs the standard application server to retrieve the resource "index.html" in our example. For reasons of simplicity let us assume that we have just two resource types: a static HTML file and a CGI program. In the case of a file the standard application server will resolve the path and simply retrieve the HTML file. In the case of a CGI program the standard application will take the call parameters and form a CGI environment. Then it will invoke the program using the environment to pass the parameters to it. The program executes and generates an HTML output as a result, which the standard application server passes to the resource manager.

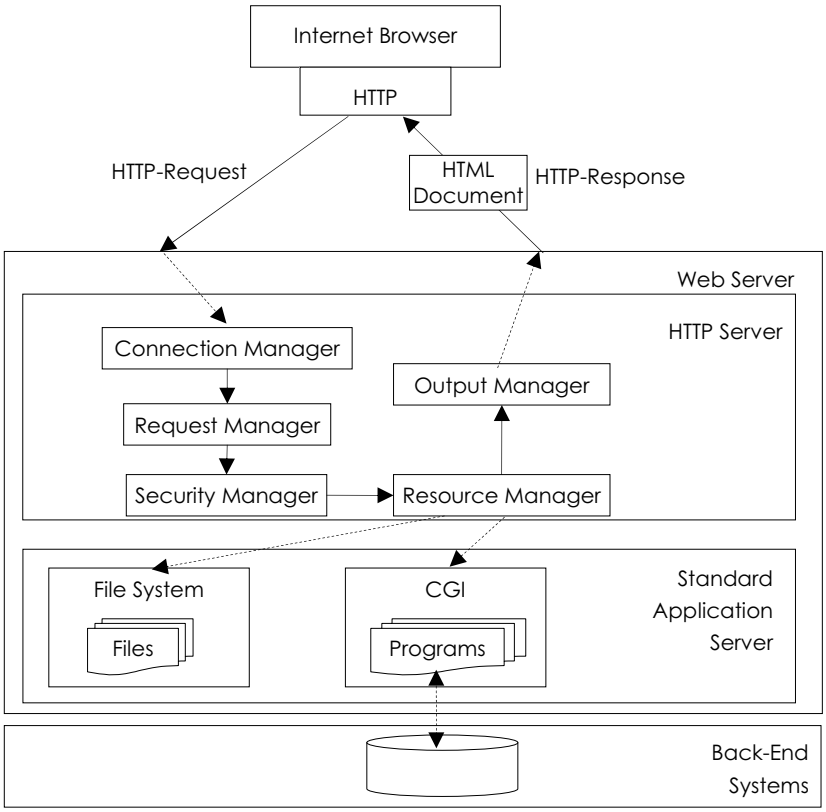


Fig. 2.11. Functional structure of a web server

Next, the output manager is activated. Its task is to form syntactically correct HTTP responses. It takes the HTML output from the resource manager and wraps it in an HTTP response. In the case of failure it generates the respective HTTP failure response (for instance, 404 error code will be returned if the file input.html is unavailable).

Historically, CGI was the only way to let programs "execute from the Web browser" i.e. to trigger server side program execution with an HTTP request. CGI as standard is supported by almost any Web server.

In the early phases of Web evolution CGI was an appealing choice, and there were almost no alternative technologies. The standard application server soon became a bottleneck. There was an ever growing demand for business logic, and all invocations were made through the CGI. Standard application servers exhibit poor performance running CGI programs due to poor resource management. There were two groups of disadvantages: sluggish execution performance and poor memory management, which resulted in poor scalability. There was a clear need to extend the standard application server. Two possibilities evolved:

- A more efficient technology to substitute CGI as a "back-end" interface for HTTP servers.
- A better model for logic. A solution to this problem was already available in the field of enterprise computing – it suggested the use of more sophisticated application server technology.

2.3.5 Application Server

In this section we will elaborate on the notion of application server. As motivation we will use the arguments given in the previous section and also in Sect. 2.3.2.

Before we start let us shortly review some of the most relevant arguments. application server is a notion emanating from the field of enterprise computing. It appeared as the result of the logical evolution of client/server systems and the development of more sophisticated computing models. It "resides" on the "server tiers" and is typically between the HTTP server and the back-end systems (e.g. database systems) as already indicated by Fig. 2.11. An application server is part of the WPA and runs one or more containers, which are an execution environment for server side business logic components as we will see.

2.3.5.1 Where Did Application Servers Come From?

The idea of having an application server acting as a "container" for different business logic components is not new. It is deeply rooted in the concept of transaction processing and more precisely in the concept of "Component Transaction Monitors" (CTMs) [Mons01].

The original idea emanates from the fact that complex transactional behavior is in principle considered orthogonal to the business logic [GrRe93]. This has two important practical implications. Firstly, the application becomes much simpler. And secondly, the platform can take the burden of implementing and enforcing the complex transactional support. This means that transactional support can be implemented as a service offered by the platform, which can be used in an almost declarative (automatic) manner. Typical examples in this respect are the majority of large and middle-size database systems.

The declarative character of such services gained more and more importance with the shift in programming paradigms – from structural to object-oriented programming, and

later on with component-oriented programming (Chap. 6). Invocations on component methods could be made transactional in deployment time (without having to change the component itself). The declarative nature of the use of such services complements the principles of encapsulation and reuse.

Gradually, the software companies realized that other concepts can be implemented in such a service-like manner, e.g. persistency, security, connection pooling, lifecycle management, introspection, naming, etc. And this is how application servers appeared – as platform modules to which such services can be "plugged in" and offered to the business logic components deployed in it.

To recapitulate, the notion of application server is relevant mostly in the context of component-oriented programming. The application server acts as a container in which business logic components can be deployed. After deployment they can use the services which the container offers.

The following issues need to be considered in the context of application servers. Firstly, application servers bridge the gap between the HTTP server and the back-end systems. Consequently they replace the traditional standard application servers from Fig. 2.11. Secondly, they assure enterprise-scale server side business logic for Web application. Thirdly, they must provide security, transaction processing, load balancing, high performance, and above all – scalability.

Back-end systems must almost always be considered when talking about an application server. Such systems are typically applications or even whole systems which are used in one way or another by the business logic running in the application server. Databases are typical examples of back-end systems (Fig. 2.11). Some application frameworks even go as far as to introduce a standard architecture wrapping the whole system and representing it as a component. Consider for example the Java Connector Architecture [JCAr04].

2.3.5.2 Application Server Is a Composite Term

After analyzing the structure of commercial application server products (e.g. [Iona04], [WebS04], [OraA04]), one easily comes to the conclusion that the industrial concept of application server differs from the one we defined above. Indeed enterprise application servers have a much more diverse structure, including significantly more products than one might expect. Modern commercial enterprise application server products typically contain (Fig. 2.12):

- an HTTP server
- a portal server, a wireless communication system, sometimes a content management server. Big application servers such as IBM WebSphere, Oracle Application Server, and BEA WebLogic include different value-added software such as a content management server, a WAP portal, and a server for e-commerce portal sites. Although these services are not directly related to the core functionality of an application server as an execution environment for business logic, they are most useful in building industrial applications.
- a Web service system – soap router, internal UDDI node, possibly a component connector (Chap. 7)
- a scripting subsystem – J2EE Web components container (JSP, Java servlets), ASP engine, etc. It contains built-in support for scripting approaches (Chap. 5). Although scripting approaches do not belong to the core functionality of an application server they are still considered a useful add-on.

- one or more containers for different types of business logic, EJB container, CORBA ORB
- different kinds of connectors to back-end systems
- typical enterprise computing services such as load balancing, TP monitor, message-oriented middleware server, etc.
- some application servers even include a database system. A typical example of such a system is the Oracle Application Server, which offers full functionality when coupled to the Oracle Database Server. Many database servers such as Oracle Database Server, and IBM DB2 contain native support for Java. Close integration with the application server facilitates the development of business logic.

To put all these into the context of Sect. 2.3.4 the transition from Web server to application server follows the paradigm shift. Initially the problem was having a Web presence. Nowadays the problem is making Web applications better and more robust, and of course how to merge the two fields – the one of Web and the one of enterprise computing. At the very beginning a simple standard application server was sufficient. With the tendency to put more complex business logic on the server side, the standard application grew and became the predominant module.

In summary, both the application and the Web server are complex notions. They comprise two notions (HTTP server and component container) and a number of different technologies. They are part of the server side platform for Web applications.

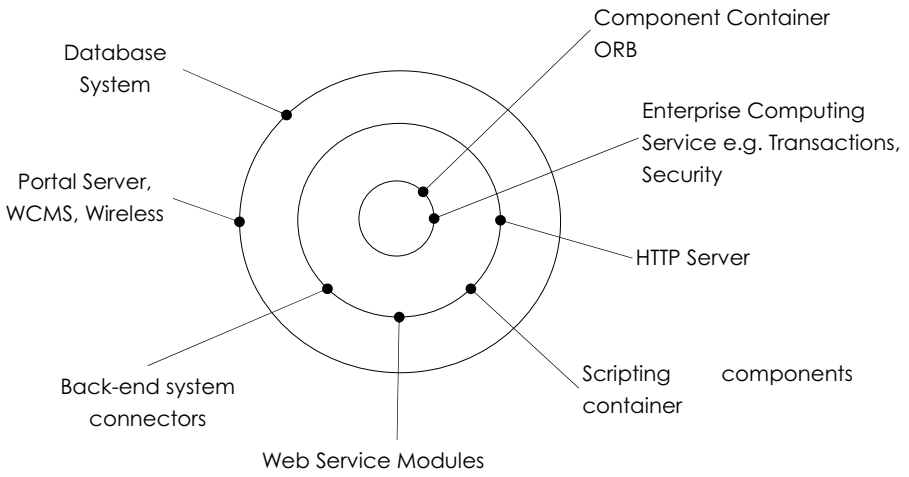


Fig. 2.12. Components of a commercial enterprise application server

2.3.6 The Three- and Four-Tier Architecture

Now we have completed our discussion on Web servers and application servers, let us present a layered view of the matter discussed up till now. This section will give a bird's-eye view of three- and four-tier architectures.

The analysis of the functional structure of a Web server (Fig. 2.11) leads to the observation that a mapping of this structure to a three-tier architecture is natural. Figure 2.13a illustrates this. However, a sort of heterogeneity can be detected in the middle tier of this architecture. The HTTP server handles the Web-related tasks, as mentioned in the previous section. On the other hand, the application server handles the execution of the server side business logic and coordinates the data manipulation operations with the underlying data tier. This separation of responsibilities motivates splitting the middle tier into two distinct tiers, the application server tier and the Web tier (Fig. 2.13b).

An obvious advantage of three-tier architecture is its simplicity. It has also proven to be suitable for data-oriented Web applications, where the emphasis was on data querying and data input/output, rather than on data processing.

The major disadvantage of three tier architectures is the shared responsibility and they result in scalability worse than the one exhibited by *n*-tier architectures. Under shared responsibility we mean handling presentation and business logic simultaneously. The way to solve it is by splitting the middle tier into two. The reasons for this are the complex and rich user interfaces of Web applications. These translate into personalization/individualization and once again scalability. To get a glimpse of what lies underneath, imagine a hundred users using our order entry system, who belong to different groups, and yet get a deeply personalized view of the application.

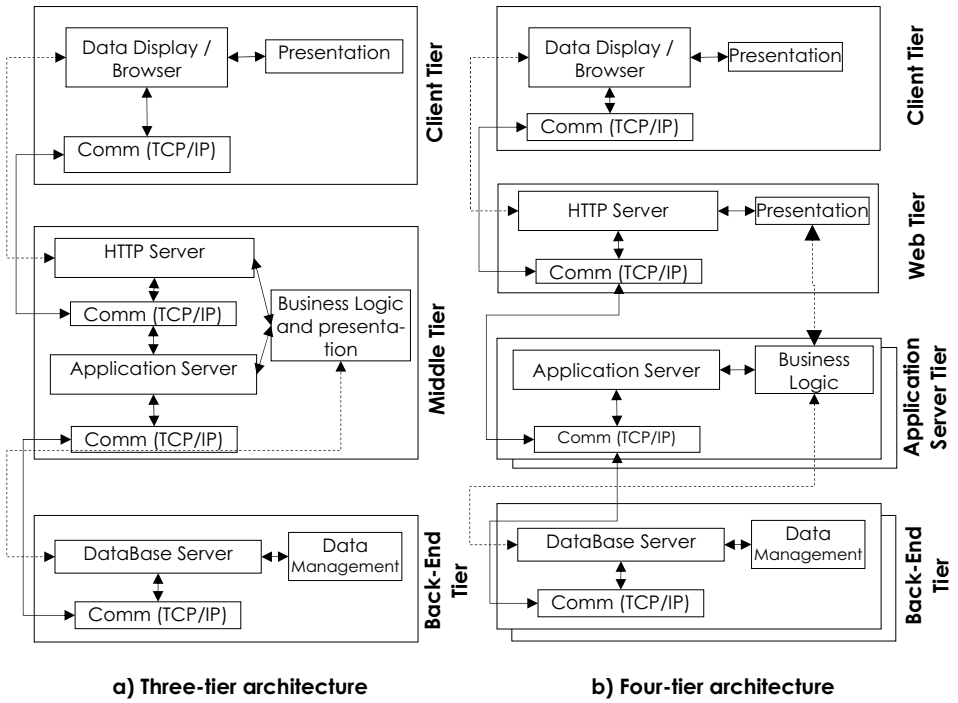


Fig. 2.13. Three- and four-tier platform architectures

The tiers are perceived as layers of abstraction. An interesting property of the layered architectures is that an upper layer can handle many of its immediate lower layers, i.e.

there is a 1:*n* relationship between adjacent layers. A Web server can control a cluster of application servers; each application server can control multiple database servers. This yields improvements in system performance and scalability.

2.4 Web Platform Architecture (WPA)

In this as well as in the next section the components of our Web application framework architecture will be presented. The framework architecture we propose comprises a platform architecture for Web applications, the WAA (Sect. 2.5), and the taxonomy of Internet standards and technologies as the last dimension (Chap. 5).

As we said in Sect. 2.2.1, the application platforms are complex – significantly more complex than stand alone application platforms. To reflect this complexity we prefer to use the more general notion of "infrastructure". Infrastructure cannot simply be reduced to the software platform. It comprises the hardware platform, the software platform modules, and all the configurations, as well. Web applications are still in the early phases of their evolution. This is one of the reasons why the infrastructure plays such a considerable role in Web application design. In our view this is about to change as the technology matures.

The term platform stands for a combination of software modules, which if configured to work together form a basis on top of which various Web applications can be developed, deployed, and executed. Different platform modules serve different purposes and/or implement different technologies. Therefore platform modules can be assigned to layers. For example, an Internet browser will become part of the client layer, an HTTP server belongs to the Web layer, etc.

The modules on each tier provide a set of dedicated services, which an application program can use. We prefer to call these services "capabilities", i.e. the application builds on top of whatever capabilities the platform provides. In this sense we distinguish several kinds of capabilities:

- Communication/networking capabilities – by this we mean the typical networking capabilities (hardware infrastructure and software services) the platform provides to the applications. For example, the typical TCP/IP infrastructure provides socket connection capabilities to applications using its services. The raw socket capabilities may additionally be enhanced by wrapper-functions implemented in software libraries. These libraries are further used in the application code and linked as part of the application. The networking capabilities for Web applications will, in the most general case, involve HTTP connection capabilities.
- Data store capabilities – by data store capabilities we mean the possibilities the platform provides for storing data. These involve writing data into files, storing data in a database by using JDBC and/or ODBC data sources, and so on.
- Visualization capabilities – these involve the technical possibilities the platform provides, allowing the application to depict information graphically. These include text rendering capabilities, drawing capabilities, and various graphics capabilities.
- Logic execution capabilities – these are especially relevant in the context of the Web. On the one hand, logic is something "exotic" in the context of the Web, which was initially designed as an environment for processing text-

based content, e.g. HTML documents. On the other hand, platforms for Web applications must provide possibilities for executing pieces of logic written according to certain open standards. These reasoning results directly form the principles of openness and ubiquity. Typical examples for such standards are JavaScript and CGI.

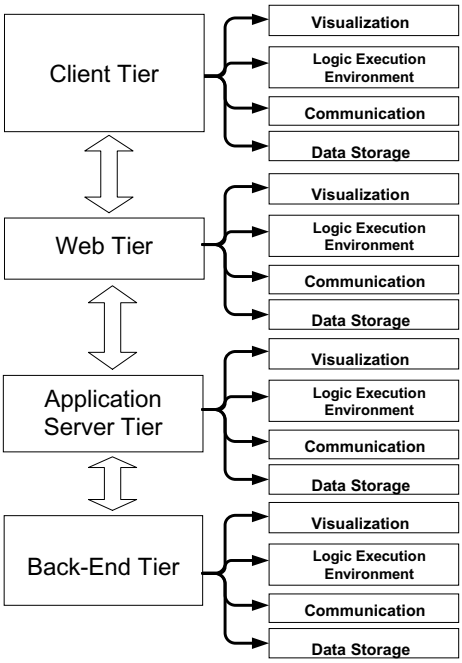


Fig. 2.14. Two, three- and four-tier architecture with capabilities for each tier

One could easily associate these capabilities with the layers of two-, three-, or four-tier architectures. However, in Fig. 2.14 we only do this mapping with a four-tier architecture since this provides the greatest flexibility. A mapping as depicted in Fig. 2.14 shows the principal solution space. For a concrete Web application this mapping has to be reduced to a suitable amount. Let us now explore how these capabilities are distributed over the layers based on the order entry example. Since we will hold this discussion at a very general level, the result will provide some principal statements about the association.

Client Tier	
+++	Visualization
++	Communication
+	Logic Execution Environment
-/+	Data Store

Fig. 2.15. Client tier capabilities

The main task performed at the client side is handling the user events and rendering client side HTML presentation. Therefore the client tier needs to provide mostly visualization and HTML rendering capabilities (Fig. 2.15). If client side logic is used, logic execution environments such as the Java Virtual Machine will also be required. Due to the fact that thin-client tiers are preferred in Web applications, there will be almost no data source capabilities (reduced to client side caching).

Web Tier	
-/+	Visualization
+++	Communication
+++	Logic Execution Environment
-/+	Data Store

Fig. 2.16. Web Tier platform capabilities

It is the Web tier where the presentation (the HTML) files are actually generated (Fig. 2.16). A logic execution environment with considerable performance will be needed, which has implications for the hardware platform. Communication is also a factor since Web tier presentation-oriented business logic actually controls the business logic on the application server tier.

Application Server Tier	
-/+	Visualization
++	Communication
+++	Logic Execution Environment
-/+	Data Store

Fig. 2.17. Application server tier capabilities

The application server tier is where the major part of the Web application's business logic is executed (Fig. 2.17). Therefore significant logic execution capabilities are required, which entail high hardware and software performance. Visualization is important only for administrative and maintenance purposes. All data store capabilities are delegated to the back-end tier (Fig. 2.18).

Back-End / Data Storage Tier	
-/+	Visualization
+	Communication
+	Logic Execution Environment
+++	Data Store

Fig. 2.18. Back-end tier platform capabilities

Figure 2.19 is just another representation of Fig. 2.15 through Fig. 2.18. It shows how the various platform capabilities are utilized by the application components across the

different tiers. The height of the shaded bars indicates how important a capability is on a tier.

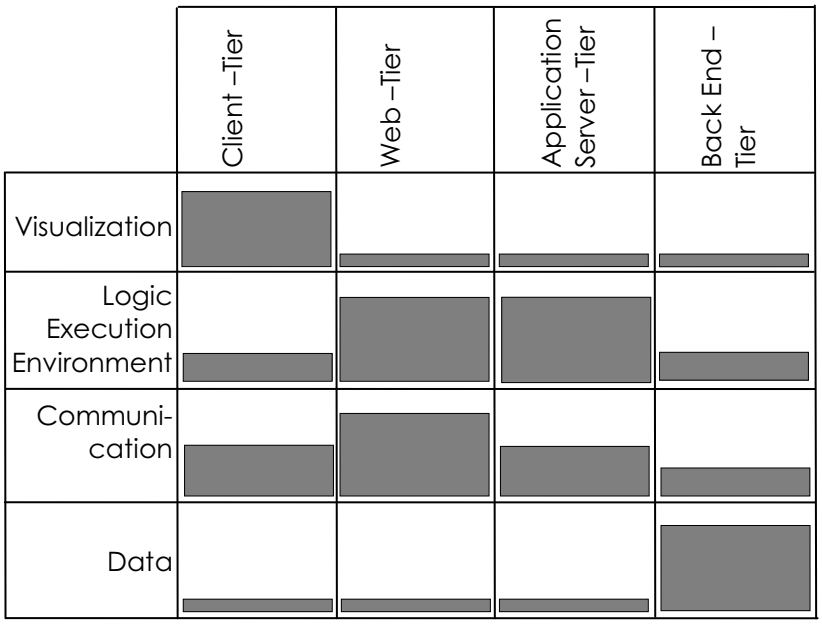


Fig. 2.19. Capabilities and tiers

2.5 Web Application Architecture (WAA)

Web applications are applications, regardless of certain specifics they may have. It is common to treat Web sites and Web pages as something extraordinary, which does not obey the rules of traditional application design. In this section we will try to present a different view – we attempt to approach Web applications as enterprise computing applications, accounting at the same time for their specifics [BaGP00].

By analyzing conceptual architectures we can recognize typical modules (packages, components) very well known from traditional application design. Such packages cover presentation, business logic, data manipulation [HaHP00], [ShSN01]; in other words, user interface, application functionality, and data manipulation.

When treating Web applications in such a "standard" way we must also account for their specifics. The fact that Web applications do not need to be executable (the simplest Web application is a Web page) is a simple but very illustrative example of their specifics. Web applications are primarily content based. The user interface (presentation) consists of mostly dynamically generated HTML files. The navigation is implemented using hyperlinks. All issues regarding the specifics of Web applications are considered in detail in Chap. 9.

2.5.1 Modules of the WAA

In this section we will take a short journey and introduce the different components (packages) of WAA in a step-by-step manner, starting from the simplest case and ending with some complex issues.

Consider for example the order entry system in its static variant (Sect. 2.3.3.2). It represents the simplest version of a Web application – a static Web page (Sect. 2.2.2, Sect. 2.4). This application exhibits an important property – it is complete, in the sense that it can be deployed, and operational.

By closely analyzing it we can distinguish two components – business logic and some data management (Fig. 2.20). The static version of the order entry system is analogous to the simple C/C++ order entry application we considered in Sect. 2.2.1. The C/C++ application business logic uses the printf() system function to display textual data (the "Order Entry Example" character string in this case), which fetches the string and sends it to the standard output device. The string itself is handled by the data management module. The Web application business logic is triggered by a user clicking on a hyperlink. It is automatically translated to HTTP commands (requests), instructing the Web server to fetch the desired data, in this case the order entry HTML file, and send it back to the client as a response for display. The HTML file is handled by the data management.

To model the architecture we use UML packages (Fig. 2.20). The business logic in this example stands for "classes" handling the requests, and data management stands for the actual HTML page and its content.

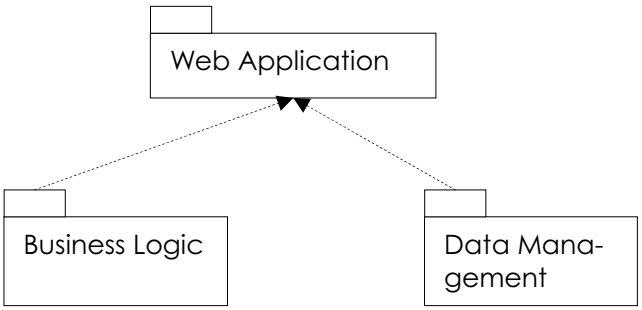


Fig. 2.20. WAA – case 1

Such simple static Web pages are rather an exception than the rule. Static Web sites are difficult to maintain, and dynamic content cannot be embedded. The large majority of Web applications nowadays utilize complex dynamically generated Web pages as is the case with the second variant of the order entry example (Sect. 2.3.4).

A new package appears, the presentation (Fig. 2.21), handling all the issues of HTML generation. Some applications tend to consider the presentation functionality as part of the business logic, which is a poor architectural style since they serve different purposes. The name "presentation" is not arbitrarily chosen. It stands for the fact that all user interface pages are generated by the server side application package presentation (and are just rendered at the client side). In practice the presentation package can become extremely complex – imagine for example a full-scale order entry system with thousands of pages.

In the order entry example, for instance, the presentation module will generate the dynamic HTML file containing all pending orders, upon user request. A close association with terminal-based (Sect. 2.3.2) systems can be made at this stage.

The business logic (Fig. 2.21) has a twofold function. A small part of it handles user requests (Sect. 2.5.2). The largest part of the business logic, however, contains the implementation of Web application-specific functionality. In the order entry example the business logic will enumerate all order entries using the data management module, will then filter just the pending ones, and will next invoke methods on classes in the presentation package.

The "data management" package (Fig. 2.21) handles the retrieval of data and manipulation. If for example the designer decides to use sequential files instead of relational database systems as the data store, the data management module will be where all data will be processed, and where create, read, update, and delete operations will be implemented (CRUD operations). Roughly speaking, the data management module will play the database management system.

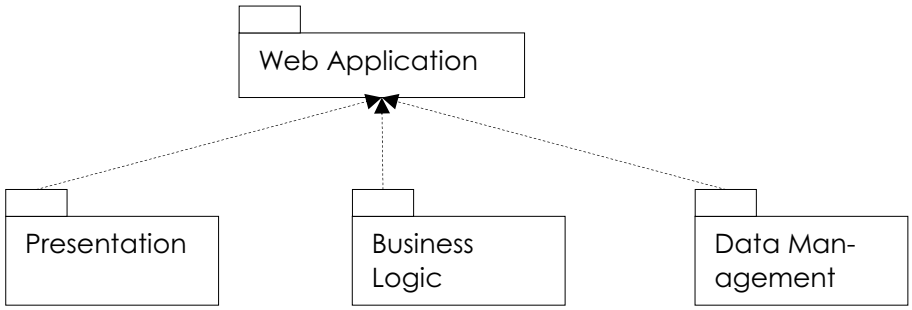


Fig. 2.21. WAA – case 2

Let us continue to extend the order entry management example, by utilizing a relational database system as the data store. This will require a significant change in the data management module. However, no change in the rest of the application will be necessary. A new package to handle the communication between the DBMS and the data management module is needed. We prefer the term interaction, to avoid terminological collision with the platform capability "communication" (Fig. 2.14). In general interaction will be used to handle the communication among different application architectural components. Remember that WAA components will be mapped onto technologies and later onto WPA modules, which are organized in layers, so interaction is needed to bridge the components on different layers.

Specific communication can be used for interaction between presentation and business logic between business logic, data management, etc. The interaction component needs to provide support for transactional management, user sessions, and security (encryption).

Clearly security is a very important aspect in the context of Web applications. Security aspects must be implemented by the platform modules, and provided by certain technologies such as SSL, digital signature, and so forth. Security, however, must also be reflected in the application architecture in the context of user management and access rights. Interestingly enough, application-level security can be designed in two ways – as

an integral part of each WAA module, or as a separate WAA module (Fig. 2.22). We prefer the latter.

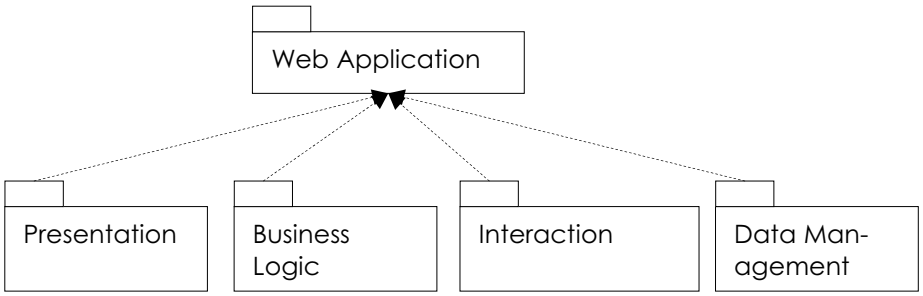


Fig. 2.22. WAA – case 3

Having a good user policy is a characteristic of any modern Web application has. It is tightly associated with security. Assigning a user access rights for different resources, capturing user preferences, and modifying the presentation with respect to them is a typical personalization task. To handle all these issues we introduce the WAA component personalization (Fig. 2.22). The issue of modeling users, user groups, and whole organizational structures will be discussed in Chap. 11.

As we already mentioned in the previous sections, Web applications are content based. The contents of all HTML Web pages, e.g. text, formatting instructions, pictures, are simply treated as information which should be displayed. There is, however, limited means to introduce "meaning" to the contents. For example, in our order entry system there will be a lot of textual information about the company that placed a certain order. Such information may include things such as e-mail, phone number, and address. All these elements will be treated as simple text, with no special semantic meaning, i.e. the company addresses are treated as plain text and not as addresses.

Recently the World Wide Web Consortium introduced Semantic Web as a technology for describing the content. We introduce the WAA component description (Fig. 2.22) to reflect the content description issue in Web applications.

Last but not least, we will consider one of the newest development trends in Web applications. It reflects the fact that business logic of a can be exported for use by third parties and conversely certain Web application can use (import) functionality from other applications. To reflect this we introduce the WAA component "Export/Import" interface (Fig. 2.23). This idea is gaining in importance with the introduction of Web services. They provide an excellent (platform-neutral, Web-based, loosely coupled) way of using the business functionality of other Web applications. This is how the business logic can be "licensed" to other applications and used on pay-per-use basis. There are numerous examples of services already existing such as the Web service from Amazon.com [Amaz04] or from Google.com [Goog04]. The "Export/Import" interface together with Web services provide an outstanding basis for Web application integration. For more information see Chap. 7 which is dedicated to Web services.

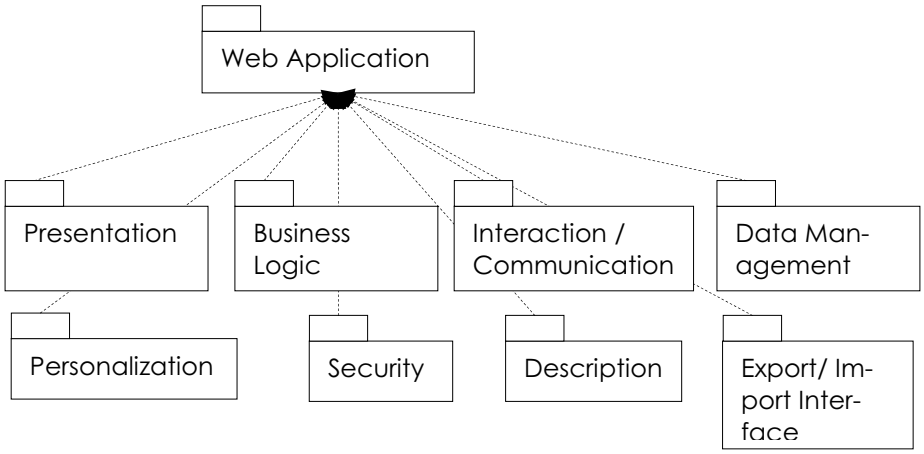


Fig. 2.23. WAA – case 4

The list of components for the WAA could be extended further. For instance, search and discovery functionality could be added, which becomes more and more relevant for Web applications. The idea is to be able to place the interfaces exported by a Web application into a registry so that other application providers can find it and use it. Such a WAA component would complement the export/import interface component. This discussion is continued in Chap. 10 and throughout the chapters of Part III of this book.

It is not our intent to provide a complete list of modules in this section. This is not possible since each application itself determines what modules are essential for it. Nevertheless, we seek to compile a list of modules relevant for most Web applications. Modules describe the functionalities required to enact an application. These functionalities determine the technologies to be chosen according to Fig. 2.5 and Fig. 2.6.

2.5.2 Example: The Model–View–Controller Design Pattern

WAA is independent of technology and platform – it is a conceptual architecture. WAA is where design patterns can be applied. Design patterns represent abstract, predefined, technology-independent solutions to standard architectural problems. Per se design patterns are mini-architectures, which are meant to be applied on conceptual application architectures. Since WAA is an abstract architecture, this section illustrates how design patterns can be applied in WAA. As an example, the model–view–controller (MVC) design pattern is considered in more detail. MVC is very suitable for Web applications (but not only) and is implemented in any dominant design paradigm like J2EE [SiSJ02] [BMR+96], and .NET [TMQ+03].

The main idea is introduced in [ShSN01] (other good sources are [GHJV97], [SiSJ02], and [TMQ+03]). The key issue is separation of concerns among three parties – model, view, and controller. The model handles the business logic and the data management. It is also assumed that the model is relatively static; otherwise changes in the model will trigger massive changes in the view and the controller, which must be avoided. In our architecture the model combines the functionality of the business logic and data management packages. The view handles the presentation. It is equivalent to a

combination of the presentation and personalization components in our architecture. In a Web application the view will be responsible for HTML output generation under the preferences of a specific user for example. The controller is the part of the business logic processing the user generated events. These events are translated into requests. The controller processes the requests and performs the corresponding calls of model operations.

The major issue the MVC design pattern solves is that it provides a way to organize the user interface (user interaction) of a Web application so that possible changes in one module affect minimum parts of other modules. Its usability is determined by two factors: the frequency of user interface changes or customizations, and the number of different concurrent views of the same data, which an application must provide. The former has to do with the degree of personalization. The latter has to do with the need to coordinate and update the multiple views. All these factors are especially favorable in a Web environment since the user interface is generated; thus it must not be redistributed and redeployed on the client side platform.

The MVC pattern is independent of any technology and is also independent of the application type (whether standalone or Web application). It can be realized with the large majority of programming languages. Different frameworks, e.g. J2EE, .NET, include technologies to implement MVC. In J2EE, for example, the view will probably be realized as JSP, the controller probably as a Java servlet, and the model probably using EJB. In [TMQ+03] the reader can find an implementation of MVC for the .NET platform in C#.

To recapitulate, MVC is a design pattern defining the general organization for personalization and presentation-intensive application architectures. In this respect it is especially useful for Web applications. The application of MVC regroups WAA packages as considered in the sections above. For example, the packages of presentation, personalization, and description (Fig. 2.23) are part of the view; the packages of interaction, export/import interface, and parts of the business logic are part of the controller; and the packages of business logic and data management are part of the model. This example shoes how WAA is influenced/changed by the application of a design pattern.

2.6 Requirements for a Framework Architecture

There are three major characteristics exhibited by the proposed Web architecture – general applicability, extensibility, and comprehensiveness. General applicability in this context implies that the proposed architecture must subsume as much of the currently existing architectures as possible, which implies an open architectural style. To model all different facets of a Web application we consider two issues. On the one hand, we consider the differentiation between WAA, WPA, and the technologies presented in Fig. 2.2. On the other hand we introduce different aspects of both WAA and WPA. The concrete goals we are aiming at include:

- comparison with related approaches based on the classification;
- discovery of technological gaps in the existing landscape and finding possible alternatives;
- identification of new integration concepts among the technologies based on the recognized gaps and an architectural comparison.

Extensibility is the second characteristic of the framework architecture which has to do with how easily it can be extended to handle future developments. The ease with which users can adapt the framework architecture to the concrete problems they have is

also an integral characteristic. The visualization capabilities provided by a Web application platform are a typical example. Normally one would expect that visualization services are mainly required at the client side. Therefore the designer might ignore the presence of a visualization layer on the server side. However, in complex Web applications, due to administration and support tasks visualization services may also be needed on the server side. Therefore the designers must be able to extend and adapt the framework architectures to their own purposes thus deriving their solutions from the general architecture.

The third characteristic of the framework architecture can be summarized in a single word – comprehensiveness. Firstly, it has to do with a suitable way to represent/visualize most of the facets of the Web application design. Secondly, it has to do with partitioning the problem space and defining the given technologies and platform modules in a way allowing us to achieve order. By order we also mean a way to self-validate the design. Last but not least, we use comprehensiveness in terms of a clear step-by-step approach towards building Web applications.

2.7 Guide to the Rest of the Book

This chapter introduced new concepts in designing and developing applications. It motivated the historic evolution. The modules of the platform for Web applications were discussed, and notions like Web server and application server introduced. Above all, this chapter introduced the concept of WAA.

The main message the chapter conveys is that the essence of a good Web application design is the separation of concerns between platform, application architecture, and the technologies used for the implementation. The separation implies that these three constituents are designed independently, one at a time. A sequence of mapping steps ensures the transformation among them.

What are the advantages of such an approach? Firstly, clarity of the design and conceptual separation of the different application aspects are achieved. Secondly, a single application architecture may be implemented with different technologies and on many platforms. Last but not least, the special characteristics of the platform modules and technologies do not influence the application architecture. The choice of platform modules may not have any influence on the choice of technologies. Many of the existing Web application design approaches do not account for this issue. The major disadvantage of this approach is the higher complexity. The high investment in design effort pays off only in the case of large applications.

This chapter is instrumental for the rest of the book. It defines the constituents of the framework architecture such as WAA components, WPA modules, classification, etc. These are used to define the architecture of the designed Web application, and therefore are always given or “static”. The Web application is designed “dynamically” in a step-by-step procedure, called the stepwise design approach, described in Chap. 3 and continued in Chap. 4. Chapter 3 concentrates on a technology-free Web architecture. Chapter 4 shows how to define the classification of Internet technologies and standards and how to select implementation technologies and platform modules. Chapter 5 describes technologies used conventionally in developing Web applications. Chapter 6 dwells on middleware and component-oriented technologies. Chapter 7 is dedicated to another technology referenced in Sect. 2.5, namely Web services. As explained in Sect. 2.5, the features of

Web applications with respect to HTML interface design, navigation linking, etc., are parts of a lager tropic called Web engineering, which is addressed in Chap. 8.

As already mentioned, the Web application framework architecture proves especially useful when designing complex Web applications. Analyses of existing systems show that a number of techniques such as organizational modeling, process management, and use of registries can be successfully applied in this context. All these issues are considered in chapters of the third part of the book.

3 Developing WAA and WPA

While Chap. 2 has introduced the architectural framework for the development of Web applications, this chapter will demonstrate how to develop a concrete Web application within this framework. So to say, Chap. 2 has defined the skeleton for such a development, i.e. the WAA and WPA were generally introduced. This chapter is going to fill this framework with concrete components (WAA) and layers (WPA). Last but not least, the architectures defined in this chapter must be associated with technologies; this happens in Chap. 4.

This current and the next chapter therefore are dedicated to the stepwise approach to Web applications. The step-by-step methodology will be illustrated on the basis of the order entry example defined and discussed in the previous chapters. Eventually some of the related Web application design approaches will be presented.

3.1 Introduction

Chapters 3 and 4 will describe a stepwise approach to the development of Web applications. This section provides a short overview of the steps of the design approach. It summarizes not only the sequence of steps but also the goals and the results achieved in each step.

Our overall Web application design approach is inspired by the idea of separation of concerns. Three different dimensions are distinguished: architecture of the platform (WPA), architecture of the application (WAA), and the technologies with which the application is implemented and which must be supported by the platform. Specifics can be better accounted for due to the separation between the platform and application architecture. The separate consideration of technologies allows a much richer and technology-independent design.

From everything read till now the reader may be under the impression that the approach we propose is suitable only for Web applications developed from scratch. However, it is equally applicable to legacy applications. It can be used to reconstruct (reverse engineering) the architecture of older applications. Partly, the utility of the proposed design approach is to provide new ideas regarding conceptual architecture or to help to evaluate and compare existing designs.

The goal pursued in our stepwise approach is to have a systematic way of designing Web applications, guiding the designer through the process of synthesizing Web applications. This design process is an iterative one. The designer may start with a simple case and then make a number of iterations throughout the whole architecture. On each iteration some extensions may be made, new modules or features may be introduced, and then requirements and design decisions from previous iterations may be tested. Briefly the stepwise approach passes through the following steps.

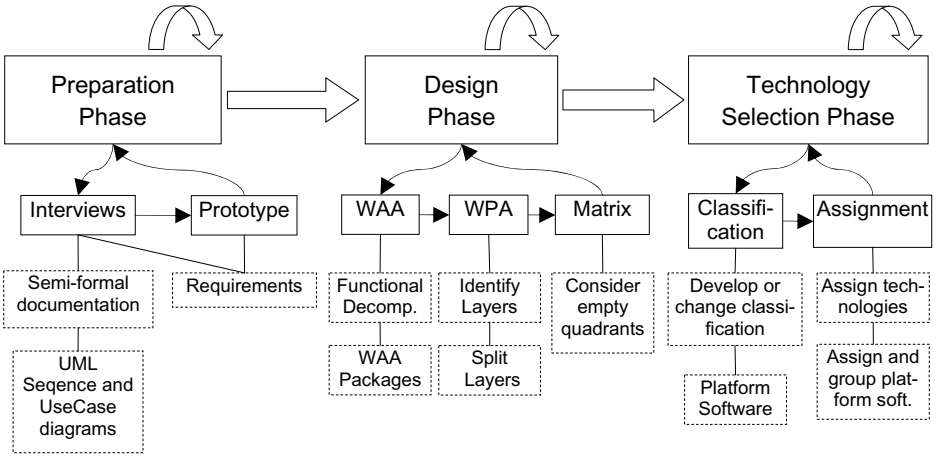


Fig. 3.1. Web application development – stepwise approach

The stepwise approach to Web application development consists of three phases (Fig. 3.1):

1. Preparation phase (Sect. 3.2): The preparation phase aims to discover user requirements, provides information about the tasks the application must implement, and a general feeling as to how it is expected to function. The designers hold a series of interviews with the clients, investigating their requirements. The results are documented as use case and sequence diagrams. These will be further used in the design phase as a basis for both application and platform architectures. The preparation phase of the Web application development ends up with the creation of a prototype. Its purpose is to validate all results gained in this phase. It is recommended that the designers make several iterations, increasing the degree of complexity and the level of detail gradually.
2. Design phase (Sect. 3.3, Sect. 3.4, and Sect. 3.5): The design phase follows the preparation phase. From a conceptual point of view the design phase represents the most important step in the design process. The major goal to be achieved here is to design the Web application architecture (Sect. 3.3), to develop the Web platform architecture (Sect. 3.4) and to construct the matrix (Sect. 3.5). The WAA represents the architecture of the application functionality. It is derived by functional decomposition of the use case diagrams (see preparation phase) into classes and grouping them into packages prescribed by a generic WAA. Some of the interactions among the classes and packages can be derived by sequence diagrams. To facilitate the transition from the WAA to the WPA all WAA components are assigned capabilities, e.g. presentation, logic execution, communication, etc. Capabilities stand for generic properties of a set of platform modules. The WPA is organized into layers. It may represent a two-, three-, four-level (or more) architecture. The process of mapping the WAA onto the WPA is iterative. It starts from the two-tier architecture. To evaluate a set of criteria, during each iteration the

designers take a decision of whether and how to split a layer into two. Constructing the Web application framework architecture matrix is the final step in the design phase. The vertical dimension of the matrix is the WPA; its horizontal dimension is the WAA. The goal is to easily identify empty fields in the architectural space and to validate the overall architecture.

3. Technology selection phase (Chap. 4): The last phase is the technology selection phase. The application architecture developed in the design phase is technology independent. The goal of this phase is to choose a proper set of technologies and assign them to the entities in the matrix quadrants (Sect. 4.2). By technologies actually both technologies for the WAA components and technologies for the WPA modules are meant. Using a classification of internet standards and technologies is the cornerstone of this approach (Sect. 4.1). It facilitates the selection process, providing alternative suggestions and directing the decision focus to a specific group of technologies. The classification and the technologies themselves impose restrictions and dependencies, facilitating the design process. Another set of restrictions can be derived from the requirements investigated in the preparation phase. All these sets of restrictions help to reduce the many "degrees of freedom" present in the technology-free architecture created in the design phase.

Maintenance and support are major issues of the Web application lifecycle. In reality, major investments are made into systems that minimize maintenance costs, and into buying support options. Although application designs can be evaluated with respect to such factors, it is relatively difficult to reflect on them throughout the development process.

On completing the four main phases of Web application development, an iteration phase follows (Sect. 3.6). Within this phase the designs produced within the former phases are refined. The approach provides a way to discover weaknesses and incomplete solutions in the created WAAs. In other words, the designers are provided with some means to self-validate the architectures they have created.

The proposed approach is aligned with the state-of-the-art, model-driven application development paradigm [KlWB03] [Fran03]. It covers many aspects of the model-driven application development; however, it does not fully fit into it. Another issue worth mentioning is the UML compliance. This approach makes extensive use of UML, but it is not completely UML based.

3.2 Preparation Phase

A number of preparation steps needs to be taken before the designers can start the actual application development. These typically involve a detailed study of the problem domain, investigating clients' and users' requirements, and prototyping. The common software engineering term describing this phase is "requirement engineering" [Somm00].

The traditional software engineering methodologies define the requirements inception phase as consisting of a number of steps: feasibility study; requirements capture and analysis; requirements definition; requirements specification; prototyping – briefly the waterfall model. A feasibility study is used to test whether the system or application can be built, and whether it will be cost effective. Requirements capture is the process of deriving requirements, possibly by observation and interviews. Requirements definition aims at abstract formation of the captured requirements. Requirements specification produces a detailed, precise, and formal document of the requirements [Somm00]. The ulti-

mate goal of the prototyping phase is to validate the requirements specification and possibly trigger a new iteration.

The approach described in the next chapter is closely aligned with traditional software engineering and with object-oriented application development. There are, however, a few minor differences. A feasibility study is beyond the scope of this chapter and is therefore not discussed. Requirements capture and analysis and the requirements definition phases are considered as one step although they are clearly separate.

3.2.1 General Remarks

A study of the broad problem domain and the environment of the application must be carried out during the initial phase. Such a study involves the expected number of users, the formation of user groups with respect to personalization, user requirements, or different capabilities required from the system. The designers must also define an assessment regarding the expected performance, scalability, and distribution, which will then influence the choice of communication lines, hardware platform, and software platform.

The designers must prepare a list of the essential requirements and characteristics of the application. This is an interactive process involving many interviews, extensive communication, and documentation. Finding the true characteristics is a key issue since they may not be directly specified by the client. For instance, the client may require an appealing user interface, but it is the designers' task to mention the issues of portability, communication, and maintenance. Some of the relevant factors that need to be considered at this stage involve:

- Architectural issues – issues related to the expected approach, the number of layers and distribution, and also integration with existing systems.
- Implementation requirements – factors mainly focusing on the expected technology and the related platform modules but also focusing on possible extensibility.
- Operational details – target the clients' expectations regarding the system performance, its maintainability, and the ability to migrate some of its components to different platforms.

Experienced designers carry out interviews with clients or some representative potential users to estimate the degree to which all these factors will influence the architecture. There are many more pieces of valuable information which can be extracted during this phase where designers must get a feeling on how the application is expected to function.

UML is an established modeling language containing diagramming mechanisms such as UML use case diagrams and sequence diagrams (Fig. 3.2 and Fig. 3.3), which can be successfully used here. In fact, the designer can use any semi-formal method to capture and document the information gathered at this stage. The authors recommend an alignment with UML.

Last but not least, a rough-cut prototype of the system may be prepared during the preparation interface. It implements the majority of use case diagrams and gives the client a perception of how the application will function and how the users will interact with the system (Fig. 3.6). Interestingly enough, such prototypes are relatively simple to create in the realm of Web applications.

3.2.2 Preparation Phase for the Order Entry Application

This section shows how the preparation phase has to be performed using the example of the order entry application. We have chosen a two-iteration approach. In general, the designers have to determine the number of iterations necessary. The following rule of thumb holds: the more complex a Web application becomes, the more further iterations are recommended.

3.2.2.1 First Iteration

Some of the characteristics required in the order entry application include:

4. A wide range of users operating with different client computers and devices running a heterogeneous set of software – this issue translates into interoperability i.e. compatibility with as many browsers as possible and a thin-client architecture.
5. If possible, the user interface should contain some nice-looking elements such as buttons – this translates into simple and widely supported client side scripting.
6. Frequent order entry list changes – this maps to the usage of a database as a data store and is a mechanism to notify clients and refresh their state–session support.
7. Extensibility – new functionality and frequent changes in the page layout have to be considered. This translates into the use of scripting approaches in the presentation package and a component-oriented approach for the business logic.
8. Future support of personalization – this translates into the use of dynamically generated pages. Additionally, future support for authentication security technologies is necessary.

After carrying out a series of interviews, the way the system must operate can be determined. As a first step a use case diagram of the client side interaction is created, in which just the sequence of client side actions and the resulting Web pages are considered. In a real system such diagrams will be significantly more complex involving many user interaction steps; therefore, starting with a simple client side view is advantageous.

Two use cases are shown in Fig. 3.2 – the "Start" use case, which is also extended by the "Generate Pending Order Entries List" use case. The "Start" use case involves steps to start the application by opening the URL http://oderentry-example.de/input.html and generating the start page. This use case may also be extended to include some authentication and personalization activities which we disregard for the time being.

The "Generate Pending Order Entries List" use case is more complicated and will be modeled at full scale during the next iteration. At this stage it is sufficient to register the need for such a use case in order to account for the fact that the pending order entry list page will be dynamically generated.

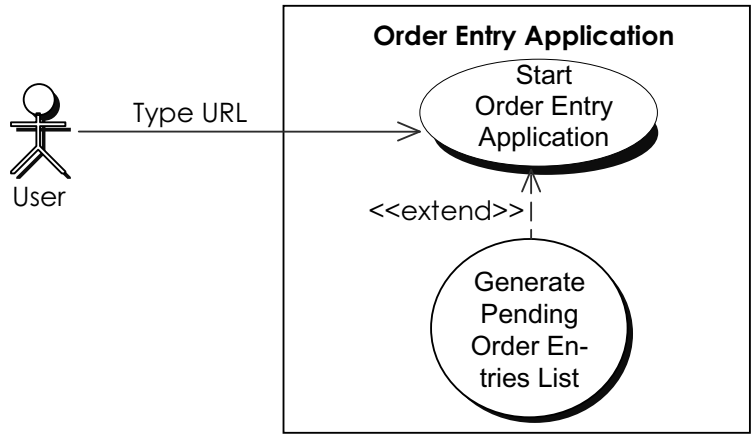


Fig. 3.2. Client side use case diagram in the first iteration

The sequence of high-level interactions shown in Fig. 3.3 includes two round trips. The first one (steps 1, 1.1, and 1.2) describes the loading and generation of the start page, while the second round trip (steps 2, 2.1, and 2.2) describes the generation of the pending order entry list. Clearly it is described as a sequence of too coarse-grained steps. Therefore designers need to go through a second iteration in order to extend it.

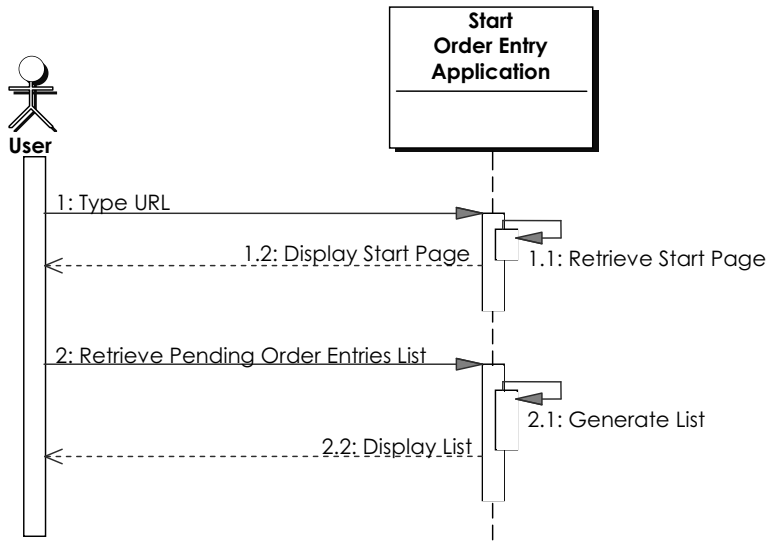


Fig. 3.3. Simple sequence diagram for the use case "Start Order Entry Application"

3.2.2.2 *Second Iteration*

Once the sequence of user interactions is roughly agreed upon, the designers can extend the diagrams to include the overall application, i.e. not only the user interaction but also the server side operations. As you can easily imagine, the use case diagram grows fast as the designer models a real system. It is therefore important to find the right level of detail. In other words, do not document every single step; rather consider general use cases and use notes to document the task each use case performs.

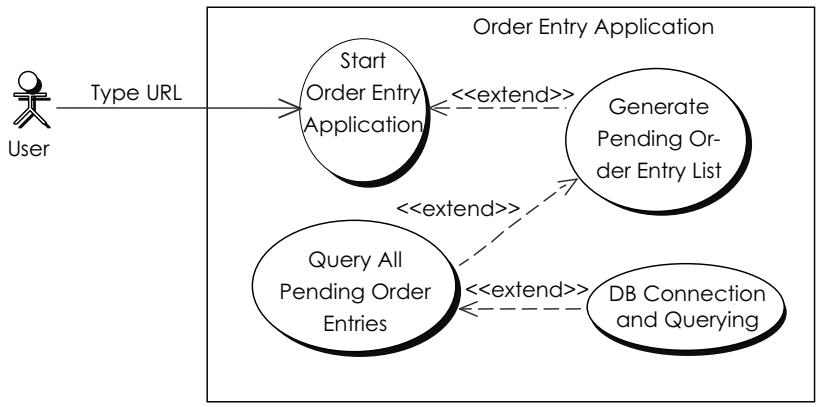


Fig. 3.4. Extended use case for the order entry application

The "Generate Pending Order Entry List" use case can be extended with the use case "Query All Pending Order Entries" (Fig. 3.4), which is in turn extended by the use case "DB Connection and Querying". The use case "Query All Pending Order Entries" describes how a query is formulated, parameterized, and executed. The use case "DB Connection and Querying" reflects the specifics of the concrete data store. It involves a set of activities serving as a layer of abstraction. The corresponding sequence diagram is shown in Fig. 3.5.

It can be easily seen how the two round trips from Fig. 3.3 are extended involving more entities. The first round trip (steps 1 through 1.2) describes the retrieval of the start page (Fig. 3.6). The second round trip is significantly more complicated (steps 2 through 2.2). It describes the actions the order entry application must execute in order to generate the list of order entries once the Generate List button (Fig. 3.6) is pressed. The application processes the `generatePendingOEPage` request, which involves querying all available entries and filtering the pending ones. Having a list of pending entries the application "serializes it in HTML", i.e. an HTML page containing a table with all the data that is generated. The page is then sent back to the browser for display.

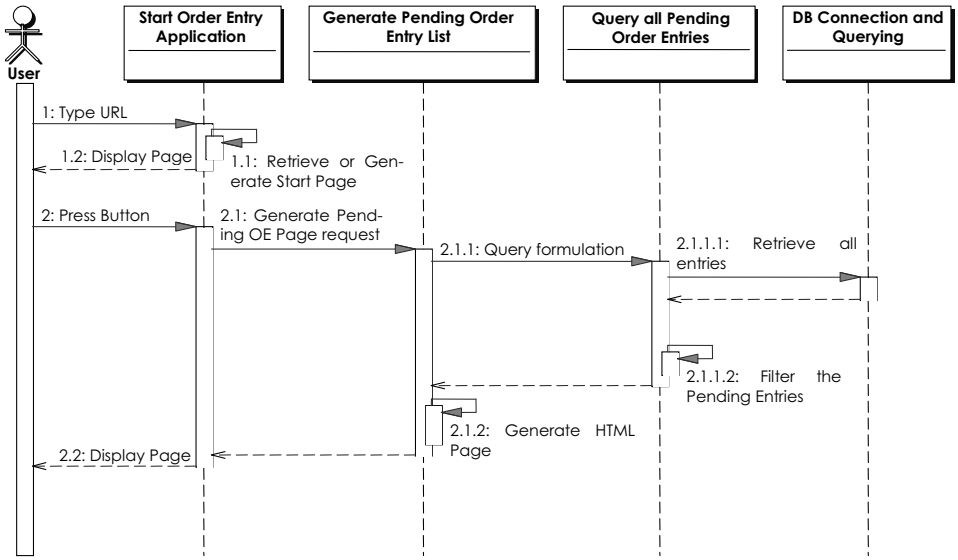
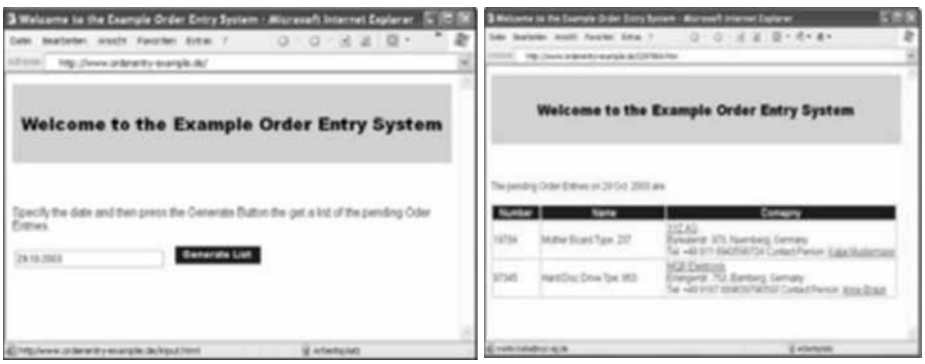


Fig. 3.5. Extended sequence diagram for the order entry application

After documenting the application's principle of operation, the designers may discuss it with the future customers. Therefore it is a good idea to use a partially complete prototype of the system. Web applications are relatively easy to prototype since most of the pages, generated as a result of system operation, must simply be substituted with static ones. With a prototype the user interaction (page order, alternative paths, information displayed) and the principle operation of the Web application can be tested. If personalization is required, the designers may get a feeling of how it is implemented. Figure 3.6 shows a prototype of the application. Figure 3.6a shoes the start page with the Generate List button, which is implemented with client side logic to apply the bevel-out effect. Once it is pressed a sample order entry list is generated (Fig. 3.6b).



a) Start page of the order entry application b) Generated pending order entry list

Fig. 3.6. Order entry application prototype

Figure 3.6b results from the operation of the application which does not exist at that time; therefore, at the prototyping phase it is implemented as a static HTML page pointed to by the Generate List button.

3.3 Design Phase: WAA

Diagrams such as the one in Fig. 3.4 serve as a basis for the WAA. It is a good idea to group most of the use cases under the "predefined" WAA packages. To do so we tag each action with the WAA package name (Fig. 3.7).

When developing the WAA we try to stick to the general packages defined in Sect. 2.5.1 (e.g. presentation, logic), but do not consider them as mandatory. Eventually some of them will be discarded or new ones may be included. For example, if a presentation-based Web application is required then it is likely that the export interface package will be "ignored".

Of course the WAA is far from ready at this stage. The current structure is simply a more formal representation of some of the requirements and the actions to be taken. At this stage we propose leaving the UML realm and using a more loosely but still formal notation, to simplify the transition from UML use cases to the WAA. Instead of using class diagrams and packages immediately, we propose to do a first iteration drawing some special entities, containing objects from the WAA tagged with the corresponding WAA package. The goal we pursue at this stage is to refine the structure of the entities by grouping them together under classes and subsequently forming packages.

Consider for example the "Order Entry Start Page" entity (Fig. 3.7), whose existence was hinted in Fig. 3.4 and Fig. 3.5. It represents an object or a class of objects logically belonging to the presentation package. It is generated by the presentation package and therefore the <<generate >> relationship is used.

The "Generate Button" box is part of the start page (connected with <<include>>); however, it represents a piece of business logic because it issues the command for generating the list of pending order entries processed by "Process List of Order Entries". It results from the initial (Fig. 3.2) and the extended (Fig. 3.4) versions of the use case diagrams. It is registered as part of the business logic WAA package because it clearly has to do mostly with processing and querying, i.e. with the major part of the core functionality.

Opening the database connection and the database schema boxes can be derived from the extended use case (Fig. 3.4). They are part of the WAA interaction and data management packages respectively.

At this stage we start to gradually return to the UML notation. Now the packages can be filled in and refined to contain the corresponding classes generating the pages in Fig. 3.8. Two new classes appear in the presentation package – namely, `GenStartPage` and `GenOEPage`. They represent the functionality required to generate the two HTML pages. Of course modeling the generators just as classes represents an oversimplified solution to the problem. Later, when we consider the choice of technologies we will assign concrete technologies to the respective packages and classes.

The business logic package contains two classes – `RequestHandler` and `PrepareOEList`. A different possibility which is not shown in Fig. 3.8 is to introduce a subpackage called `ClientSideLogic` to account for the logic behind the Generate button.

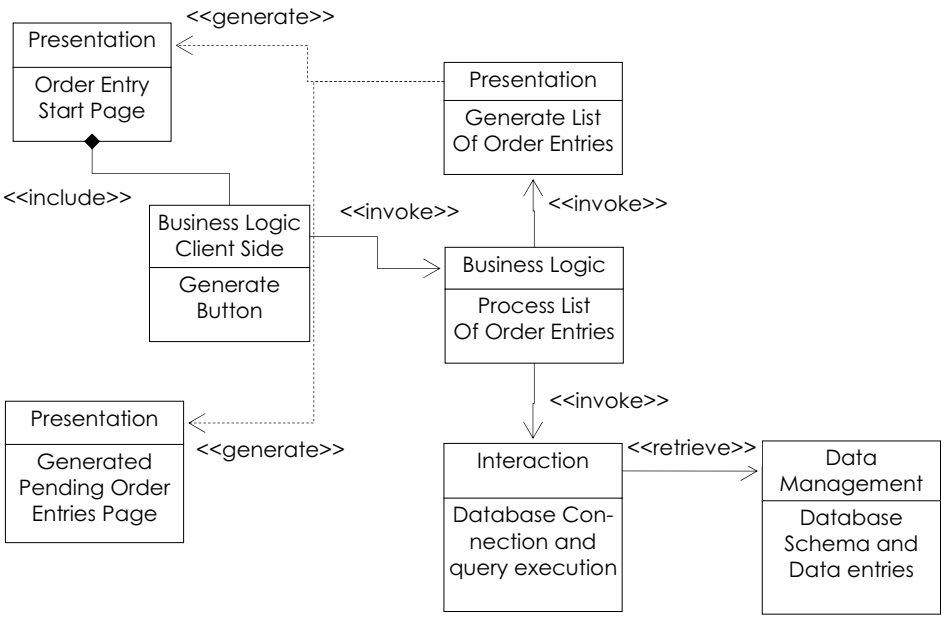


Fig. 3.7. Initial WAA

The `PrepareOEList` handles the business logic required to process the existing order entries and filter the pending ones. For the time being the `RequestHandler` class will be left out. `PrepareOEList` must open a connection to the data store in order to retrieve the order entries. To do that, the `PrepareOEList` class uses the `DataStoreConnection` class to perform this task. The `DataStoreConnection` class implements data store specific communication, and controls the query execution and partly the query formulation.

This is the right time to apply design patterns. Analyze which subproblems are standard, choose the proper design pattern solution, and model the appropriate classes. Consider for example the `Interaction` package: the `DataStoreConnection` class is the ideal place to apply the Iterator pattern. This is also the right place to consider the `RequestHandler` class. Due to the extensibility requirement it is important that the Order Entry application implements the MVC design pattern (Sect. 2.5.2). The `Presentation` package implements the View part of the pattern, the `PrepareOEList` class the `Interaction`, and the `DataManagement` packages implement the model part. To implement the Controller part, however, we need to call the `Business Logic` package handling the input requests. This is why the class `RequestHandler` is introduced.

Always bear in mind that your WAA can be implemented with different technologies (and different programming languages) so avoid using technology-specific modeling. As a first step towards constructing WAAs, consider an intermediary diagram. Iteratively construct the WAA packages and model the corresponding classes.

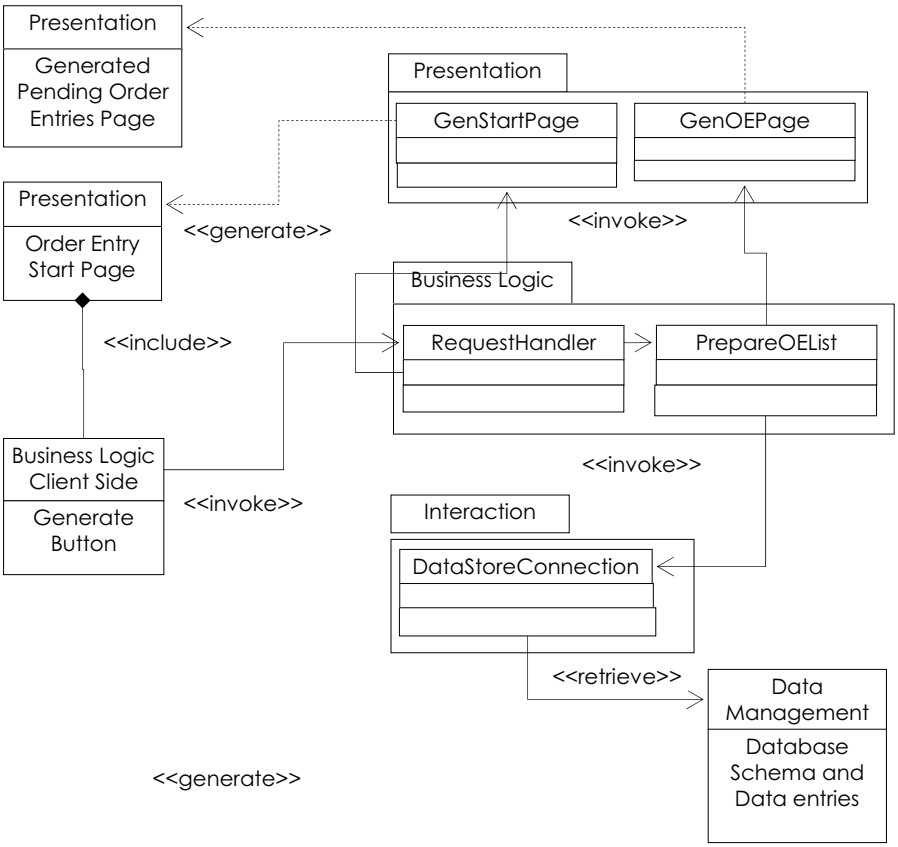


Fig. 3.8. Refined WAA

3.4 Design Phase: WPA

In Sect. 2.4 we defined the conceptual architecture of a Web application platform based on layering. The platform was defined as a set of modules providing different capabilities, organized into layers. In this section we will show how to iteratively map different WAA components onto the WPA layers. The basic principles are:

- To think in terms of topology and layering.
- To start off small but aim at large applications.

Now the designer needs to solve the issue of assigning WAA components to WPA layers. The issues of caching and distribution (data and business logic) represent exceptions to the general guidelines introduced above.

3.4.1 Split Criteria

One of the essential operations the designer has to perform during platform design is to split a tier into subtiers. The goal of splitting is to resolve the issue of sub layering within a tier and thus to make a layered design separating WAA packages in a clear way across tiers.

The existence of an `Interaction` package representative on a tier is a clear sign for sublayering, which must be avoided. This conclusion is of such importance in the step-by-step platform design that it will be termed the interaction principle.

Another criterion is the existence of a stack of presentation and business logic classes in a single tier. In this case the designer must spilt the respective tier and reorganize the packages in an appropriate way. This is a relatively soft criterion since it relies on the designer's expertise and provides no objective inductions.

3.4.2 Two-Tier Platform Architecture

Let us start our discussion by considering a minimal solution represented by the two-tier architecture – a client tier and a server tier. It can be illustrated by slightly changing Fig. 3.9 – in other words, by just merging the middle and the back-end tier into a single server tier (the rest remains unchanged). Typically all modules handling immediate user interaction (or are client specific) will be placed in the client tier. All presentation and business logic (for the time being) will be placed in the server tier. All data-related modules will also be placed in the server tier.

There are two factors motivating the transition to a three-tier architecture. The first one is the interaction principle – the `Interaction` package (`DataStoreConnection` class). The designer must question the need for a database system as a data store at this stage. The database system is motivated by requirement number 6 (Sect. 3.2.2.1).

The second factor substantiating the transition to a three-tier platform architecture is a more functional one. Two tier Web application platform architectures are especially suitable for simple Web sites consisting of static Web pages. The order entry application does not have this static character because of the requirements 6 and 7 (Sect. 3.2.2.1).

3.4.3 Three-Tier Platform Architecture

As motivated in the previous section the server tier will be split into two new tiers – the Web tier and the back-end tier (Fig. 3.9). The newly introduced layers represent the logical separation and modularization entailing several advantageous implications. The Web tier serves as a layer of abstraction of the database tier. From an architectural point of view there may be multiple data stores working with the Web tier entities.

The separation of layers has serious implications for the interfaces between packages and the interaction, i.e. the lines connecting them. A tier may eventually be located on a separate computer (or even cluster of computers), meaning that the designer must account for different interaction technologies and their implications for the functionality as long as there is a relationship between WAA packages crossing the border of a tier.

In general three-tier architectures are a very good choice for Web applications with simple business logic and for applications based on dynamic content stored in a data store like a relational database. Three-tier architectures are also appropriate for applications undergoing rare presentation changes and supporting a low degree of personalization. But these architectures are a bad choice when a significant amount of scripting is

considered by designers or results from client requirements. Three-tier applications exhibit poor scalability, therefore they may not be the best choice for user- and data-intensive applications.

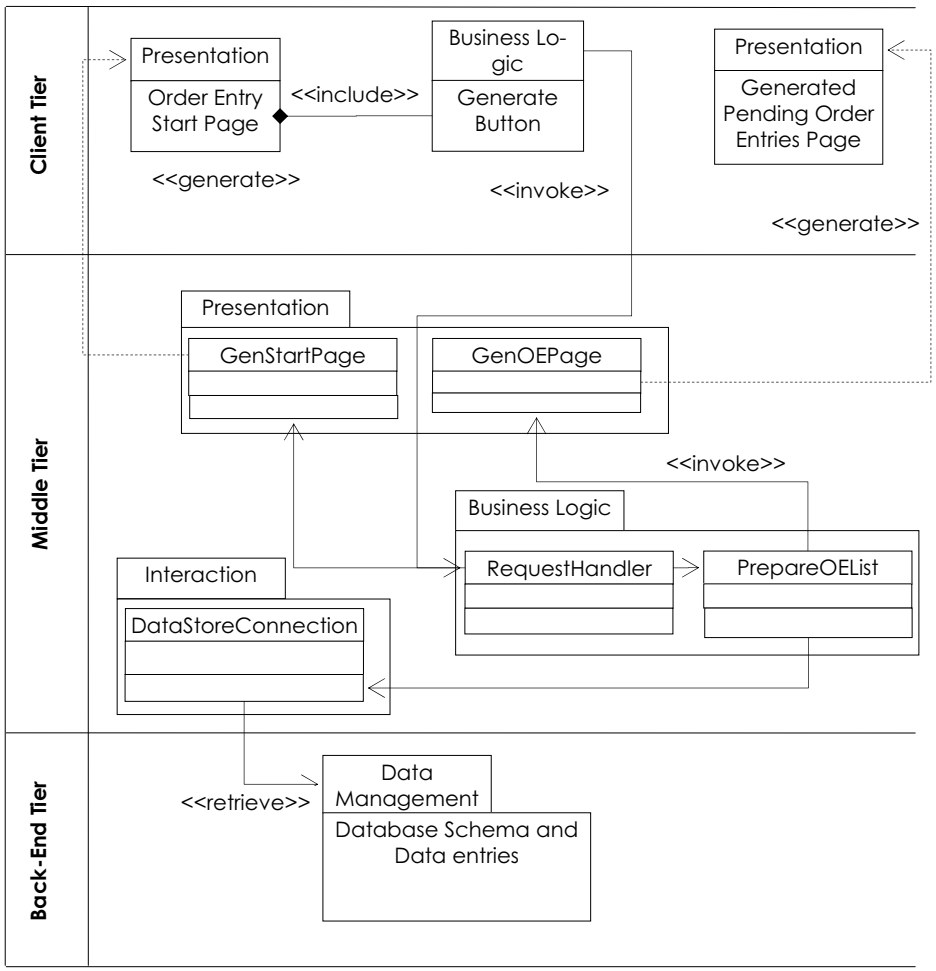


Fig. 3.9. Initial WPA

Having completed the transition to a three-tier architecture we are in a position to re-evaluate the requirements formulated in the preparation phase. By doing so we are about to complete the second iteration. Let us reconsider requirement 7. Scripting approaches (Chap. 5) contain a significant amount of presentation oriented logic which may invoke methods on the business logic. To reflect this a new set of classes belonging `Interaction` package must be introduced on the Web tier. By doing so the designer will eventually end up with two different groups of classes belonging to the `Interaction` package, which is a clear symptom for sublayering. Therefore the middle tier is split into two layers – the Web tier and the application server tier.

Of course there are other more technical factors leading in the direction of four-tier architecture. Scalability and load balancing are one group. Different execution environments for business logic and presentation represent a different group.

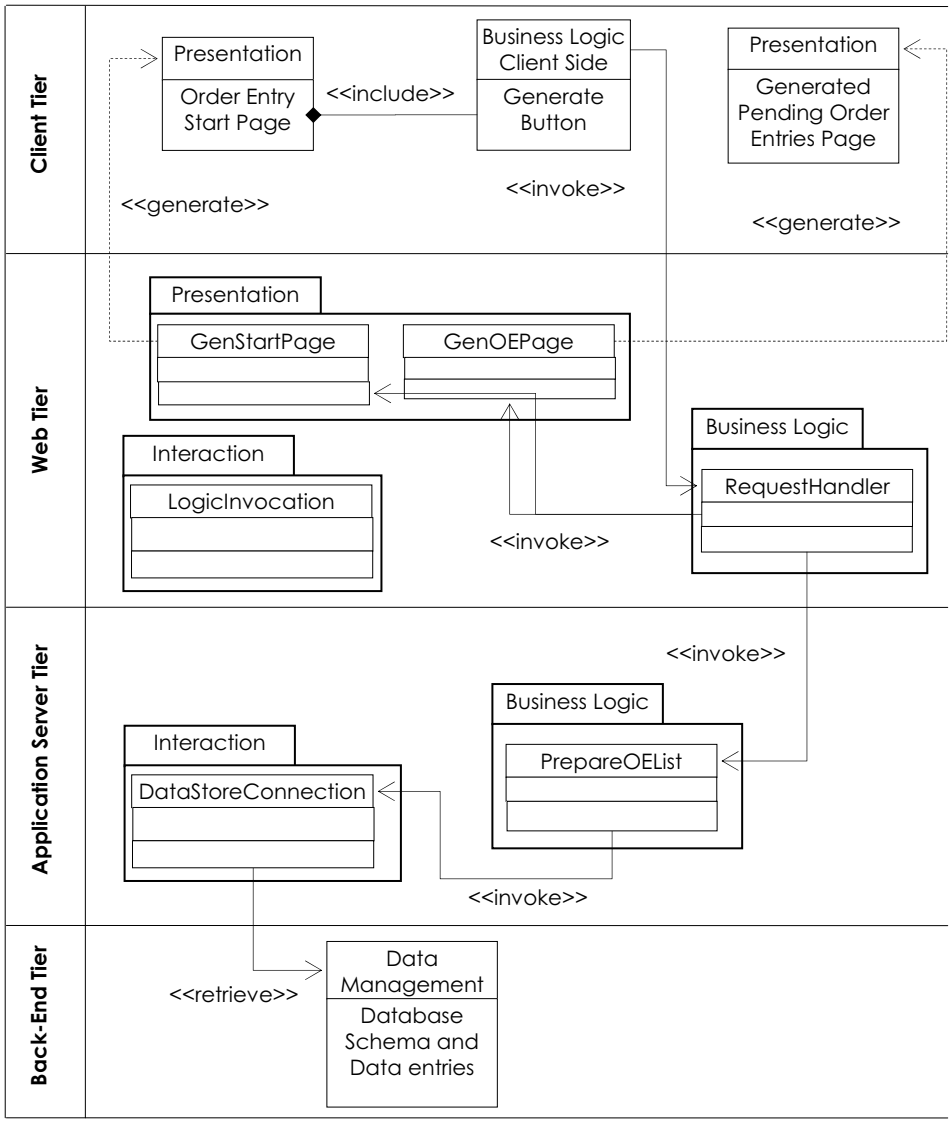


Fig. 3.10. Extended WPA

3.4.4 Four-Tier Platform Architecture

Figure 3.10 shows the four-tier architecture for the order entry example. It illustrates the clear logical division between presentation logic and business logic. Most of the presen-

tation-oriented functionality and the `RequestHandler` are located on the Web tier whereas the `PrepareOEList` business logic class is located on the application server tier.

Four-tier platform architectures offer a much better logical separation of concerns. The business logic is isolated in the application server tier. The Web tier represents a Web interface on top of the business logic, handling the presentation generation (`GenStartPage` and `GenOEPage`) and the input requests (`RequestHandler`). The benefits of such a separation become evident if the designer decides to introduce a Web service interface. The Web service specific infrastructure will be located on the Web layer, will be separate from the current classes, and will use the business logic directly. From the client's point of view, however, the Web application will have two completely different interfaces. One will be presentation based, the other Web service based – both operating on top of the same business logic.

There is another, more practically oriented set of advantages offered by the four-tier architecture. It is concerned with scalability. Different servers can handle the presentation generation and the input request processing. They can also be clustered. Additionally, a Web tier can work with different application server tiers.

In summary we would like to draw the conclusion that better and more precise platform modeling can be achieved by increasing the number of tiers. The technical properties of the system are improved as well. At the same time, however, the complexity of the architecture increases.

3.5 Design Phase: Assign Capabilities

The next logical step is to assign capabilities to the WAA packages located on the different tiers. By doing so we define what capabilities are used by what packages. Capabilities are defined in Sect. 2.4. They represent generic properties that the WPA platform modules must provide to the WAA components. In other words, capabilities are characteristics of the WPA layers, which the WAA packages can use.

Generally speaking, four different kinds can be distinguished: visualization, logic execution, communication and data store. Visualization stands for the display of HTML presentation files, providing graphical capabilities etc. Logic execution stands for the generic execution environment for different logic pieces. Data store stands for generic data persistence mechanisms – databases, serialization, files, etc. Communication describes generic communication capability, i.e. communication allows different WAA modules to exchange pieces of data.

Assigning capabilities is a first step towards specifying what kind of technologies will be required. By working at such an abstract level the designer can evaluate the pros and cons of different alternative solutions. Fig. 3.11 shows a slightly modified version of Fig. 3.10, where an additional field is introduced for each entity, containing the platform capabilities used (in italics).

The result of this phase is a WAA which is abstract enough and is still technology-independent, but concrete enough to be validated. The assignment of generic capabilities is an intermediate step hiding the details of technology selection and the choice of concrete platform modules. These will be discussed in the next chapter.

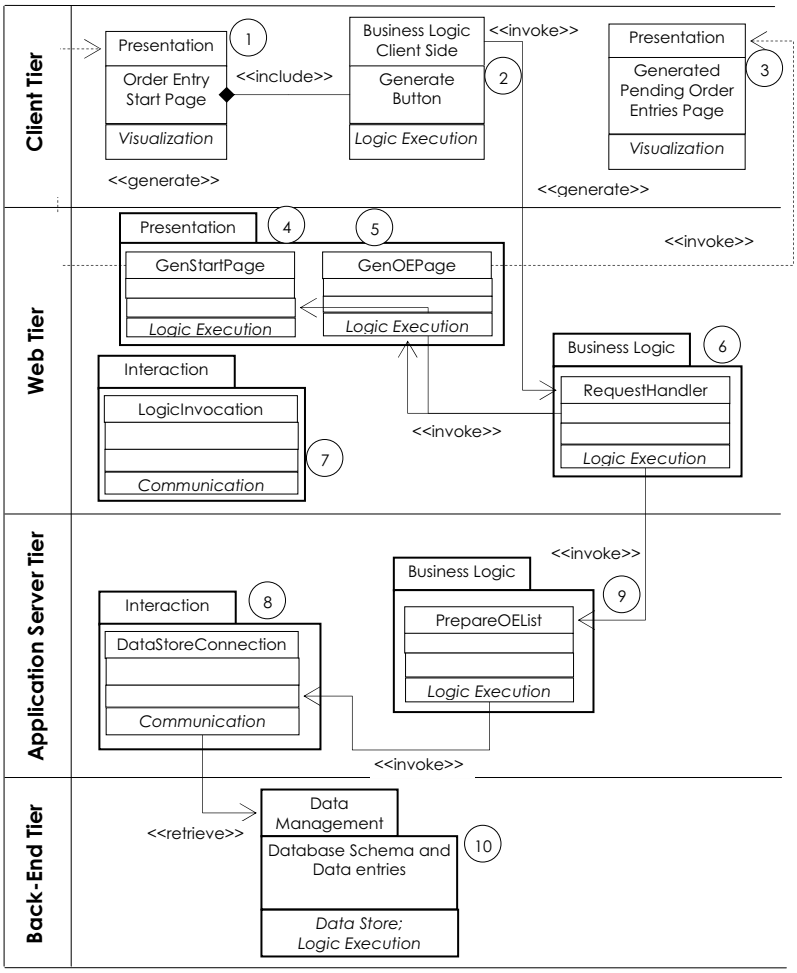


Fig. 3.11. WAA with assigned capabilities

3.6 Design Phase: Iterate and Improve

The next iterative improvement step involves two activities. Firstly, the platform architecture is split into more layers, which leads to an extension of the WPA. This means iteratively populating the layers with packages and splitting the layers, once a sublayer appears. Secondly, the WAA is extended by introducing new modules.

Our experience shows that it is relatively easy to lose track of the extensions or run out of new ideas. Therefore we recommend using matrices as in Fig. 3.12 showing the mapping of the WAA onto the WPA. We call the matrix cells (matrix elements) quadrants. Every quadrant contains a set of classes or whole packages located on the corresponding WPA tier and belonging to the respective WAA package.

	Presenta-tion	Business Logic	Interaction	Data Manage-ment	Personal-isation	Security	Descrip-tion	Import / Export Interface
Client Tier	1 3	2						
Web Tier	4 5	6	7					
Application Server Tier		8	9					
Back End Tier				10				

Fig. 3.12. Web application framework architecture matrix

There are three direct benefits designers can obtain by using the matrix. The first one is that designers can easily identify deficiencies in their existing designs. For example, it can be clearly seen that no security has been used and that there is no support of personalization. The first four by four quadrants (starting from the upper left corner) contain some entries. Designers may use this information to validate whether the proper WAA components are assigned to the right quadrants. The second benefit results from using the matrix as a source of new ideas. The presentation of the architecture and the ordering in quadrants with respect to WPA and WAA provide a basis for an analyses and architectural comparison. Designers can easily identify a lack of certain features by the missing representatives. For example, the fact that the column "Import/Export Interface" is empty may be perceived as a hint for an alternative application design – perhaps the use of Web services. Last but not least, the matrix can serve as a validation point. For example, consider whether to use stored procedures handling data-specific operations. These will represent business logic on the back-end tier. A direct advantage of using stored procedures is a significant improvement in application performance. The matrix in Fig. 3.12 is a basis for the Web application framework architecture. It will be extended with the introduction of technologies. Not only will the WAA classes and packages be assigned to technologies but also WPA capabilities will be assigned to platform modules. This step will be dealt with in the next chapter.

Testing is an important issue in the application lifecycle [Patt00]. Almost all traditional testing methods developed for "standard" applications are also applicable to Web applications.

3.7 Alternative Notations

The notation we have introduced for the WPA and WAA is a new one. The scientific community proposes different approaches for modeling Web applications. Thus, our notation should be compared with alternatives ones. In this section three approaches will be introduced, namely the pole shoe notation, the UML approach, and WebML.

3.7.1 The Pole Shoe Notation

The first approach we will consider is introduced in [NMMZ00] and is called "the pole shoe" notation. After an analysis of WAAs the authors outline a number of building blocks in the sense of architectural or modeling elements. They also illustrated how two-, three-, or four-tier architectures can be created from these building blocks. The following architectural elements are defined:

- System service – this is used to model an execution environment or a server for specific components. The Java Virtual Machine (JVM), an Internet browser, or an application server will be modeled as system services. In the pole shoe notation the platform is modeled mainly as system services on different layers or as protocols.
- Java applet – these are representatives of a whole class of technologies called client side logic (Chap. 5). Client side logic is used to implement pieces of functionality which are to be executed on the client computer.
- HTML pages – these are documents formatted according to the HTML standard. The presentation of the Web application is typically implemented as a number of linked HTML documents. Such an HTML document can contain client side scripting logic.
- Multimedia – to provide rich content Web applications may not just comprise a set of linked HTML but also refer to resources of different types such as sound or video clips. Such resources are modeled as multimedia elements.
- Program – this is used to model any kind of business or presentation logic. Components such as EJB or scripting logic such as PHP are modeled as programs.
- Protocol – this is used to model any kind of transport or communication protocol used to implement the interaction between different program elements or the communication between system services.

An example for a Web application modeled in pole shoe notation is shown in Fig. 3.13. It shows the order entry example architecture. The application is modeled in four tiers. Each tier includes a set of system services corresponding to the platform components located on a layer. The communication between the services on the different tiers is modeled as protocols. The Web tier is modeled in more detail to account for the scripting logic and the corresponding scripting environment, and for the invocation logic (the request handler) and its environment.

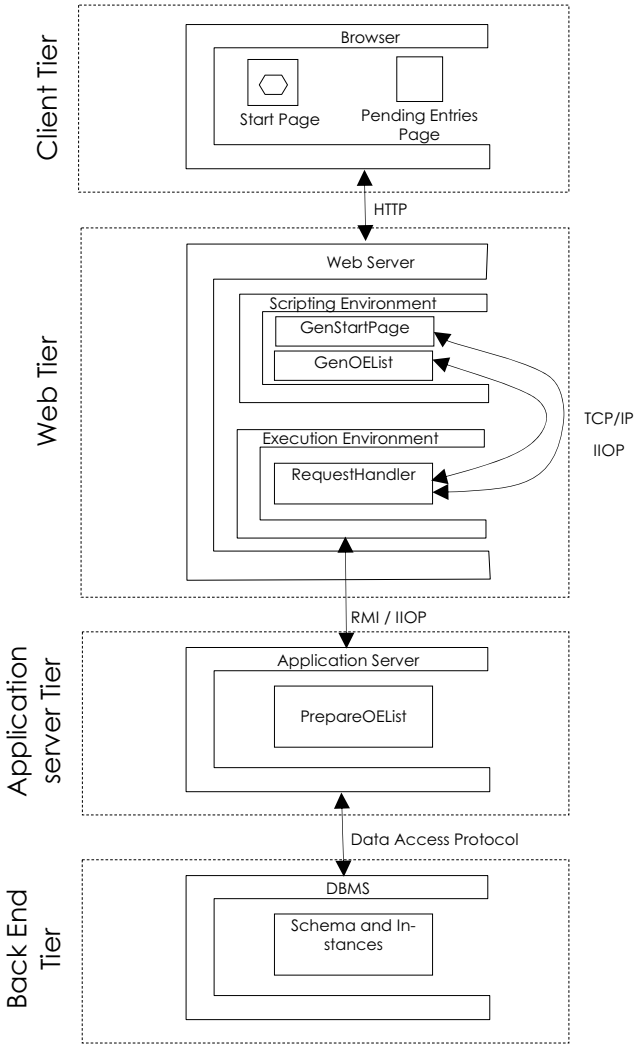


Fig. 3.13. Pole shoe notation order entry application architecture

To evaluate how the approach we propose and the pole shoe notation relate consider the following arguments. The pole shoe notation is very intuitive. It provides a comprehensive set of basic modeling primitives (Fig. 3.14). The developed architectures are easy to read and understand since they are mostly modeled at a relatively high level of abstraction. The authors also considered the idea of modeling platform architecture, presented in terms of layers and execution environments. However, this separation is not designed or as clearly pursued as in the approach we propose.

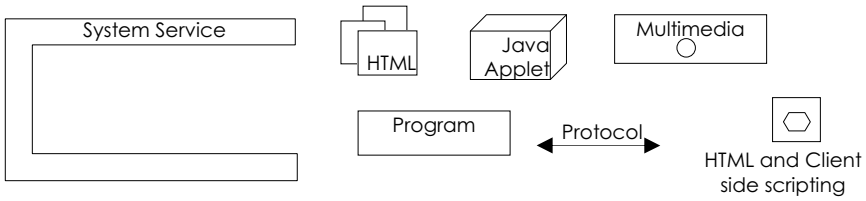


Fig. 3.14. Some of the pole shoe notation primitives

3.7.2 UML WAA Design

UML is a well-established modeling language, which is associated with application modeling methodology. The use of UML goes beyond the strict scope of object-oriented design. A proposal by Jim Conallen [Cona99] was made regarding the modeling of WAAs with UML.

The core concept is to use a set of predefined UML stereotypes to model different parts of a Web application. For example hyperlinks are modeled as associations from a certain stereotype. In the general UML approach [Cona99], [Cona02] there are stereotypes for all elements of the WAA and for some of the technologies. For example, there are stereotypes for Web pages with client side logic, Java applets, or ActiveX components for forms and JSP. All in all, there are 23 such stereotypes defined.

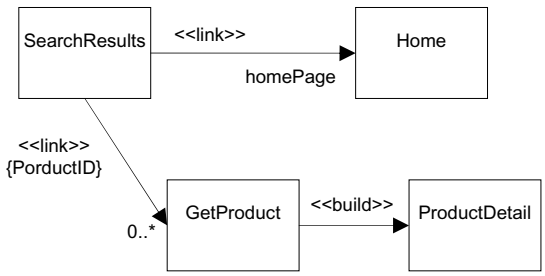


Fig. 3.15. Preparation phase in the UML approach

The UML Web application design also involves an iterative design approach. As a first step all interactions are modeled (Fig. 3.15). As a second step UML classes derived from the respective primitives are introduced to model the different parts of the application across the different tiers (Fig. 3.16).

Consider the following arguments when evaluating the technologies. UML is a widely accepted modeling language so designers using UML will not have to spend time and resources learning a new notation – rather they can reuse their knowledge.

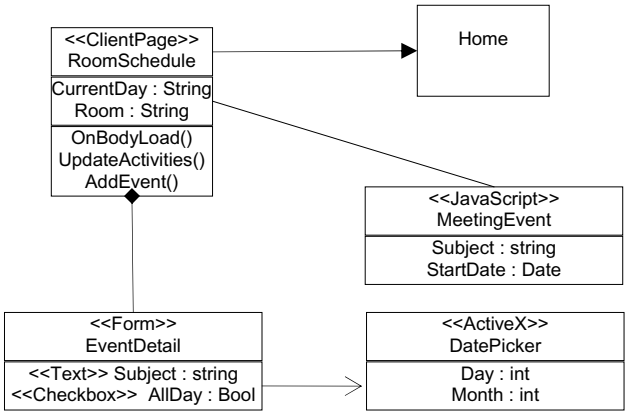


Fig. 3.16. Application architecture in the UML approach

In summary, one of the greatest strengths of the UML approach is that it uses modeling that is the industry's standard. In is implemented in many CASE tools (e.g. Rational Rose, XDE). And the UML approach is more or less intuitive. The modeling elements are a UML profile consisting of a set of stereotypes. One of the obvious weaknesses is that the concept of considering platform modules and layering is not present. The design methodology suffers from a lack of generality. Different technologies are hard-coded as stereotypes, which adversely affects the quality of the design process.

3.7.3 WebML

WebML [CFB+03] is a novel approach to designing Web applications. In essence it provides the methodology for model-driven Web application development. Compared to the approach discussed in this book, it deals predominantly with WAAs. Therefore it can be classified as a Web engineering approach. Web engineering is discussed in detail in Chap. 8.

The foundation of the WebML approach is the separation between content, structure, and presentation (Chap. 8). WebML distinguishes clearly between data design, hypertext design, and presentation design. To enable true model-driven development all these different "aspects" are combined during the different steps of the development process (design process, workflow). The design process serves to glue them together.

Data design deals with the development of data models for storing content. Data models are meant to be language and product independent. They are created using E/R as modeling methodology. Concrete data schemata are generated at a later stage for the specific products thus ensuring "platform independence". Typical data models consist of entity types having attributes. Entity types are connected with relationships – binary or *n*-ary. Only two types of relationships are supported: a generic relationship (association) and is-a (generalization specialization). Interestingly enough, aggregation (which is crucial for Web application contents) is not modeled explicitly. Rather it is considered as a special case of the generic association.

The hypertext design is based on three generic modeling constructs: units, which are generic information elements; links which are generic connectors; and pages, which are

composite objects comprising a number of units. There are five types of units: data units, multidata units, index units, scroller units, and entry unties. Any unit is associated with an entity type (from the data model). Every unit has a set of selector conditions allowing it to select a set of different entities (instances of the entity type) and also has input and output. The input consists of selector parameters required to compute the set of data entities the unit is associated with. The output can be used to compute other units, i.e. to implement chaining. Links represent directed connection between two units. Links always connect two units: a source and a target unit. They provide navigation or data transport from one unit to another, and can also be used to implement a business logic functionality call. Pages are generic containers of units which can be displayed, and are typically organized into hierarchies.

Presentation in this approach is implemented on top of XSL. It provides device and document format independence. Thus presentation can be generated in a rather flexible manner promoting personalization (personalization parameters and client identification are implemented as properties of sessions).

To recapitulate, WebML is a novel, general, and quite flexible approach to designing Web applications. It is content management and Web engineering oriented, as well as model driven. It relies on generation and aspect independence, which contributes to its richness and high applicability. Compared to the design methodology proposed here, WebML can be described as a WAA development methodology.

3.7.4 Model Driven Architecture

A new technology handling code generation and transformation is the OMG model driven architecture (MDA, [KlWB03]). MDA handles metamodel-guided transformations of models. For example, a WAA of an application which uses component-oriented programming can be designed in a component-technology neutral manner. Such a model is called platform independent model (PIM) in MDA terminology. It is independent of the specifics of any component technology such as EJB, COM+, or CCM. The PIM can be then transformed into a platform specific model (PSM) in a semi-automated sequence of developer-guided steps. The transformation rules and the transformation tools are an integral part of MDA. The PSM represents the PIM mapped onto a concrete technology, e.g. COM+. One or more PSMs are generated from a PIM by applying the transformation. It may also be expected that there will be relationships between the different PSMs. Since they span across the borders of technologies (a PSM corresponds to a technology) these relationships are called bridges [KlWB03]. The final MDA phase involves code generation – the skeleton of application code for the future application is generated from the PSM. Code generation is also termed transformation.

A direct comparison of MDA to our approach would show that MDA is very useful in mapping the WAA on concrete technologies (Chap. 4). As shown in Sects. 3.3 and 4.2 we first create a technology-independent WAA, which is later mapped on selected concrete technologies. MDA, however, does not handle the concept of platform modules. MDA does not allow assigning components to application modules. Therefore, MDA cannot help developers to define the dependencies among WAA components and how they are mapped on WPA modules.

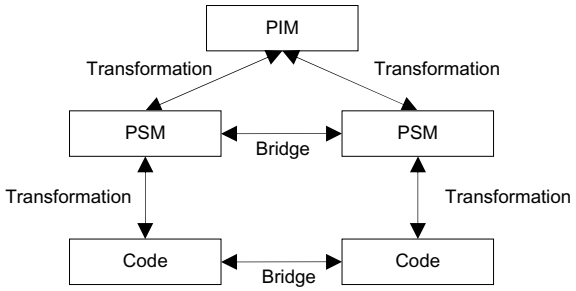


Fig. 3.17. Steps in the MDA development

In our Web application design approach we provide the possibility to handle application architectures and platform architectures. In addition, we start with technology-neutral design and then map the corresponding architectures on technologies. MDA is a very efficient design paradigm for the development of technology-neutral application architectures, handling all subsequent steps until code generation. In our approach we treat platform and application in an integrated manner, which provides significant advantages especially for Web applications.

In brief, the Web application design approach proposed in this book is a conceptual approach. It serves as a guideline. Therefore, it can be compared to traditional software development methodologies such as RUP or extreme programming (XP). MDA, alternatively, is an approach to model transformation, which can be used as part of our approach to perform the mapping between WAA and WPA.

3.8 Conclusions

In this chapter the step-by-step approach towards Web application development was introduced and illustrated with a practical example. The process of developing Web applications was characterized as strictly iterative. It was shown how to go about designing the WAA based on a set of characteristics resulting from a requirements engineering phase. Next the mapping of the WAA onto the WPA was demonstrated. Throughout the process criteria were presented on how to split different tiers and the characteristics of the two-, three- and four-tier architectures. Last but not least, the Web application framework architecture matrix was discussed.

A question which remained more or less open throughout the last two chapters is whether any large application is a Web application. In this book we try to distinguish between two notions. All applications which have a Web interface are considered Web applications. All other applications are not considered here, they are not Web applications.

4 Classification of Internet Standards and Technologies

Technologies are the third dimension in the Web application framework architecture introduced in Sect. 2.2. The need to consider technologies throughout the process of Web application design is motivated in Sect. 2.2.3. The architecture developed as a result of the stepwise approach described in the previous chapter is not tied to any concrete technology and is free of the specifics of any module of the framework architecture. Now we have to connect the neutral architectures for the Web application and the platform to concrete technologies in order to determine how to implement them.

In this chapter we introduce an exemplary classification and discuss some principles as to how it is constructed and how it can be changed (Sect. 4.1). In this we continue to develop the stepwise approach begun in the previous chapter by shaping the technology-free architecture through assigning technologies to the WAA components and WPA modules in each matrix quadrant.

Almost any of the Web application design approaches, some of which were presented in the previous chapter, have some kind of classification. Some of these approaches formulate it explicitly (our approach, the UML approach), others assume it implicitly (pole shoe notation). The classification introduced in this chapter is no exception to this rule, when it comes to deriving a number of concepts such as logic or presentation, which will be further used to characterize the WAA modules. What makes the role of the classification unique is that it boils down to concrete technologies. If designers go about assigning these technologies smartly, they will easily discover that the classification gives them a tool for the comparative evaluation of different application designs.

It is important to point out that the proposed classification is simply one of many alternatives. Myriad technologies and standards that can and are used in the process of developing Web applications exist. It is an illusion to expect that a thorough classification can be prepared and agreed upon broadly. The readers are encouraged to view our "proposal" critically, to develop it further, and to extend it by tailoring it to their individual needs. The lack of absolute precision of our classification does not impair the design approach. Without doubt, it fulfills our overall goal to provide some structure in the jungle of Internet standards and technologies as depicted in Fig. 2.1.

4.1 Classification

The proper choice of the root concepts is crucial to any classification. The packages of the WAA (Sect. 2.5) are chosen as classification roots (Fig. 4.1). This is a choice which is empirically made and is difficult to motivate. The authors' practical experience shows that WAA packages are a plausible choice which can be successfully used when building Web applications.

The classification involves multiple trees starting from the root packages. Either abstract categories or category bags (CBs) are used to classify further. An abstract category is (consider for example category 1.3 in Fig. 4.3) a category to which no technologies are directly assigned; rather they contain further category bags. Category bags contain different alternative technologies or standards (consider for example CB 1.3.1 in Fig. 4.3). The goal pursued with this taxonomy is not to classify all outstanding standards and tech-

nologies. It is to have a classification which is detailed enough and extensible enough to help designers find alternative and appropriate solutions.

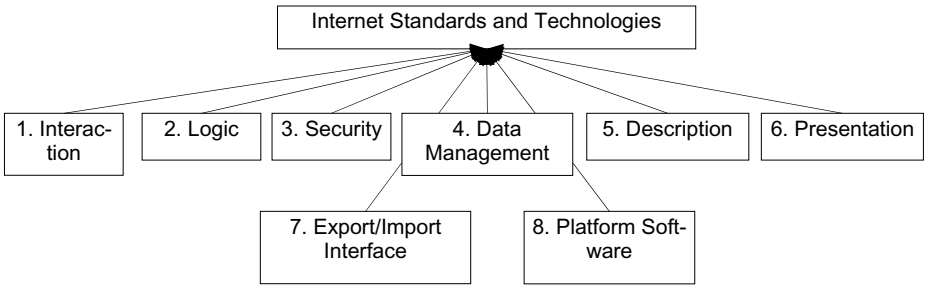


Fig. 4.1. Root classification entities

It is rather difficult to assign a technology or a standard to a single category bag. The reason for this is partly due to the fact that modern technologies and standards aim at integration and therefore cover multiple areas. A typical example for such a technology is ASP.NET – it represents a scripting technology and a Web invocation technology. Therefore the reader can expect that a single Web standard or technology is classified under multiple categories. Yet this needs to be minimized because it reduces the quality of the classification.

The fact that diverse technologies having some functional differences are classified under the same category is an issue. CB 3.1 (Fig. 4.5) is a typical example. Reflecting the primary objective for a technology is the guiding classification principle. Therefore it may happen that technologies having somewhat different orientation but still belonging to one group are put in one category bag.

Last but not least, finding the difference between a product, a technology, or a standard is a tricky issue. There are some technologies which are *de facto* standards (TCP/IP vs. ISO/OSI) and there are products which are based on a proprietary technology also having a rank of *de facto* standards (e.g. PGP). When performing classification or extending the one proposed it is important to concentrate on the general applicability and on the group to which the classified standard or technologies belongs. Do not discard products from further consideration when they represent a *de facto* standard, especially when classifying platform software (Sect. 4.1.8), but consider this choice carefully.

Categories	Example standards and technologies	Chapter
1. Interaction	ODBC, JDBC, SOAP, TCP/IP	2, 5, 7
2. Logic	Java Servlets, CORBA, EJB	5, 6
3. Security	XML Encrypt, LDAP	7, 10, 11
4. Data	HTML, XML, JAR	2, 6, 7, 8
5. Semantics	RDF, OWL	8
6. Presentation	HTML, XSL, CSS	5, 8
7. Export/Import Interface	WSDL, IDL	6, 7
8. Platform Software	Application and Web Server, database	2, 5, 6, 7, 8

Fig. 4.2. Discussion of the Internet standards and technologies

In this chapter we refer to many Internet standards and technologies. However, we avoid providing references to these since all of them are discussed in the second part of this book. Therefore, Fig. 4.2 indicates those chapters of this book where all the subsequently mentioned internet standards and technologies are explained. The reader is asked to refer to these chapters in order to get more information about them.

4.1.1 Interaction

The first group of technologies which will be considered is interaction (Fig. 4.3). In our view interaction is the application-specific part of the communication, i.e. the sequence of exchanged messages, the transmitted data structures, and so on and so forth. As mentioned in Sect. 2.5 it must distinguish between interaction and network communication. The latter is regarded as a capability of the platform.

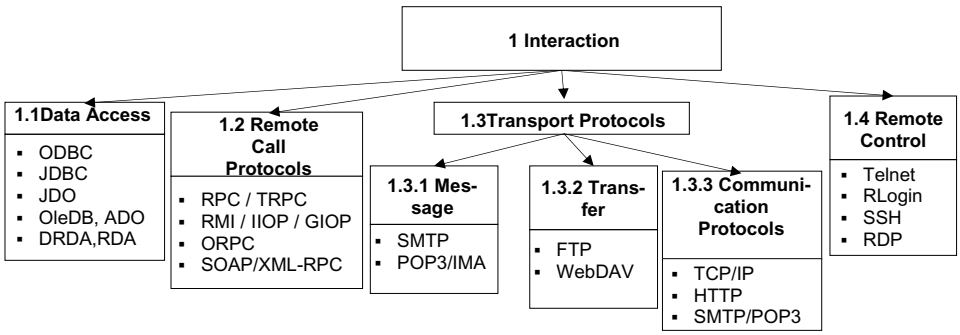


Fig. 4.3. Classification of interaction-related standards and technologies

We consider four subcategories, but are not limited just to them. For example, a new CB 1.5 could be added to account for the real-time protocols, if the designers decide to use any.

The first group of technologies is “Data Access” (CB 1.1). The primary focus of these technologies is to provide access to data collections of various types (predominantly databases). Despite the similarities, which is the reason for putting them in one category bag, there are also some genuine differences. ODBC and JDBC are competing technologies providing access to relational and object relational databases. OleDB and ADO (also ADO.NET) are database access technologies in the Microsoft realm based on top of ODBC. JDO (Java Data Objects) is a flexible object-oriented persistency mechanism. RDA (Remote Database Access, [ISO87]) and DRDA (Distributed Relational Database Architecture) [IBM04] [NeGr91] are technologies (standards) for remote database access. The support of database languages is not discussed explicitly. Most of the technologies are SQL based. Others, like JDO, support their own query language but are based on the ODMG standard.

The second category bag is CB 1.2 “Remote Call Protocols”. It contains a collection of protocols and technologies for remote procedure call-based invocations. These are mainly used in enterprise computing (middleware, component orientation). DCE RPC (Remote Procedure Call) is the most famous representative of this group. TRPC is a variant of RPC specially designed to provide transactional support to remote procedure calls.

IIOP and GIOP are CORBA-specific communication protocols. RMI (Remote Method Invocation) is Java-specific technology which is based on IIOP. ORPC is the equivalent technology for DCOM. SOAP (and its predecessor XML-RPC) is a lightweight protocol handling the communication in the realm of Web services.

The third category of interaction technologies is the abstract category 1.3 “Transport Protocols”. It is subdivided into three subcategories, namely message protocols, transfer protocols, and communication protocols. The category message protocols contain basically e-mail-related technologies such as SMTP (Simple Message Transport Protocol) used to send e-mail messages and POP3 (Post Office Protocol Version 3) and IMAP (Internet Message Access Protocol) to retrieve and organize e-mail messages. Interestingly enough, SMTP can also be used in the context of Web services as a communication protocol – therefore it also appears in CB 1.3.3. Protocols such as FTP (File Transfer Protocol) and WebDAV, which can be used to transfer files from one computer to another, are classified in CB 1.3.2 “Transfer Protocols”. There are probably many protocols which can be labeled as “communication protocols”, i.e. they can be used for application-specific interaction. Some of these protocols are TCP/IP, HTTP, the couple SMTP/POP3, and many others.

Last but not least, various protocols for remote control and operation can be classified under CB 1.4 “Remote Controls”. Technologies such as Telnet and SSH allow users to log on remotely, execute commands, and control the remote machine as if they were logged on locally. Telnet and Rlogin are nowadays succeeded by SSH (Secure SHell).

4.1.2 Logic

Logic (Fig. 4.4) forms the second category which will be considered. This category contains logic-related standards and technologies which are not directly related to the business application logic, which is the main reason for naming the category simply “Logic”. There are at least four subcategories: Scripting Languages, Business Logic, System Specific Logic, and Web Invocation Mechanisms.

CB 2.1 “Scripting Languages” is divided into two subcategory bags, CB 2.1.1 “Server Side” and CB 2.1.2 “Client Side”. CB 2.2 “Business Logic” classifies business logic-related standards into two category bags, CB 2.2.1 “Server Side” and CB 2.2.2 “Client Side”. As can be easily seen, the server side logic approaches are dominated by component-oriented technologies (COM/DCOM, EJB, etc.). The quite broad notion of J2EE has been written in the category bag. As discussed in Chap. 6 this implies that not only the EJB component technology but also the J2EE Web components can be used to implement a certain amount of business logic.

The next category bag is CB 2.3 “System Specific Logic“. The reason for calling this group system specific logic is that the application uses some of the capabilities provided by the system to program logic pieces. Database stored procedures are a very illustrative example. They represent a part of the application business logic associated with exclusively data-related operations programmed in a programming language supported by the database and stored and executed in it. If a data storage system different from a database were to be chosen then stored procedures would not have been available as a technological possibility and thus the designers would have had to think of an alternative solution. Further technologies are Web server or browser plug-in technologies or even executables or scripts implementing CGI, for example.

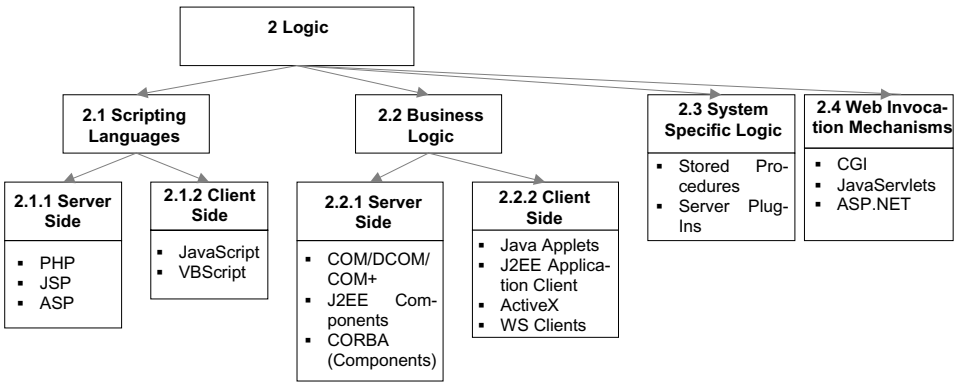


Fig. 4.4. Logic related standards and technologies

The last category bag CB 2.4 is "Web Invocation Mechanisms". It contains technologies such as CGI or Java servlets which are typically used to trigger the invocation of a method on business logic from an HTTP request. In the case of Java servlets this is not quite true. They can contain big and rather complex chunks of application logic. Still this does not diminish the fact that servlets can be used to trigger transactional and secure business logic method invocation.

4.1.3 Security

Security is the third classification category (Fig. 4.5). Although there are many alternative ways here it is preferred to classify the security-related standards and technologies into three different category bags: CB 3.1 "Message Security", CB 3.2 "Authentication", and CB 3.3 "Communication Security".

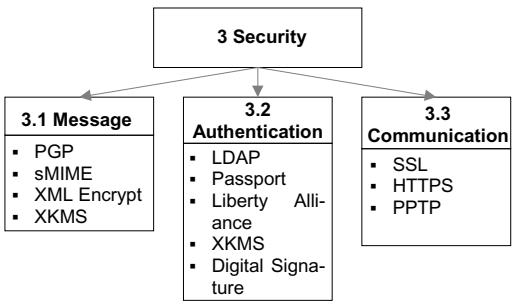


Fig. 4.5. Security-related Internet standards and technologies

Message security (CB 3.1) is a field of very active research, involving numerous competing technologies. As the name implies, the technologies gathered in this category serve the purpose of encoding just the contents of the exchanged documents or messages

in order to provide secure communication by leaving all other parties involved in the communication process unaffected. Some of these are PGP (Pretty Good Privacy), secure MIME (sMIME), XML Encrypt, and XKMS (XML Key Management Specification).

CB 3.2 contains technologies providing authentication services. These technologies are used as a means to identify and authenticate various users with the system and as a next step assign them the proper access control privileges and apply user preferences. Some of the technologies belonging to CB 3.2 are LDAP, XKMS, Microsoft Passport, and Liberty Alliance. Digital signatures are used to guarantee the authenticity of the digitally signed documents or messages, i.e. to make sure that the documents or messages indeed originate from the organizations or individuals claiming to be their authors.

Last but not least, several communication-level security-related technologies are classified under CB 3.3 "Communication Security". These technologies are providing encrypted communication as a means for secure data transfer. In contrast to the technologies in CB 3.1, these provide encrypted communication channels leaving the messages unchanged. Such technologies are for example Secure Socket Layer (SSL), its derivative HTTPS, and PPTP (Point to Point Tunneling Protocol).

Web applications (except for intranet applications) are much more "exposed" to threats than regular applications running as part of an IT system within an enterprise. Apart from all the security technologies shortly presented here, there is a lot of work which has to be done by the platform software. For example, regardless of whether a Web application uses HTTPS to transfer critical data, system engineers need to make secure the configuration of the Web server. For example, the proper rights on the file system at OS level must be set. This process goes all the way down to the hardware architecture. For example, the routers must be properly configured.

4.1.4 Data

The fourth category (Fig. 4.6) contains the classification of some of the data-related standards and technologies. In this classification category we concentrate predominantly on files and file formats. Another major kind of data is the message format and the message data. It is left out here in favor of the interaction category where they actually need to be discussed. Another relevant issue is document versus message formats.

The reader will see a mixture of them in most of the category bags discussed in this section. XML is an interesting example in this respect – its primary goal is to be used as document format; there are, however, progressively more protocols formatting the messages in XML. To reduce complexity and increase readability the category bags will not be further subdivided in document and message categories.

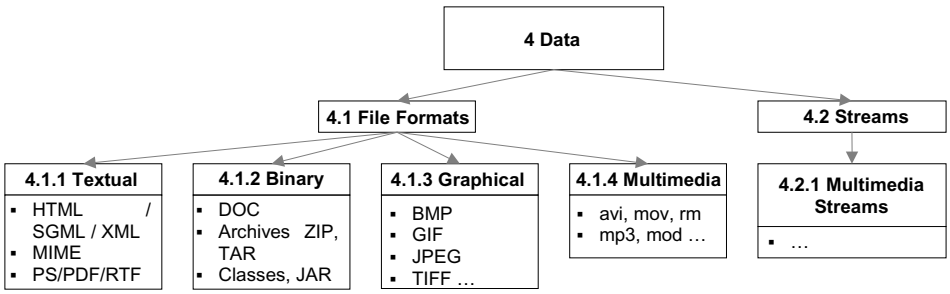


Fig. 4.6. Data-related standards and technologies

There are two subcategories of classification category “Data”, namely CB 4.1 “File Formats” and CB 4.2 “Streams”. CB 4.1 (File Formats) is subdivided into four category bags: CB 4.1.1 “Textual”, CB 4.1.2 “Binary”, CB 4.1.3 “Graphical”, and CB 4.1.4 “Multimedia”. There is such a wide variety of file formats on the Web that it is almost impossible to classify all of them in the proper category bag. Therefore it is attempted to classify just the most characteristic representatives in each category bag.

CB 4.1.1 contains standards and *de facto* file formats for textual documents and messages. File formats like HTML, SGML, or XML are markup-based document formats, which are also standardized. MIME (Multi-Purpose Internet Message Extension) is a standard widely utilized to format e-mail messages. Its use, however, is not limited just to e-mail messaging. Portable document formats are another issue to be reflected in the current category bag. File formats such as Adobe PDF (Portable Document Format), Adobe PostScript, and Microsoft RTF (Rich Text Format) are more or less *de facto* standards for documents created with only the goal of portability, i.e. operating system and device independence. For the most part these document formats are text based. There are, however, versions of these standards which are binary (e.g. linearized PDF).

CB 4.1.2 gives examples of binary file formats available on the Web. There are some examples of binary and proprietary document file formats such as Microsoft Office documents (based on the Compound Object Model) or Sun Open Office document formats. Additionally there are different archive formats such as ZIP, RAR, TAR, etc., which must be considered and also some executable files like for example Java archives and byte code .class files.

CB 4.1.3 contains some of the graphics formats available on the Internet. Formats such as GIF or JPEG or Bitmap or TIFF are standards on the Web.

Last but not least, some multimedia file formats are classified in this category. We chose not to develop the classification further and classify the standards into audio and audio/video. Certainly formats such as MPEG2, MPEG4, MPEG7, and file types such as AVI, MOV, or RM are part of this category bag. Some of the audio formats include .mp3, .mod, and many others.

4.1.5 Semantics

The semantics category contains standards for descriptive metadata (Fig. 4.7). It is subdivided into two subcategories: CB 5.1 “Web” and CB 5.2 “Multimedia”. The goal of using semantics-related technologies is to provide more and high-quality semantic descriptions, which can be used in searching and querying, composition, automated processing, automated reasoning, and many other fields. One of the major problems the Web faces today is the fact that there is quite a lot of information published. It is available in the form of structured or semi-structured documents. Unfortunately it is not “schematized” – that is, there are no schemata determining the type of a piece of information content. This is why the contents are mostly untyped – for example, the address of a person is simply available as text and is not of type address.

In the context of searching the missing schema leads to the fact that only text-based searches may be performed, i.e. string matching. This type of search leads normally to low-quality results. An attempt to solve this problem is made by introducing semantic descriptions. By doing so the contents are schematized and descriptive metadata attributes are assigned. Both of them are considered when searching, which improves significantly not only the search results but also the automated processing.

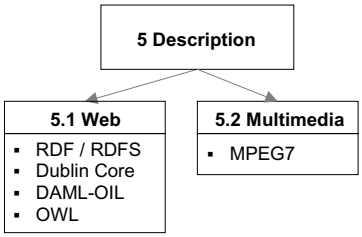


Fig. 4.7. Semantics-related Internet standards and technologies

CB 5.1 contains mainly ontology languages used to code semantic descriptions. Semantics and descriptive metadata play an important role not only on the Web but also in the field of multimedia where descriptions are becoming increasingly widespread with MPEG7. Such standards are used to describe structural parts of movie scenes such as characteristics of the persons on it or characteristics of the background. Organizing multimedia metadata requires extensive classification and standardization efforts.

4.1.6 Presentation

CB 6 contains standards and technologies used for presentation purposes (Fig. 4.8). In other words, these are technologies which are used to encode presentation data, which is then rendered (mostly graphically) by the client side platform.

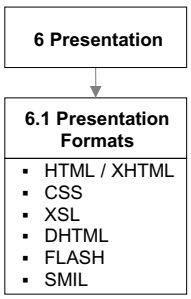


Fig. 4.8. Presentation-Related Internet standards and technologies

CSS (Cascading Style Sheets) guarantees a consistent layout of the presentation content regardless of the device or browser differences. Using XSL (XML Style Sheets) different transformations can be applied which eventually can be used to apply different presentation styles on the content. DHTML (Dynamic HTML) is a technology combining client side logic (VBScript or Java Script) with HTML. Flash and SMIL are proprietary multimedia extensions with which designers create high-quality animated multimedia Web sites.

4.1.7 Export/Import Interface

CB 7 contains standards for interface descriptions (Fig. 4.9). These can be used to define some parts of the application's business logic as the export interface. A description of the

interface made in one of these languages can be used for different purposes. The two most important are: to register the interface in a registry for discovery at later stage and to build subs and skeletons. The former determines, among other thing, why the term import interface is used. The application is actually imported by importing the construct derived from the interface description.

This idea is not genuinely new – the original term was API; then the idea gained significant importance with component-oriented programming. Web services are a technology which can be used to export the direct business logic interface in the context of Web applications.

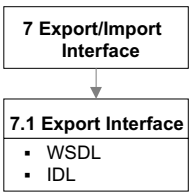


Fig. 4.9. Export/Import interface standards

IDL (Interface Definition Language) is a *de facto* standard for interface description originating from DCE RPC. It is used (with some changes and extensions) in CORBA and DCOM. WSDL (Web Service Description Language) is an interface description language for Web services, described in more detail in Chap. 7.

4.1.8 Platform Software

A classification of platform modules will be presented now (Fig. 4.10). Platform software is independent of the Web application architecture; however, it is preferred to discuss the platform software classification here because it logically belongs to the classification section. The different modules appearing in this classification will be used further in the platform design.

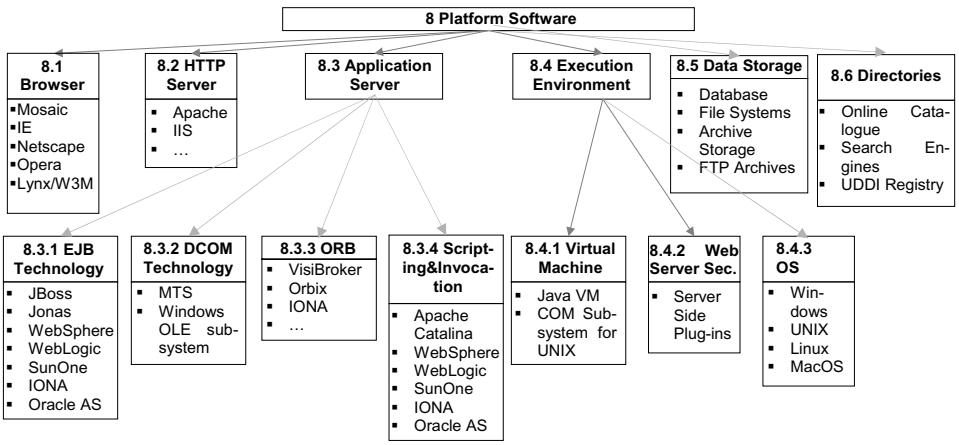


Fig. 4.10. Classification of some platform modules

There are several classification categories covering various platform-related technologies. This classification does not pretend to be exhaustive: there are platform modules which were intentionally left out; others require a higher level of detail.

Two interesting classification categories are CB 8.3 “Application Server and 8.4 “Execution Environment”. They are considered separately to underline the fact that there is a difference between execution environment for business logic components and execution environment for other WPA modules. Let us assume that large parts of the business logic of a Web application are written in EJB. The EJB components are deployed and executed in their container which is the EJB Server. The EJB server itself may be written in Java (e.g. JBoss) and must be executed as a “normal” Java application within the JVM. In this case the JVM is an execution environment for the EJB Server. We can apply the same reasoning with respect to the JVM and the OS. The JVM executes as a “normal” application on an OS and so on and so forth.

The application server category treats the notion of application server in the sense of component container. Three major branches can be distinguished: EJB component container, DCOM and COM+ component container, and CORBA ORB. Several implementations are available for these technologies some of which are listed in CB 8.3.1 through 8.3.3.

There are also several kinds of execution environments: virtual machines, Web servers, and operating systems (CB 8.4). Web servers are considered to act as the execution environment because some of them offer extensibility APIs for writing plug-ins. The server side plug-in executes in the environment of the Web server which controls their lifecycle, and offers memory management and resource control functions. The scientific community does not have a single position on whether or not this group must be classified under the application server group. Virtual machines such as the Java Virtual Machine are another kind of execution environment providing for portability.

4.2 Developing WAA and WPA – Continued

Having created a classification of technologies the designers are in a position to continue developing the architecture of the Web application in a stepwise manner. It was initiated in Sect. 2.2 and ended with the architecture of the Web application and the Web platform. At this stage the designers must make the decision of assigning technologies with which the architectural entities of the WAA will be implemented (Sect. 4.2.1). This decision can be based on the alternatives the classification offers and in a way reflecting as many as possible of the client requirements (Sect. 3.2); these requirements are meant when requirements are referenced throughout this section. The next step is to assign platform components to each module of the WPA (Sect. 4.2.2). To do so the designers consider the classification of platform modules.

4.2.1 Mapping the Technologies

Figure 4.11 is an extension of Fig. 3.11 and Fig. 3.12 showing the WAA components introduced in the matrix. To simplify the figure the columns in the matrix containing no entries were ignored. In order to put the technologies in the graphical representation, a gray box on top of each class or entity containing the technology in curly brackets is used.

The Generate button must be implemented using client side logic. JavaScript is a very good technological choice since it is natively supported by the large majority of Internet

browsers. Alternative technology from the same category bag is VBScript, however, it is browser and OS dependent, which contradicts requirement 1. Designers can also consider using a small Java applet as representative of CB 2.2.2. It is less likely to be chosen because it is not consistent with the thin-client ideology implied by requirement 2.

As discussed in Sect. 3.2, requirement 5, scripting must be used, which leads us to CB 2.1.1. On the other hand, we can assume that an implementation aligned with Java technologies is assumed due to the interoperability requirement. Therefore JSP can be used to implement `GenStartPage` and `GenOEPage` (Fig. 4.11). Since the only criterion was interoperability designers can also consider PHP, though a native Java technology implementation will lead to a coherent application. For similar reasons the choice of implementing the class `RequestHandler` in a Java servlet is made.

Requirement 5 calls for using a component technology to implement the major part of the business logic. The choice of EJB as the component technology to implement the prepare list is predefined. Last but not least, the choice of JDBC (CB 1.1, Fig. 4.3) appears logical as well.

4.2.2 Choosing Platform Software Modules

Having assigned different technologies the designers are now in a position to select platform software modules for each quadrant. To do so, designers may consider the classification of platform modules (Fig. 4.10).

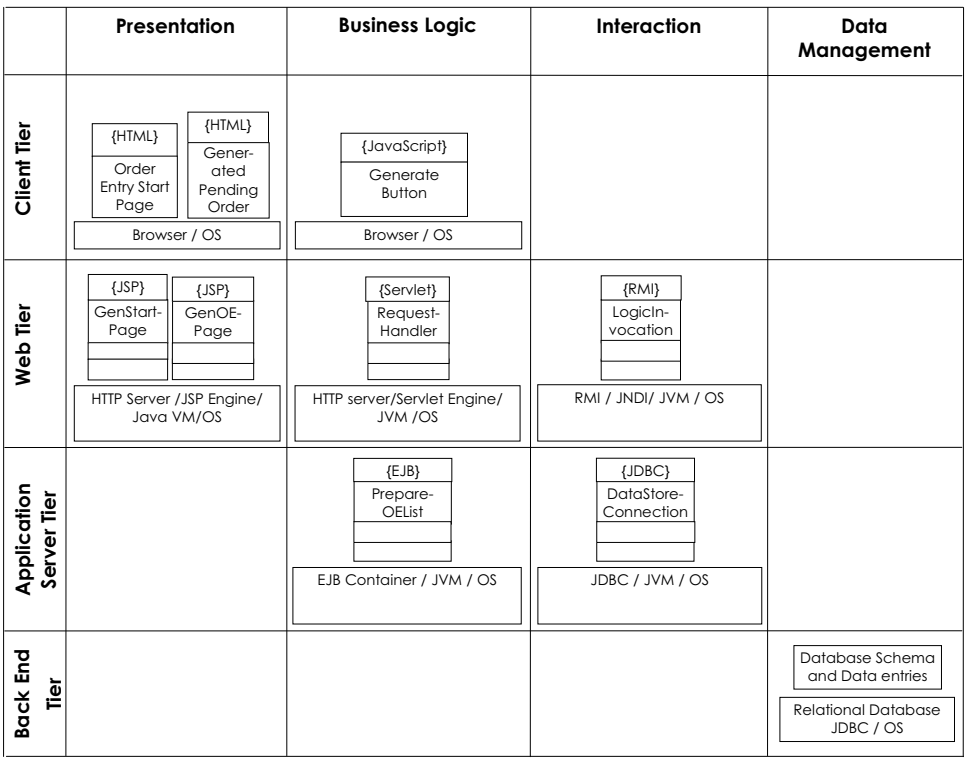


Fig. 4.11. Architecture of Web applications – continued

Any of the conventional Internet browsers (CB 8.1) will be in a position to effectively host all client tier entities. The thin-client architecture in the order entry example yields a minimal set of requirements. Therefore the client side platform is a simple one.

The platform of the Web requires an HTTP server by default. In order to account for the servlet and the JSP technologies (J2EE Web Components) the designers need to choose the appropriate engine. There is a single engine called J2EE Web components container for these two technologies (consider CB 8.3.4). All platform modules for Java technologies require the existence of a JVM (consider CB 8.4.1); therefore it must also be considered as platform software module. Additionally the designers need to choose an OS. The choice of an OS is arbitrary (no client requirement regarding the OS exists). Furthermore the JVM itself assures interoperability of the rest of the platform software.

No special platform software is required for RMI – all the needed technologies (JNDI, RMI) are included in the JVM.

An EJB container is necessary to implement the EJB business logic (consider CB 8.3.1). The designers need to make a strategic decision – whether to use an open source implementation or rather a commercial integrated application server product. For the purposes of the order entry application an open source implementation will deliver sufficient performance and reliability; therefore JBoss [JBOS04] or JONAS [JONA04] may be selected as EJB containers [SDK04].

Now that the matrix in Fig. 4.11 is constructed the designers are in a position to take a bird's eye view of the architecture and reevaluate it iteratively. By doing so the designers can reevaluate:

- The choice and the distribution of platform modules. Consider for example the Web tier – it is evident that almost all the required technologies are available in most of the software packages (commercial or open source) available today. Based on this architecture, however, the designers can pinpoint the precise use and discover and remove inconsistencies.
- The proper design of WAA packages and classes and their distribution over WPA tiers – the designers can review once again, considering the technology and platform software mapping and whether the chosen distribution is the proper one. It may well be the case that some new WAA classes come into play. Consider for example a future personalization, which will require storing the user preferences and settings on the client side platform. If they become too extensive, HTTP cookies will no longer be an appropriate solution. Hence WAA must be changed.
- New improvements – the quadrant <application server, presentation> is empty. A server side logging is precisely the right candidate to be positioned there. Having reached that conclusion the designers can implement a logger module to implement this functionality (which can later be used for auditing).

At this point we have reached a stage of the design which allows us to implement a first prototype or first operational version of the order entry application. As with all other design methodologies, our approach is also finally based on iterations and refinements.

Having defined the WAA and having chosen the WPA, developers can start generating tests for certain WAA components. If they are completely implemented and operational certain unit tests may be created automatically and synthetic data and function calls may be generated to empirically prove the proper functioning of certain WAA components.

Part II:
Internet Standards and Technologies

The second part of this book is dedicated to the Internet standards and technologies introduced in the classification of Chap. 4. We describe a set of basic standards and technologies common to most Web applications (Chap. 5). These so-called "conventional technologies" include technologies for the generation of HTML and for invoking application logic such as CGI and Java servlets.

There is a significant overlap between enterprise computing technologies and Web applications. Component-oriented approaches find extensive use in Web applications to implement their business logic. These technologies can be applied on the Web tier, the application server tier, and the back-end system tier. For this reason we discuss different middleware and component-oriented technology in Chap. 6. Last but not least, we discuss technologies for Web services (Chap. 7) and Web content management (Chap. 8). Web services are a new technology for the integration of Web applications. Web services provide for a general and standardized infrastructure that facilitates the integration of business functionality. Web content management is an approach to create presentation and data-based Web applications. Especially, its association with Semantic Web concepts makes it most relevant for future Web applications.

At the end of this chapter the reader will not only know the conceptual differences between the different programming techniques for Web applications, but also be able to distinguish different application areas and to decide which technologies are appropriate in which case.

5 Basic Programming Concepts for Web Applications

This part of the book introduces programming concepts for Web applications. We distinguish two different sorts of approaches, basic ones and more complex ones. In this chapter, we will talk about the basic concepts, whereas Chap. 6 will be dedicated to higher level approaches. The big advantage of the solutions presented in this chapter is the ease of cost at which they come. They do not require extensive environments or special infrastructures, but can be realized comparably simply. On the other hand, they only offer basic means for application programming. They cannot compete with high-level approaches when it comes to complex services like transaction management or sophisticated access control mechanisms. Irrespective of their simplicity, some of them can be considered to be as powerful as conventional programming languages. Thus, even complex scenarios might be realized using these simple approaches, avoiding the complex overhead required for higher level technologies.

5.1 Overview

At the dawn of the Web there was no dynamic content. The Web was just a collection of static HTML pages. These files were located on file systems of Web servers and transferred to the clients via HTTP. The scenario changed dramatically when the first programming languages with support for the integration of active components into Web pages appeared. Examples of such active components include presentations of query results to databases or displaying access statistics for Web content. The desire to work with these active contents gave a boost to the development of new technologies.

When we talk about Web-oriented programming languages, we need a platform where these applications can be executed. In conventional environments this is just the operating system on the local computer. The first generation of programming environments followed this concept. By using an interface called CGI, developers were able to execute conventional programs on a server and return the results to the client by treating the output of the program execution like a requested static HTML page. Later on, more complex execution environments were developed, offering higher level services to users (Chap. 6).

In this chapter, we introduce the basic concepts for Web application programming. First of all, we differentiate between client and server side approaches. This distinction horizontally divides the realm of programming approaches for Web applications. Then, we introduce three major concepts that extend static HTML pages, namely generation, extension, and enrichment of HTML code. These mechanisms provide a vertical classification. After we have structured the various approaches, we will provide an insight on existing technologies and map them into our classification. The motivation for the first Web-oriented programming languages was the capability to present data on the Web in a dynamic way. Most of this data is stored in databases, so we will take a brief look at some important database access technologies. To complete this chapter, we provide guidelines which provide decision support for tackling the problem of choosing the right concept for Web application programming.

5.2 Client vs. Server Side Approaches

To get an overview of the basic programming concepts, we classify them into different categories. We will first differentiate the location of execution. Functionality can be provided and executed on either the client or server side. This allocation leads to different requirements for clients and servers with respect to performance and communication infrastructure.

When using a client side approach, the application's program code must first be transferred from the server to the client before it can be executed there. This implies that the performance of the application and the range of functionality are determined only by the capabilities of the client. Performance is of course dependent on the hardware and software resources available on the client. Functionality might be limited by the availability of equipment, e.g. the presence of special hardware devices.

To run the application, an execution environment is required. This might be built into the operating system (e.g. .NET) or may require extra software (e.g. a Java Run-time Environment or a browser plug-in). Clients that offer these high-level resources are called rich clients. They are responsible for a significant amount of the execution, so server operators prefer these kind of clients, as they cause just a little effort on the server side. This makes it possible for many clients to be served by just a few servers. One problem with this approach is that transfer time and net traffic vary depending on the size of the program code and the performance parameters of the network. So the amount of code transferred from the server to the client should not exceed a size well treatable by the communication system. Client side approaches facilitate the implementation of rich graphical user interfaces, as these demand more sophisticated resources of the platform they are running on.

On the contrary, server side approaches execute program code on the server. This way, only static HTML code is transferred to the client. Imagine an online product catalogue. The product data is stored in a database. If a user requests information on a certain product, the corresponding data is read from the database and a custom Web page containing the desired information is generated. This HTML page is then transferred to the client, where it can be displayed without any further effort. Since most of the computation happens on the server, so called thin clients are sufficient. This term depicts clients with scare computing resources that are in general just capable of displaying information and handling user interaction. Thin clients are of course cheaper for the users on the client side, so they will be happy to use such a system. On the other hand, the service provider on the server side has to bear most of the cost of the application. Thus, if you want to handle many clients, it requires enormous server performance. Pages created on the server side underlie the restrictions of the HTML standard and thus have limited capabilities for presentation compared to full-fledged applications.

5.3 The Session Problem

One big problem for server side approaches is the session problem. Conversation-oriented applications demand identification of users throughout a whole session. But with HTTP being stateless, the problem arises of how to keep track of users while they navigate around a Web site. The solution is to define a unique session identification token. This token is generated for each user when visiting the first time and must be saved

for all further visits. There are mainly three different approaches to implement this session concept: HTTP cookies, URL coding, and hidden fields.

HTTP cookies are small text files stored on the file system of the client. Web servers can tell browsers to store certain information in such files. Typically, the unique identification token is stored this way. Entries in these files might look as depicted in Fig. 5.1.

```
...
www.myWebsite.de    user_name    John Smith
www.shop.com user_id      421128
...
```

Fig. 5.1. Cookie example

Whenever a user visits a Web site, the Web server can ask for a cookie. If the client has already visited this server, it will have one. Otherwise, a new cookie will be generated. The server can read out the identification token to recognize the user again. Cookies have a limited lifespan. When they are generated, a time interval is specified for which they are valid. Users can adjust the cookie policy of their browser for security reasons. If cookies are disabled by the user, session handling via cookies fails.

The second approach to the session problem is URL coding. It is possible to transfer the session identification along with the requested URL to the Web server. In practice, such an HTML call looks as depicted in Fig. 5.2.

```
http://www.shop.com/index.html?user_id=421128
```

Fig. 5.2. URL coding example

The server can read the session identification directly from the URL. While cookies allow the storage of user information over several sessions, URL coding is limited to one single session, as the additional parameters in the URL have to be passed on while navigating around the Web site. On every initial visit, users have to register to obtain their identification parameter.

Hidden fields are the third approach to transfer session identification. The token is stored in a field of an invisible form (Fig. 5.3). This approach is supported by all browsers but has to be coded into all HTML pages on the server consequently. Just as with URL coding, users cannot be recognized across session boundaries and have to register initially.

As you have seen, there are several approaches to solve the session issue for server side concepts. However, each of them has significant drawbacks. Thus, the alternative chosen for a specific Web application must be selected thoroughly.

```
<html>
…
<form name="Hidden" action="http://www.shop.com">
  <input type="hidden" name="user_id" value="421128">
</form>
…
</html>
```

Fig. 5.3. Hidden fields

5.4 Generating , Extending and Enriching HTML

In the last section we distinguished client side from server side approaches. In this chapter we will complement this horizontal distinction by a vertical one. We distinguish three different cases:

- generation of HTML
- extension of HTML
- enriching of HTML.

Generation of an HTML page means that the HTML page to be delivered is the product of an execution on the server. Extension of HTML describes the process of evaluating script language elements inside an HTML page to create the final page that can be delivered. Finally, enriching of HTML describes how certain parts of the page are substituted by, for example plug-ins. As one can see, the generation approach is exclusively useful for server side approaches, the enrichment approach is exclusively suitable for client side approaches, and the extension approach can be implemented on either client or server side.

Let us first look at the client side approaches. We distinguish between the extension of HTML by scripting languages and the enriching of HTML by applications or plug-ins. Figure 5.4 provides an overview of the available concepts.

Client Side Approaches	
Extending HTML	JavaScript VB Script
Enriching HTML	Java Applet ActiveX Controls

Fig. 5.4. Client side approaches

The following example (Fig. 5.5) shows an extension of HTML by JavaScript. The statements are embedded within a regular HTML file. The instructions are executed by the browser on the client. In this example, the current date is printed out. The scripting code that generates the current date is an example of an extension to the HTML page.

```
<html>
<body>
<p>Current date is
    <script language="JavaScript">
    var DateVar = new Date();
    var day = DateVar.getDate();
    var month = DateVar.getMonth() + 1;
    var year = DateVar.getYear();
    document.write(day + "." + month + "." + year);
    </script>
</p>
</body>
</html>
```

Fig. 5.5. Extending HTML

Web pages can be extended whenever you want to combine mostly static content with a dynamic component. Another approach to enrich Web pages is to include references to external active components, e.g. Java applets (Fig. 5.6). When the HTML page is loaded, the referenced active component, here the Java applet, is executed. This requires loading the active component from the server to the client. To be able to execute such applications, the client must have installed a special environment depending on the type of the application. Examples of this technology include Java applets and ActiveX controls.

```
<html>
 <body>
  <h1>This is an applet</h1>
  <object classid="java:Hello.class"
    codetype="application/java-vm" width=150 height=100>
 </body>
</html>
```

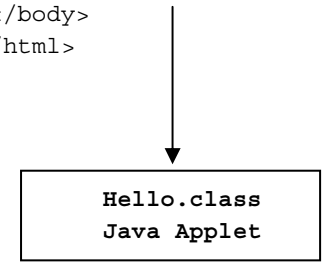


Fig. 5.6. Enriching HTML by client side applications executed in the browser

The enrichment approach is advisable whenever the capabilities of HTML are insufficient: for example, when you need more profound presentation technologies like animations. However, it requires a decent amount of programming effort, as the external active elements have to be programmed in their specific language. Furthermore, they require support on the client side for the execution of such active elements. In the case of an applet, a Java Run-time Environment is necessary. Clients without such an environment will not be able to view the applet.

As we have mentioned before, on the server side there are two mechanisms available: the generation and the extension of HTML. The first class comprises CGI programming and Java servlets. Generation of HTML is the right choice if the overall Web page is very dynamic. Good example scenarios are applications that require a high amount of personalization. This means that, depending on which users view a page, content and visualizing elements vary strongly.

The approach of extending HTML is comparable to the extension approach on the client side. First of all, there is a template file. Upon request, this template is evaluated and scripting commands are executed to create the final page which will be delivered to the client. Commonly known technologies realizing these concepts are Server Side Includes (SSI), Active Server Pages, and Java Server Pages. A classification of both the extension and generation of HTML can be seen in Fig. 5.7.

Server Side Approaches	
Extending HTML	Scripting Languages (ASP, JSP, PHP) Server API, Server Side Includes
Generating HTML	CGI Java Servlet

Fig. 5.7. Server side approaches

Now that we have introduced the various approaches, the question arises of when to use what. There are two main criteria that can help you make that decision:

- the dynamics of the content and
- the performance distribution between client and server.

Content that is changing frequently suggests the use of server side generation. If the content does not vary too much over time, extension might be a good solution. The external active elements then have to update only their information, whereas the HTML backbone stays the same. Where to put the emphasis of the application depends on how much performance you can expect from your clients. If your clients offer sufficient resources, it makes sense to source out computation from the servers to relieve them and enable better scalability. If you cannot expect your clients to be too powerful, server side approaches are the right solution. But bear in mind that they come at the cost of a higher load per client and of larger amounts of data to be transferred.

Figure 5.8 shows the client and server side approaches that will be discussed in the following sections classified according to client and server side approaches. Before we discuss these approaches individually, a more detailed classification of programming concepts will be presented.

Client Side Approaches	Server Side Approaches
Java Script	Scripting Languages (ASP, JSP, PHP)
VB Script	Java Servlet
Java Applet	Server API
ActiveX Control	Common Gateway Interface (CGI)

Fig. 5.8. Client and server side approaches

5.5 Client Side Approaches

After having provided this broad overview, we will now focus on concrete implementations. As Fig. 5.4 indicates, there are two sorts of approaches on the client side: HTML extension and HTML enrichment.

5.5.1 JavaScript and VBScript

JavaScript [FlFe01] and VBScript [Loma97] are client side approaches to extend HTML. JavaScript can be embedded directly into an HTML page or transferred separately by referencing a JavaScript file (suffix .js). JavaScript is a scripting language that must be executed on the client side. It is often programmed as event triggered. Typical events are the process of loading a Web page or clicking on a button. JavaScript is an object-based language that follows a hierarchical object model. The downside is that this object model is proprietary and cannot be extended by user-defined classes. The typical application fields of JavaScript are less complex extensions of HTML pages on the client side. Inputs in forms can be validated and counters can be implemented. Along these lines it is possible to open windows or frames and to change them. Among other things, comfortable menus can be implemented that enable better navigation. VBScript is a very similar approach, originally developed for Microsoft's Internet Explorer and by far not so wide spread.

5.5.2 Java Applets and ActiveX

Java is a platform-independent language interpreted and executed by the Java Virtual Machine (JVM) [Java04]. Applets can be understood as small applications that are embedded into Web pages [Appl04]. These applets are transferred to the client upon request. Their reference is included in an HTML page and the output of the applet execution is presented in a rectangular sector in the Web browser or in a separate window. The Java applet `Hello.class` (Fig. 5.6) will be executed in a rectangular window of width 150 and height 100.

Java applets allow the programming of small applications with the full power of all the services available on the Java platform. Amongst other things, the applets can do calculations, access databases, create network connections, and most importantly create graphical user interfaces. Compared to JavaScript, applets can be considered much more powerful, as they comprise a full-fledged programming language. Because Java applets are not in binary format but in intermediate byte code, they are executable on various hardware platforms. The Java source code for the applet is firstly translated into this platform neutral byte code. This code can be executed by a special environment, namely the

JVM. It maps the neutral byte code onto binary instructions that can be understood by the local hardware. As Java is an object-oriented programming language, applications (and thus applets as well) consist of various class definitions that cooperate to form the overall functionality. To load each class separately over the network would cause unnecessary overhead. Thus, class definitions are packaged into Java archives (suffix .jar). This package can be loaded in one transfer and the virtual machine will take out of it whatever definitions it needs.

The Java applet security concept is called the "sandbox". A sandbox is an additional sphere of control in which running applets are encapsulated. A user can specify the security policy for the sandbox, allowing or disallowing certain capabilities. Examples of such capabilities are the right to establish network connections or access local file systems. The default settings of a sandbox allow Java applets to establish connections just to their originating Web server (where the jar file was loaded from). Typically, Java applets are deployed in applications that need a lot of client side functionality or many presentation capabilities.

ActiveX [Acti04] is a collection of techniques, protocol, and APIs to realize network-based applications especially for embedding multimedia content into Web sites. The concept developed by Microsoft is similar to the Java applet approach. Both aim at the integration of executable programs into Web sites. ActiveX resides on the Microsoft component model (COM – Common Object Model, DCOM – Distributed Common Object Model) [DCOM04] [Kirt98] (Chap. 6). Therefore it is platform dependent. Microsoft Windows COM subsystem and DCOM support are needed to execute ActiveX applications on the client. To run an ActiveX component a control container is needed, which is integrated into Microsoft's Internet Explorer.

5.6 Server Side Approaches

Fig. 5.7 introduces two wide spread approaches to generate HTML pages on the server side, namely CGI and Java servlets. After we have shown how they work, we will present the approaches for server side extension of HTML.

5.6.1 CGI

CGI [CGI04] was one of the first approaches for the implementation of dynamic Web applications. While HTTP describes how to realize connection between Web servers and browsers, CGI defines a way of communicating between the Web server process and another process, running a local application. One of the facts that made is so successful was its programming language independence.

Upon request, the Web server starts the target application specified by CGI in a separate process (Fig. 5.9). Parameters that were passed to the server inside the request by coding them into the URL are passed further on to the newly created process by setting environment variables with the corresponding values or as the standard input stream for the process [Stev92]. The target application can read the parameters either from these variables or from standard in. The output of the application to standard out is read by the Web server and afterwards sent back to the client as the requested HTML page. Thus, the output of the helper application should form a valid HTML document.

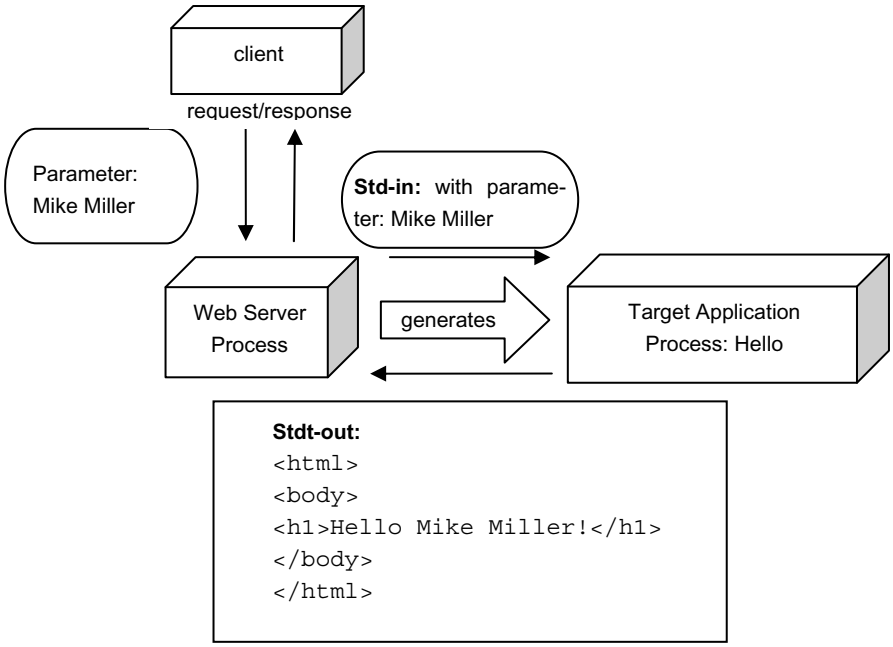


Fig. 5.9. CGI processing

This also explains why CGI is independent of any programming language. All languages offer means to access standard in, standard out and environment variables. Thus you can write your target application in your favorite language. An example CGI request is shown in Fig. 5.10. In this request, a variable called name is encoded into the URL. The standard input of the CGI application `Hello.c` is used to transfer the variable into the program. The parameter Mike+Miller is divided into the parts Mike and Miller. It seems to the program as if it was invoked as Hello Mike Miller. Thus the two parameters in argv are Mike and Miller.

As mentioned before, for every CGI call there is an individual process being created. This allows good isolation between processes running in parallel, but can lead to bad performance due to process management overhead. FastCGI [FCGI04] is an approach to overcome this issue by using threads instead of processes for each CGI call.

5.6.2 Java Servlets

Java servlets [Serv04] present a prominent server side approach to generate HTML pages. Advantages arise from the use of Java: class definitions can be loaded dynamically, which allows the developer to extend Web applications by additional modules without having to restart the Web server. Web applications developed as Java servlets are also portable; they only require the inclusion of the JVM into the Web server. The Java servlet API introduced by Sun Microsystems gives the possibility of developing Web applications using object-oriented concepts independent of the Web server platform.

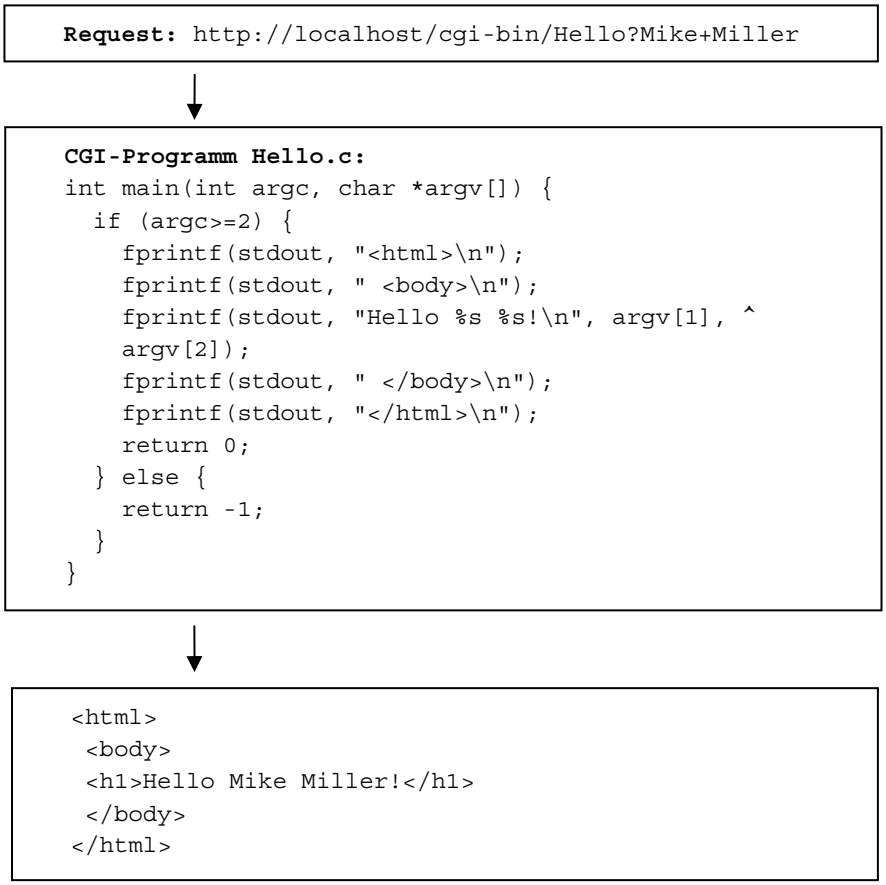


Fig. 5.10. CGI example

A Java servlet is a Java class that extends the functionality of the J2EE server. Instead of looking for a file matching the URL requested on the local file system, the request and its context are passed on to a Java class. This class must extend a certain interface (`javax.servlet.http.HttpServlet`), which ensures that it processes the context of the request provided with the constructor in the specified way. On invocation, the Web container creates an instance of the required class and passes the context information to the newly created instance. The servlet interface also provides the means for the creation of appropriate reply messages. In the simplest case, it writes HTML output to a character stream, which will form the HTTP response.

Compared to other technologies, servlets offer certain benefits. Servlets are compiled Java classes, so they run faster then server side scripts that would have to be interpreted. Servlets offer a certain amount of security, as they run inside the JVM, which provides measures to limit the access of the servlet by using sandbox technology. Portability is

also an important issue. Servlets can be moved both on the source code level as well as on the byte code level. Java byte code is just as standardized as Java servlet source code. And finally, servlets benefit from the rich set of standard services available in the standard libraries that come with the JVM. Thus programmers can rely on those common services and do not have to program everything from scratch, allowing them to focus on programming business logic.

Java servlets offer the complete functionality of Java class libraries including a large part of the network and database support. Java servlets are not limited to HTTP, they can work together with all protocols following the request/response principle. A Java servlet class is able to treat several requests in parallel by generating a servlet instance for each request. The example in Fig. 5.11 shows a simple servlet that returns a simple HTML page containing the "Hello World!" message.

In contrast to Java applets, Java servlets are executed on the server side. Code does not have to be transferred to the client side. Java servlets are also able to store session states. Nevertheless, both concepts do complement each other: Java servlets can be used to build highly dynamic HTML pages on the server side, while Java applets are adding some client side logic and presenting the HTML pages in an appropriate way.

Java Servlet:

```
import java.io.*
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet {
public void doGet(HttpServletRequest req, ttpServletRe-
sponse res) throws ServletException, IOException {
       res.setContentType("text/html");
       PrintWriter out = res.getWriter();
       out.println("<html>");
       out.println("<html>");
       out.println("<body>");
       out.println("Hello World!");
       out.println("</body>");
       out.println("</html>");
}}
```

Result:

```
<html>
<body>
Hello World!
</body>
</html>
```

Fig. 5.11. Example of a Java servlet

5.6.3 Server API

In contrast to the approaches described so far, most Web server vendors also offer specific server extensions, called server API. A server extension allows one to dynamically load user-defined functions (custom functions) into the Web server by using a vendor-specific server API. In addition to these user-defined functions, a server API can offer server application functions (SAF). These are proprietary Web server extensions introduced by the vendor of the Web server.

Extensions are loaded during the first request to the Web server and stay in memory as long as possible and needed. Compared to CGI, the communication between Web server and server functions is done via function calls. Session handling can also be facilitated in the Web server.

There are also some disadvantages of using a server API: by including functions directly into the Web server debugging gets more complicated and errors in the server API application can crash the whole server. Common representatives of server APIs are NSAPI by Netscape [NSAP04] or ISAPI by Microsoft [IIS04] [MISA04].

5.6.4 Server Side Includes– SSI

The concept of SSI allows the developer to embed instructions into HTML documents which are executed on the Web server [SSI04]. The HTML page including the results produced by these instructions are afterwards transferred to the client. Typically, SSI customizes parts of an HTML page at the specific request of a user.

SSI is not standardized; thus, not every Web server supports SSI. Also, there is no standardized language to write SSI statements; each Web server provider offers their own proprietary format for SSI.

Typical application fields for SSI are small modifications of a HTML page like the inclusion of a timestamp (e.g. the last modified date of an HTML page). Database access can also be done by some Web servers using SSI.

The example in Fig. 5.12 shows how the last modification date and a menu bar are included in an HTML page. The variable LAST MODIFIED is interpreted and replaced by the Web server when the HTML page is requested. The inclusion of the menu bar is a typical customization of an HTML page depending on the user request. The advantage of doing this with SSI is that it is done and controlled at a central point.

```
<! -- #echo var=„LAST_MODIFIED“ -->
Result: Monday, 06.October-2003 07:39:00 GMT

<! -- #include virtual=„menubar.html“ -->
```

Fig. 5.12. SSI example

5.6.5 Server Side Scripting Languages

Server side scripting languages work similar to SSI. Server side scripting statements embedded in HTML pages are parsed and executed by the Web server and the results are

sent to the client. It would be cumbersome to check all documents for scripting statements. Therefore, only documents with a specific suffix (like .jsp, .asp, .php) are parsed.

Compared to SSI, server side scripting languages offer richer programming functionality. They are comparable to conventional programming languages. The most popular representatives of scripting technologies are

- JSP: Java Server Pages (Sun) [JSP04],
- ASP: Active Server Pages (Microsoft) [ASP04], and
- PHP: Hypertext Preprocessor [PHP04].

We will take a closer look at ASP and JSP. PHP is very similar with respect to functionality and usability and is therefore not discussed further. However, in contrast to ASP, PHP is an Open Source Project.

ASP is a server side scripting approach developed by Microsoft. The scripting languages VBScript (Visual Basic Script) and JScript (Microsoft's version of JavaScript) can be used to enact it. ASP merely provides the infrastructure to execute scripts and enables their interaction with the Web server. ASP is included in the Microsoft Internet Information Services (IIS; [IIS04]).

For server side scripting using Java, JSP is the right choice. If you look at servlets, they often contain instructions to print out huge blocks of static HTML code. The reason is that most Web applications produce pages that change only in data values and not in basic page structure. After all, you want to provide the clients with the same layout every time they look for an article in an on-line catalogue, so they can easily orientate. That is what JSP was developed for. Instead of embedding HTML generation into a Java class, Java functionality is embedded into an HTML file. A JSP page is a document containing fixed template text plus special markup information for including other text or executing embedded logic. The fixed text is always served to the requester just as it appears in the page, like traditional HTML. The special markup can take one of three forms: directives, scripting elements, or custom tags.

Directives are instructions that control the behavior of the JSP page compiler and therefore are evaluated at compilation time. This happens every time a client requests this page. Directives can be compared to compiler settings in the comments of C source code. Scripting elements are blocks of Java code that are embedded into the JSP page. To separate them from the static HTML, they are marked by the delimiters <% and %>. Custom tags are programmer-defined markup tags that generate dynamic content when the page is served. The JSP specification defines a set of standard tags that are available in all platform implementations. The idea behind the tags is to allow for the definition of a reusable set of instructions. Writing the same code over and over again in scripting elements (Java source blocks inside JSP) produces redundancy and is error-prone. Developers can define their own custom tags that wrap commonly used functionality. This approach offers several benefits. First of all, these tags are reusable, whereas scripting elements are not. Secondly, libraries of custom tags provide high-level services for JSP pages that are portable across JSP containers. Custom tags make maintenance a lot easier, as they reduce redundant code. If the developer changes the definition of a tag, the behavior of the tag changes everywhere it is used. One of the greatest advantages of custom tags is that developers can focus on their core skills. The Web page author can focus on creating the right look for the Web page. Invoking business logic is hidden by the use of tags. Programmers, on the other hand, can focus on programming business logic without having to worry about presentation. Thus, a clear separation of presentation from

logic is achieved. An example of a JSP script is given in Fig. 5.13. The actual date is placed in the HTML page.

```
<%@ page language="java" contentType="text/html" %>
  <html>
    <body>
       <p>Hello World!</p>
           <p>
               Today is <%= new java.util.Date().toString() %>
               and it's a beautiful day.
           </p>
    </body>
   </html>

   Result:
   <html>
     <body>
        <p>Hello World!</p>
            <p>Today is 2003.10-10 and it's a beautiful day.</p>
     </body>
   </html>
```

Fig. 5.13. JSP example

The biggest advantage of server side scripting languages is to separate presentation logic from application logic.

5.7 Database Connectivity

So far we have focused on client and server side approaches for Web application programming without considering the question of how to access data stored in a database. The latter is the most important task when dynamic Web pages have to be built up. A number of well-established approaches for data access are listed in Fig. 5.14. The figure also shows for which programming concept a database access method is applicable. In the following subsection we present the most common approaches whereby their pros and cons are also discussed.

Programming Concepts	Database Connectivity Concept
Client Side Approaches	
Java Applet	Java Database Connectivity (JDBC)
ActiveX	ActiveX Data Objects (ADO)
Server Side Approaches	
JSP	Java Database Connectivity (JDBC)
ASP	ActiveX Data Objects (ADO)
PHP	Proprietary, Open Database Connectivity(ODBC)
Java Servlet	Java Database Connectivity (JDBC), SQLJ, Java Data Object (JDO)
Server API	Dependent on the server product used
CGI	Dependent on the programming language used
SSI	Dependent on the server product used

Fig. 5.14. Concepts for database connectivity

5.7.1 Open Database Connectivity (ODBC)

Similar to JDBC, ODBC is middleware to access heterogeneous data sources that allow dynamic queries. ODBC-SQL defines a language standard as the idiom of the SQL standard.

The ODBC [Geig95] architecture (Fig. 5.15) consists of an application responsible for the interaction with the user and calling ODBC API functions. A layer above the ODBC driver manager loads the drivers requested by the application and delegates the function calls to the concrete drivers, e.g. an Oracle ODBC driver or a Microsoft SQL server. The concrete driver processes the function calls and sends the SQL queries to the data source. The driver encapsulates the whole database and network functionality.

In ODBC, three different types of drivers can be can be distinguished. The level 1 driver allows one to access files as databases. For this purpose the driver must provide a complete SQL data engine. The level 2 driver allows access to typical client/server systems. The ODBC-SQL is translated into the database management SQL. The level 3 driver allows one to introduce a separate connection machine between the client and server. Such a gateway leads to better performance and allows several databases to be accessed from the gateway. The clients only have to use the protocol supported by the gateway and not all protocols from the data sources.

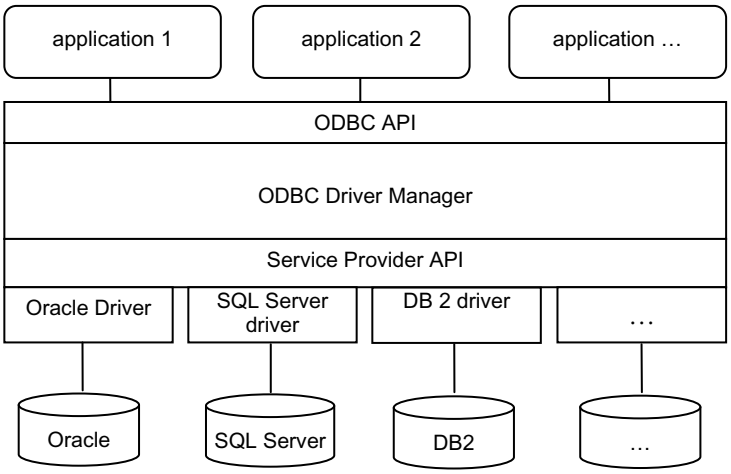


Fig. 5.15. ODBC architecture

5.7.2 Java Database Connectivity (JDBC)

JDBC [JDBC04] is a well-established approach to access databases from Java applications. JDBC is a Java-oriented approach similar to ODBC supports database connectivity to various database products based on SQL functionality. The core of JDBC is a collection of classes located in the `java.sql` package allowing the use of SQL statements, result sets, and database metadata. Like ODBC, JDBC is based on the X/Open SQL Call Level Interface (CLI). JDBC itself supports four different types of drivers depicted in Fig. 5.16 to access a database – all administered by the driver manager (java.SQL.DriverManager):

- JDBC–ODBC bridge driver (type 1 – Fig. 5.16 left)
- Partial Java JDBC driver (type 2 – Fig. 5.16 middle)
- Pure Java JDBC middleware driver (type 3 – Fig. 5.16 right)
- Pure Java JDBC net driver (type 4 – Fig. 5.16 right with direct database access).

Type 1 does not provide direct access to the database. Instead, a mapping from JDBC to ODBC is done and the database is accessed over ODBC. As a result the functionality of JDBC is limited to the capabilities offered by ODBC. To realize type 1 JDBC driver connections, ODBC binary code is needed on the client.

Type 2 differs from type 1 only in the fact that an ODBC connection is not used. Instead, a direct connection to the database is opened, which requires a database-specific driver on the client. This increases the efficiency, but the price paid is high complexity. Both alternatives 1 and 2 are not very suitable for Web applications. This is due to the presence of special purpose code on the client side. Java applets cannot work with these types. The Java applet security concept disables the use of drivers installed on the client.

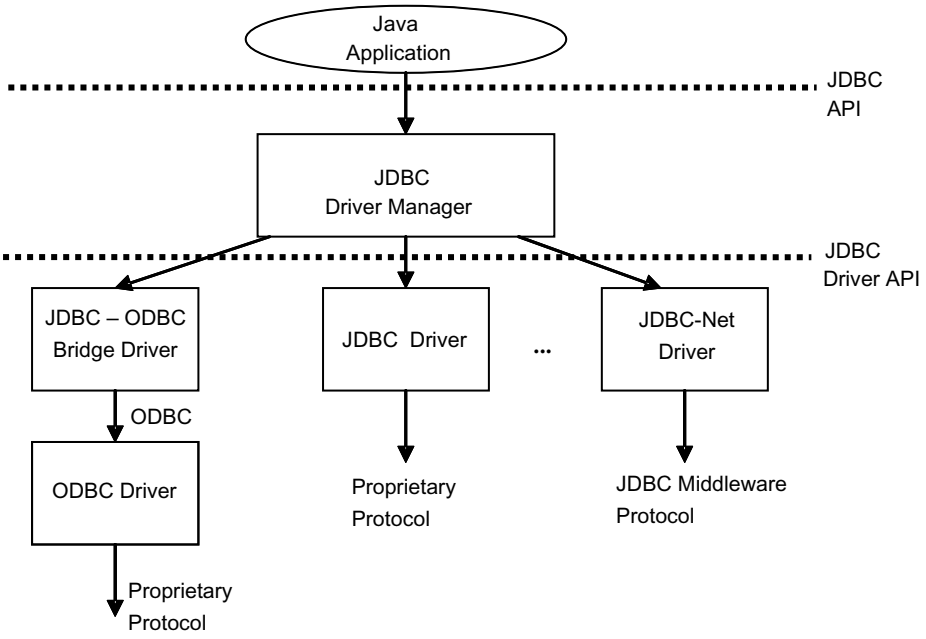


Fig. 5.16. JDBC architecture

Drivers of types 3 and 4 rest on platform-independent drivers written in Java. They can be loaded dynamically from a server to a client. Because of this they fit best in to the context of Web applications. Type 3 drivers translate JDBC instructions into a middleware-specific protocol. Subsequently they can be translated into one or more database protocols. The middleware is a layer of abstraction that encapsulates the database and allows their substitution without recognition from the client. Type 4 drivers access the database directly. The JDBC instructions are directly translated into the network protocol. The direct connection enables high performance.

5.7.3 SQLJ

SQLJ [SQLJ04] is a standard designed by leading database vendors to embed SQL in Java applications. SQLJ only supports static embedding of SQL, which implies two facts: the SQL statements can already be checked for type consistency at compile time and the statements can therefore be optimized. However, this processing leads to the restriction that SQL statements cannot be changed or adapted at run time. The SQL preprocessor translates the SQL statements into standard Java code using the Call Level Interface (CLI) to access the database. SQLJ consists of three parts:

- Embedded SQL (part 0): Describes the embedding of static SQL statements in Java.
- Java Stored Procedures (part 1): Functions that can be stored and executed in the database directly. This enables local access to the data.
- Java classes for SQL data types (part 2): This part describes how Java classes can be used to implement SQL data types.

SQLJ statements can be embedded directly into Java code using #sql as identifier. The SQLJ compiler is used as a preprocessor to prepare the statements. Java source code is generated from the SQLJ source code and the compiler checks the statements during compile time with respect to syntax and semantics. The implementation of SQLJ rests on JDBC. Both approaches – JDBC and SQLJ – have advantages and disadvantages. SQLJ supports only static SQL statements. Although such a statement cannot be changed at run time, it is favorable for performance and robustness. Due to SQLJ's static nature, its applications are compact and easy to read.

5.7.4 Java Data Objects (JDO) and ActiveX Data Objects (ADO)

So far the focus of this discussion has been set on accessing databases using SQL. The most modern programming languages rest on object orientation, which means that all data in the programs is encapsulated in objects. From this fact a central issue arises: How can objects be made persistent?

In the Java environment, there are several solutions to this problem. First of all, objects can be written directly into a stream (which could be a file) using serialization. Accordingly, if you read the object out of the stream, you get your original instance back. One problem with this approach is that data can only be accessed in a serial manner, which entails performance lags. Concurrency and recovery are also an issue, as they are not treated in this simple model. JDBC allows to store data into a relational database. The mapping between objects and relations is left to the developer, which makes the solution complicated for real-world applications. Skills in OOP and in relational database design are needed, as inheritance hierarchies must be manually mapped to relations and so on.

To overcome these problems high-end persistence mechanisms have been developed. Java Data Objects (JDO), developed in 2002, are one of the newest technologies. The goals of JDO are:

- transparency of persistence
- independence of the data store
- transactional semantics
- interoperability.

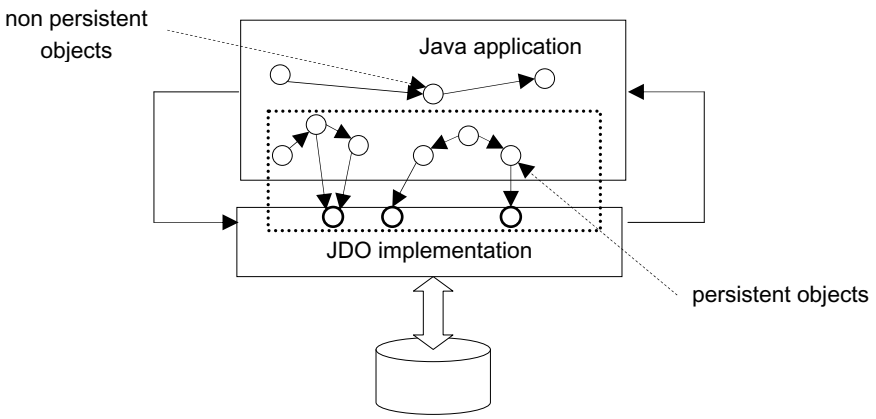


Fig. 5.17. Java Data Objects (JDO)

The general idea is depicted in Fig. 5.17. A Java application consists of multiple objects, each of which can transparently switch to the rest between persistent or in-memory states. In other words, each object can be written to or loaded from the data store without affecting the rest of the application objects.

JDO is a package that can be used in Java. JDO offers first-class objects, which correspond to data units in the used data store (e.g. a tuple in a relational database). These first-class objects are uniquely identified within JDO and can be queried using an ODMG-based query language called JDO Query Language (JDOQL). Queries can use filters (predicates), sorting, parameters, and variables. JDO allows for a comfortable data management in Java without the need for knowledge of the underlying data store. Moreover, the influence on the persistent data model is small and in control of JDO.

ActiveX Data Objects (ADO) are the technological rival of JDO in the Microsoft environment. While ODBC is only a functional API and the application using ODBC has to do the mapping between the internal data model – the classes and objects – and the API functions, ADO offers data from different data stores, whereas all data are viewed as a recordset independent of the type of the data store. The recordset object offers several methods to work on the data: new recordsets can be generated (AddNew), existing recordsets can be searched by specific criteria (Find), recordsets can be navigated sequentially (MoveNext, Move Previous) to mention a few methods, or at random by using a count method (RecordCount).

In the overall JDO and ADO there are services allowing for an automatic mapping of objects to the used data store. They do not only play an important role in the conventional approach elaborated in this section, but for the component technologies in the next chapter.

5.8 Cookbook of Recommendations

This section introduces programming approaches for building Web applications. A whole bunch of application examples are given. However, due to the multiplicity of approaches, it is still hard to decide which approach to favor in what situation. Therefore, we present a collection of guidelines to provide decision support.

However, the reader should not expect the support in this section to be sufficient for completely answering all questions regarding the selection of a programming approach for Web applications. This decision often depends on the project context and is often only slightly controllable by technical requirements. Answers to such non-technical questions will not be given here. The following questions merely point to issues that identify relevant aspects for these non-technical questions:

- What products were used in the past?
- What experiences does the project team have?
- What is the budget?

For instance, the last question can determine that PHP as open source product must be used. If the technical requirements do not contradict this decision totally, the whole decision process might already be complete.

The questions that are tackled in this section concern technical facts and are of the following form:

- Should a client- or server-based approach be used?
- Should a combination of both approaches be favored?

- Which concept is needed to implement a complex graphical user interface (GUI)?

A first overview of the programming approaches introduced in this chapter is given in Fig. 5.18. The three main functional parts of a Web application are listed vertically: presentation, logic, and data. The introduced programming approaches are arranged horizontally. The goal of the figure is to give a quick overview of the main usage areas of the programming concepts. For instance, JavaScript is classified as a concept that supports well presentation tasks on the client side and is also quite useful for implementing logic on the client side. As another example, Java servlets are good for implementing logic on the server side. Quite naturally, all the database connectivity approaches in Sect. 5.7 are only suitable to support data access on the server side.

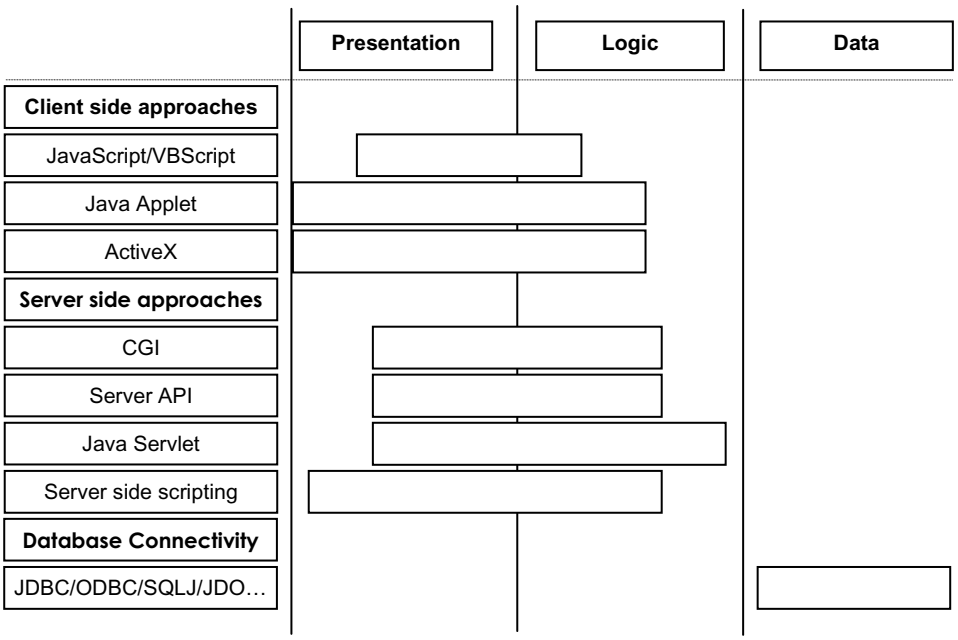


Fig. 5.18. Overview of the different approaches

As the next step towards structured decision support, the decision diagrams of Fig. 5.19 and Fig. 5.20 are used. The two diagrams are based on the principle that implementations should happen as much as possible on the server side in order to keep the client side as thin as possible.

The first significant question in Fig. 5.19 is whether an application – specifically the user interface – can be completely implemented in pure HTML. This means that an HTML page can be completely created on the server side. The complete HTML page is then transferred to the client. If pure HTML is not sufficient, then a “lightweight” client side approach is suggested. For instance, a client side scripting approach (e.g. JavaScript) should be considered to fit the requirements with respect to logic and presentation implementation. Combining this client side approach with some server side concepts might fulfill the requirements. If even this is not a solution, then a “heavyweight” client side

approach like a Java applet or ActiveX should be used. It might be the case that this client side implementation makes a server side implementation superfluous.

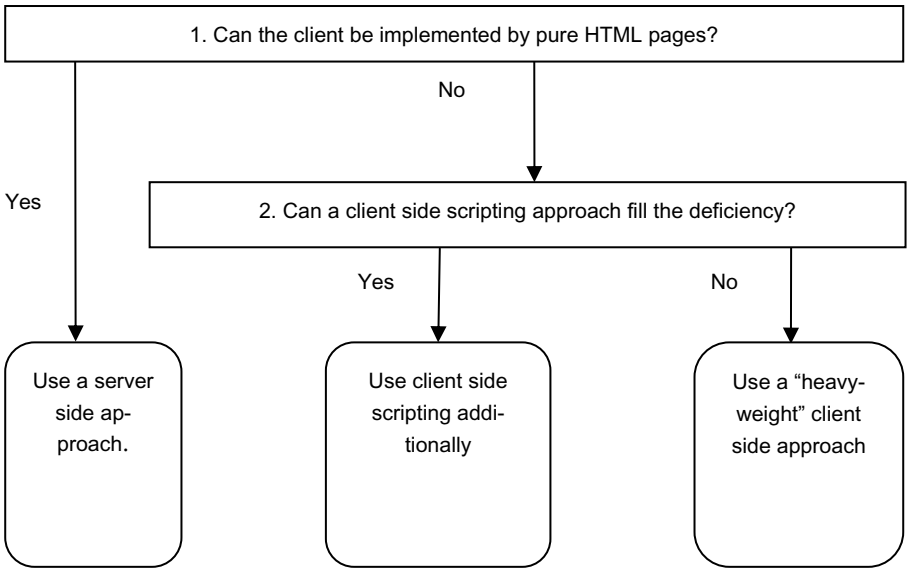


Fig. 5.19. Decision diagram – client side

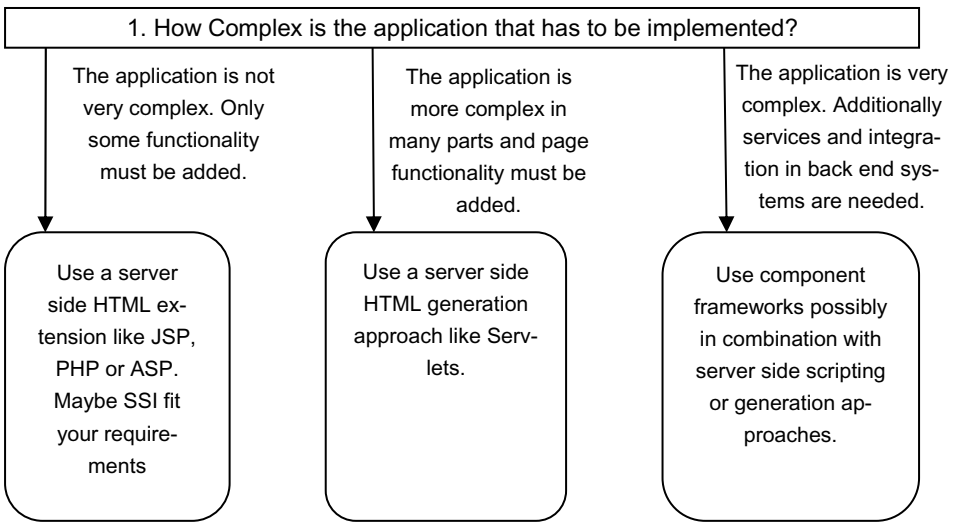


Fig. 5.20. Decision diagram – server side

As a result of the processing of the decision diagram in Fig. 5.19 the client side implementation concept should be chosen. Subsequently, the server side implementation must be determined. The decision diagram in Fig. 5.20 addresses this issue.

A central question concerns the complexity of the Web application to be implemented. If the Web application is not too complex some server side HTML extension mechanisms (e.g. JSP, ASP, or PHP) should be utilized. For more complex requirements an HTML generation approach (e.g. Java servlet) must be chosen. If this is not sufficient, a complex server side implementation is required. In such a case it is recommended to choose an application framework like J2EE or COM+ to implement the Web application. The next chapter will introduce these frameworks and will discuss their applicability.

6 Component-Oriented Software Development

In this chapter, we will introduce the concept of software components. This term describes a special way to structure software into functional, reusable units. To show the reasons why such an approach is useful, we will first discuss the problem of the reuse of code and why it is important to anyone who develops software. We will then discuss possible solutions to this problem. The outcome of this discussion will be a list of requirements, which we will use as criteria for the definition of software components. After we have defined these requested features, we will show how they can be realized.

Having developed all these ideas, we will take a look at how to use components in practice. Therefore, we will define what middleware is, as this technology is required for using components in our scope. Then we will follow the historical evolution of the technologies. Firstly, we look at RPC. Then, we will extend this basic approach by object-oriented features, leading us to RMI and object brokers, namely CORBA. After having examined this middleware approach, we will take a look at complete frameworks for developing component-based Web applications: the Sun J2EE Framework and Microsoft's .NET. Finally, we will present the latest development by the OMG, the CORBA component model.

6.1 Code Reuse

Code reuse has been and still is one of the biggest and historically most important issues of software evolution. The question is how parts of existing applications can be reused in other projects. We will refer to these parts as packages of code. The intention behind this idea is to avoid rewriting existing code over and over again.

Reusing code provides certain benefits. First of all, writing code is always an error-prone task. If you have ever written code, you know that there are always some slips of the pen sneaking in. By reusing existing code that has proven to be error free, you will avoid making the same errors again. Secondly, a package of code that can be reused will be used by many people. Thus, errors will show up earlier, as the package is used more intensively. This also results in another advantage: all the users benefit from corrections in the common package. This is beneficial not only for the correction of errors, but also for the extension of features. If packages are made more powerful, all applications using them can benefit from these extended capabilities. Another reason why code reuse is good is the ease of cost. Instead of writing everything from scratch, developers can take packages off the shelf and combine them with their hand-written code, thus saving time. This keeps development costs down as it shortens the development cycle.

The reader who develops just small projects might ask why this should be an issue. We are sorry to report that there is no such thing as a little project that is small enough not to benefit from good structuring. And one effect of writing applications in such a way that parts can be reused is that the source code becomes well structured and maintainable automatically. Now, we will discuss the alternatives of how to implement code reuse.

The first approach is to distribute source code. The programmer just has to add the packaged code to the existing one and compile it all together. This solution worked well in the beginning of the computer era, as all programs were written in the standard lan-

guage C [KeRi98]. Standards like POSIX [Posi03], which defined a standard environment, made it easy to write programs which would run on different hardware platforms. But there are problems with this approach.

First of all, the combined source code has to be recompiled, which takes extra time. Secondly, it is assumed that there is one common programming language used by all programmers. This is not true, as there exists a huge variety of languages, every single one with its special benefits and drawbacks. If you want to distribute source code, you are compelled to use the same language as the author of the package you want to use. Finally, if you distribute a package in source code, everyone can read and reproduce what you have done inside the package and how you did it. Implementation hiding is not possible. (The whole discussion on whether or not to publish the source code of the Microsoft Windows Operating Systems illustrates this fact.) Concluding, it can be said that distributing source code is a first attempt, but is certainly not the best solution imaginable.

But programs do not have to be compiled all at once. When programming big applications, the source code is typically split into many files, which not only makes the whole project easier to understand, but also allows for compiling only those parts of the application that have been changed. To understand this, we have to delve a little into compiler theory [PiPe91]. When an application is "created" from source code, two things happen. Firstly, the alphabetical instructions, written in the code of the programming language, are translated into binary instructions that can be understood by the computer hardware, the so-called object file. To create the final executable code, all the required object files have to be linked together. This approach allows the definition of so-called statically linkable libraries. The idea is to provide packages of useful code as precompiled object files, not as source code. Programmers can write their own code and link it together with these libraries. Although this approach is commonly used, it still has certain drawbacks. The most important one is that the combination of package code and self-written code is done during compile time. Thus a project has to be linked again upon changes in the library code. Especially if existing applications are to profit from upgraded versions of libraries, linking the whole application every time a new library version shows up is too tedious to be practical.

We want the consolidation of self-written code with provided libraries to happen at run time, not at compile time. This is where dynamic linkable libraries come in. The idea is basically the same as with libraries based on object files. The difference is that the library functionality is not built into the binary code at compile time. Rather the binary code will make calls to the library just when it is running. This concept is well known in the Microsoft Windows world as DLLs. In the Unix world, the dynamic libraries are called modules. This approach suffers from certain issues as well. First of all, there is the problem of versioning (Windows programmers know this as "The dll hell"). If you compile static library code into your application, you can be sure that the library code will be available when the program is executed, as it is built into the binary code. With dynamic linking, you cannot be sure that the machine the program is installed on has access to the required libraries (or the right version of these libraries).

As the title of this book implies, we are mainly interested in developing applications for the Web. This raises another issue that has not yet been addressed. The approach of dynamic libraries works well, but only inside the same address space. If an application is spread across different processes or different machines, you cannot use dynamic libraries, because you cannot call a library which resides on another machine.

The solution to the problem of distributed applications is software components. Before we look at concrete solutions, let us realize the requirements and characteristics that have to be met.

6.2 Components

As we have described in the last section, we require a mechanism that supports the concept of code reuse and is applicable when creating distributed applications by splitting them into functional units that can easily be spread across host borders: software components. To understand what these are, let us take a look at some definitions of this term. J. Harris, President of CI Labs, said in 1995: "A component is a piece of software that is small enough to create and maintain, big enough to deploy and support and with standard interfaces for interoperability." R. Orfali, D. Harkey, and J. Edwards wrote in their book *The Essential Distributed Objects Survival Guide* [OrHE96]: "A component is a reusable, self-contained piece of software that is independent of any application." Further definitions can be found in [Emme00] and [Szyp97].

To have a common understanding of what we believe components to be, we will provide our own definition, which is compatible with those above. In our context, we define components to be pieces of software that are:

- self-sufficient and self-contained,
- programming language independent,
- location transparent, and
- deployable in a container.

6.2.1 Feature 1: Self-sufficiency

The term self-sufficiency in this context describes the fact that a component is a functional unit (sometimes referred to as "black box" behavior). If you want to use a component, you provide the input parameters, hit the Start button, and the component does what is was programmed to do. After a while, you get the results. There are no means to look inside the box and see what is happening. The black box paradigm also implies that there is a well-defined interface to the outside world. Only those capabilities offered via this interface are available. Therefore, components follow the concept of implementation hiding [Ecke02]. The concept of hiding is also applied to the internal data, called information hiding. Temporary results or internal auxiliary data are hidden from the outside world. As a component is a functional unit, it contains everything it needs to provide the functionality described by the interface. Thus it is not a wrapper for calls to another piece of software, but self-contained. To put it bluntly: it comes with everything it needs. If you look at this from a conceptual point of view, component borders do not cut across internal functionality lines.

6.2.2 Feature 2: Programming Language Independence

We require components to be usable independent of the programming language they were created in. Remembering the concept behind code reuse, the goal was for the programmers to be able to pick components that fit their requirements and compose them to build the desired overall behavior. This selection should not be limited by an understanding of the programming language a component was written in. Furthermore, programmers should have the freedom to write in their favorite programming language. An expert

on sorting algorithms will thus not have to learn a special language to pass on expert knowledge useful to the world, but can write in a personally preferred language.

6.2.3 Feature 3: Location Transparency

Another important feature we demand for components is location transparency. Especially for distributed applications, this is a very important capability. When creating applications for the Web, programmers split functionality into several tiers (Chap. 3). These fragments have to cooperate to realize the overall application functionality. Therefore it should be easy for a component to access another component although it is not located on the same machine, i.e. the same address space. Of course the programmers can always take care of this for themselves by writing their own remote management code. But it makes more sense to define a system which offers an easy way to do this. Hiding the details of this semantic eliminates the risk of the programmer making mistakes writing network code. Writing such code is not as trivial as it seems, so it should be left to the experts.

6.2.4 Feature 4: Deployment in a Container

As we have described so far, software components should be “pluggable”, much like a LEGO brick. The question remains of what they should be plugged into. Components need an environment to run in, as they are not standalone programs. This environment is called a container. The process of putting a component inside a container is called deployment. As soon as a component is deployed, it has access to all the services offered by the container. Often the component comes with a so-called deployment descriptor, which describes the services required by the component. This could be an e-mail messaging system, for example. The big advantage of this system is that components get services for free without programmers having to worry about them. If you are the programmer of the component to be deployed inside a container offering e-mail services, you might consider sending an e-mail to the administrator in case you run into problems. For example, you might not be able to read from a required file. Normally, you would have to implement all code that is necessary for e-mail communication inside your component. With the container taking care of all this, you do not have to worry about that and can just make a call to the service offered by the container.

6.3 The Implementation of Components

After having defined the properties we require from components, the question arises of how those needs can be satisfied. We will take a look at every feature and provide an appropriate solution.

6.3.1 Feature 1: Self-sufficiency

This concept is well realized in object-oriented programming [Loud03]. Class definitions offer the means to encapsulate application functionality along with state. This allows one to define the required self-sufficient units of functionality. Objects also have a well-defined interface, which consists of their public methods and variables. There is no way to access any other part of the object from the outside or to see how a method is implemented.

6.3.2 Feature 2: Programming Language Independence

Making components independent of the programming language they were written in means that components should be in a binary format. Thus they do not have to be compiled or linked requiring a special programming language environment such as a specific compiler. The problem with binary formats is that they are specific to the computer architecture they run on. This is especially an issue if we think about developing distributed applications which run in a probably heterogeneous hardware landscape. There are several approaches to overcome this problem.

First of all, we could define a standardized format for binary formats. All platforms would have to support this format. However, this approach is not applicable. Not only are there already many incompatible formats in the market, which would make it hard to convice all vendors to agree on a single format, but formats also imply certain limitations. A 32 bit binary format could never make full use of a 64 bit machine. But this approach suffers from the overhead required for the translation process. Besides, a small machine does not have the necessary capabilities to emulate the complex operation of a huge mainframe.

Once again, the solution can be found by adding a level of indirection (like so often in computer science). Instead of compiling program source code into a specific binary format, a so-called byte code is generated. It describes the instructions of the program in a hardware-independent manner. Because this abstract byte code has to run on some hardware platform eventually, a way to map byte code to native CPU code is required. There are two ways to do this. First of all, one can program a virtual machine. This is a program that acts like an emulator and simulates a CPU that can read byte code as its native binary format. This principle is well known from Java, where the JVM reads Java class definitions and executes them. Another approach is to compile byte code into native binary code for the target architecture. These compilers are called just-in-time compilers. They read the byte code and output native executables.

Although these translation steps require extra overhead, we gain the benefit of being able to execute software in byte code independent of the architecture of the machine the software is supposed to run on. Thus, components should use this technology, especially for the development of Web applications, where distribution is an important issue.

6.3.3 Feature 3: Location Transparency

More problems have to be solved once software crosses host borders. There are many aspects, but we will just list some of the important ones: discovery, remote invocation, transportation of arguments, marshalling, error handling, and invocation semantics. We will not go into detail on what these problems are specifically. Let it suffice to say that there have been technologies developed to deal with all of them. It is important to point out, that components should not solve these problems individually, but should make use of one common solution. This is where the next feature comes in.

6.3.4 Feature 4: Deployment in a Container

As we have described before, containers allow components to make use of the services they offer. All the problems implied by location transparency can be solved by defining a set of remote services. These services can be used by every component deployed within the container. Thus components do not have to implement the complex algorithms to solve on their own, for example, the problem of invocation semantics. The process of in-

voking a component inside a container can be achieved by using linkable libraries; components can be plugged into and out of a container while it is running. Upon invocation, the byte code definition of the component is executed, either in a sandbox or by a just-in-time compiler. Components access services provided by the container using dynamic binding mechanisms, closely related to DLLs. Thus, this combination of approaches can be used to realize the concept of deployment.

6.4 Component Oriented Software in Practice – Middleware

After having examined the theoretical foundations of components, we will now take a look at concrete implementations. We have defined the realization of components, but we have not yet discussed the means necessary for components to communicate with each other.

Middleware is a layer of software that facilitates and manages the interaction between applications across heterogeneous computing platforms [Bern96]. It is the architectural solution to the problem of integrating a collection of servers and applications under a common service interface. Essentially, this means that middleware is software that connects applications (especially components), allowing them to exchange data. It offers several key advantages over hardwiring applications together, which typically entails adding code to all of the applications involved, and instructing them on the particulars of talking to each other. Middleware adds an independent third party to that transaction, a translator. Using established solutions for middleware over self-written code provides several benefits:

- Simplicity – all participants in an application scenario have to share one common interface, the one used for the middleware technology.
- Interoperability – software components from other vendors can easily cooperate with a programmer's project, if they both agree on the use of the same middleware.
- Hardware and implementation independence – when software components are called via middleware, one does not have to care about what programming language it is written in or what hardware it is running on. One just uses the services provided by the middleware framework and obtains the required results.
- Provision of services – when communicating, some tasks might be necessary which are not related directly to communication, but are still useful. For example, data might have to be checked for integrity, split into transportable packets, or you might need a global time service. If all these extras are provided by the middleware framework, the programmer does not have to worry about writing code for the functionalities and benefits from the standardized interface to these extra services.

Although this short description may look like providing a simple task, middleware has to solve a lot of problems. In the following sections, we will take a closer look at some middleware systems which satisfy these criteria.

6.5 The Classical Approach: RPC

A first solution to realize remote invocation was simple RPC (Remote Procedure Call) [RPC95]. It is typically used in a client/server context. Consider a program running on a client that wishes to call a procedure on the server. The first step is to define the signature for the procedure to be called (Fig. 6.1). These signatures are grouped together in an interface. The next step is to describe the interface and its signatures using IDL. IDL stands for the Interface Description Language and allows the programmer to describe what parameters are consumed by the function and what values are returned from it. This description can be understood as a specification of the service provided by the server.

The second step is to bind the IDL description with the code on the server and the client side. This process generates two pieces of code: A client stub and the server skeleton. Every interface which was defined in the IDL description results in the creation of a pair of corresponding client/server stubs. A client stub is a piece of code that has an identical signature to the procedure on the server side and is compiled and linked together with the client side code. From the client applications' point of view, calling the client stub is just a regular local function call. But the stub acts as a local proxy for the remote procedure on the server side. On invocation, the function takes the parameters, sends them to the server, and gets the remote procedure executed there. It receives the return value and sends it back to the local program on the client side. Thus, the stub makes the remote call appear as a regular local call, hiding all the necessary overhead from the programmer. The server stub works in the similar way as the client stub, but the other way round.

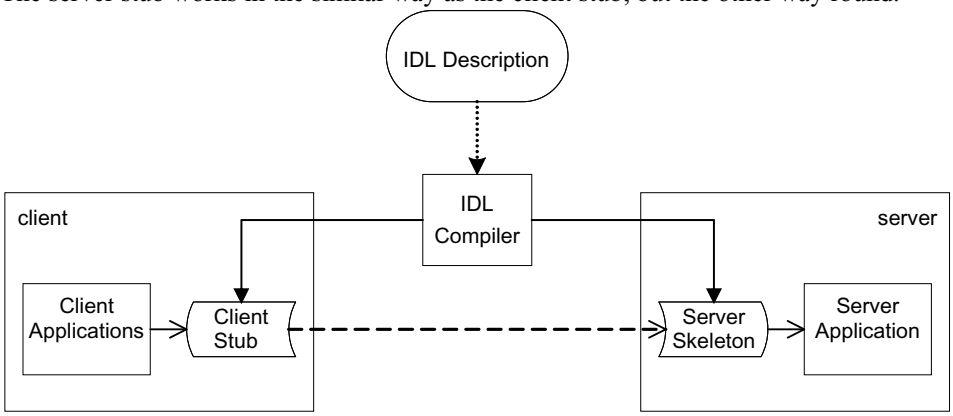


Fig. 6.1. An overview of RPC

6.6 Remote Method Invocation (RMI)

There was an effort to take RPC to the OOP level in the Java language, called RMI [RMI04]. The idea is basically the same as with RPC. The difference is that on the server machine a special process is required, the so-called RMI registry. As with RPC, all objects that are to be called by a client have to declare their interface to the outside world. This is done by registering the server object with the registry. Client objects can connect to the server and obtain a reference to a stub object. As in RPC, this stub object is a local

object on the client side and contains code to transport the parameters of a method invocation to the remote object on the server side. The method invocation is executed there and the return values are sent back to the client stub object, which returns the values to the caller. Although this technology allowed for the distribution of execution across host barriers, there were several drawbacks. For example, to invoke a method on an object, its interface has to be known a priori. It is not possible to discover the interface of an object upon discovery.

As time moved on, frameworks to realize RMI went beyond this basic interoperability and added more and more features that made it easier to develop distributed applications. These services were built into the framework and are callable, just like the auxiliary servers in the system providing this valuable service. These systems are called object brokers.

6.7 Object Brokers

Object brokers extend the RPC paradigm by providing a number of services that simplify the development of distributed applications. The goal is to hide most of the complexity of remote invocation by making them look like local calls.

The tricky part in creating such a system is that the invocation of a method is more complex than the call of a function, as there are concepts in OOP that are unknown in a pure functional environment. Due to notions like polymorphism and inheritance, it is necessary to know exactly what class an object belongs to, as this implies which definition of a method is to be invoked. This is something that classical RPC systems did not have to worry about. CORBA is the most commonly used object broker system.

6.8 CORBA

CORBA stands for Common Object Request Broker Architecture and was developed by the Object Management Group (OMG) [CORB04] [OMG04]. It offers a standardized specification of an object broker rather than a concrete implementation. A CORBA-compliant system consists of three main parts (Fig. 6.2):

- The Object Request Broker (ORB) – it provides the basic object interoperability functions.
- CORBA services – a set of services, known collectively under the name of CORBA services [CSer04], is accessible through a standardized API and provides functionality commonly needed by most objects.
- CORBA facilities – a set of facilities, commonly known under the name of CORBA facilities [CFac04], provides higher level services needed by applications rather than by individual objects. Examples include document management, internationalization, and support for mobile agents.

We will take a close look at the ORB and describe how distributed functionality is realized in a CORBA environment. Then, we will take a brief look at some of the CORBA services, but not go into detail on CORBA facilities. For further information on those services not explained here, we refer to [OrHa98].

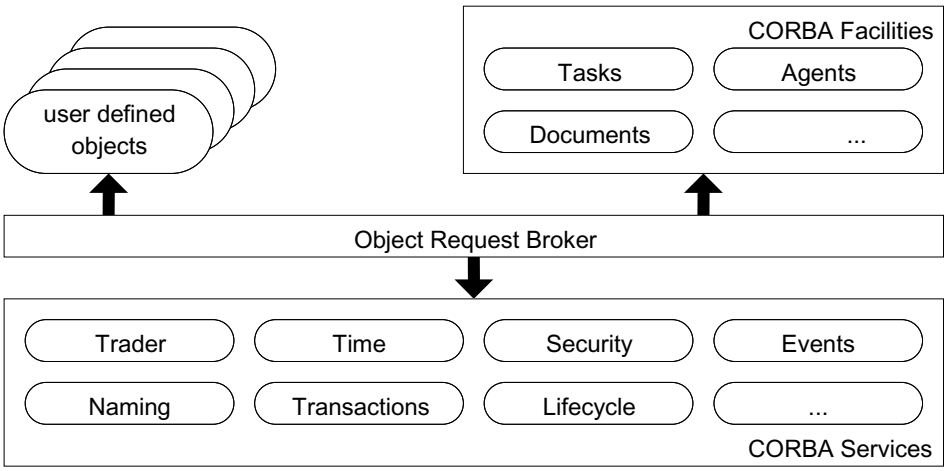


Fig. 6.2. The CORBA framework

6.8.1 How CORBA Works

As mentioned before, the ORB is the central element of the architecture. Every interaction between clients and services makes use of it, whether it is a call to a CORBA service, a CORBA facility, or a user-defined object (which could be a software component). It is the communication backbone of the whole framework.

As in RPC, before an object can be accessed through the ORB, it must declare its interface using an IDL, so that clients are aware of what methods the object provides. This process is very similar to the way interfaces are declared in RPC systems. But of course the language used to describe interfaces for objects is enriched with language elements that allow for the descriptions of notions like inheritance and polymorphism, compared to languages used for classical RPC [CIDL04]. Based on this interface description, stub objects can be generated. These objects work just the same way as in the RPC world: they hide the complex overhead necessary for remote invocation behind the call to a local object.

From a technical point of view, to develop a client object that interacts with a given server, all a programmer needs to know is the server's IDL interface. Of course the programmer must be aware of the semantics of the interface methods as well as of other constraints. For example, a specific order of execution of methods might be necessary to achieve a certain goal. If you want to access a database, for example, you first have to establish a connection to the database, then log on using a valid user Id/password combination, and then you can query the database. If you try to query data without having logged in, you will not get any results but an error message. These aspects are not formalized and are assumed to be described by other means, such as comments in the IDL, or by any other method of documenting the service provided.

But CORBA offers even more services to improve flexibility. The way we have described it so far, stubs are created before compile time. CORBA allows client applications to dynamically discover new objects, retrieve their interfaces, and construct

invocations to these objects on the fly, even if no stub has been previously generated and linked to the client. These capabilities are based on two components: the interface repository and the dynamic invocation interface. The interface repository stores IDL definitions for all objects known to the ORB. Applications can access the repository to browse IDL interfaces. The dynamic invocation interface provides operations that can be used by clients to browse the repository and dynamically construct method invocations based on the newly discovered interfaces.

But this does not solve the problem of dynamic invocation completely. The problem remains of how the client application is to know which IDL definition in the repository corresponds with the service it is looking for. There must be a way for the client to identify the required service.

In CORBA, there are two ways to do this: the naming service and the trader service. The naming service allows for the retrieval of object references based on the name of the service needed, e.g. `BuyStock()`. This approach is comparable to a regular phone book (also called white pages) where you can look up the phone number for the name of a certain person. Or you could compare this service to an Internet search engine, where you can type in the name of a company and get as a result the URL of the company's homepage. The trader service, on the other hand, allows clients to search for a service based on its properties. This requires services to advertise their properties with the trader. Different services can have different properties, describing non-functional characteristics of the service. This approach is comparable to Yellow Pages (Sect. 7.4.1). In such a directory, you can look up a certain line of business, e.g. carpenters, and get all the numbers that are associated with such a business. It is also comparable to the Semantic Web paradigm (Chap. 8). In a Semantic Web search engine, you could type in a request providing not concrete facts like names, but the desired capabilities. For example, you could search for companies that build operating systems for PC hardware and have a local branch office in your country.

Although this capability of CORBA to perform dynamic service selection and invocation is very intriguing, it is rarely used in practice. First of all, constructing dynamic invocations is in fact very difficult – not so much from a technical, but from a semantic point of view. Finding a service based on the feature list offered by the trader service has to face the same problem as the Semantic Web paradigm. To search for services, the client must understand the meaning of the service properties. This requires a common understanding of the ways to describe these properties between the service provider and the client object, which results in the requirement of a commonly understood ontology.

Another problem is the lack of description of non-functional knowledge about the service offered, e.g. the exact meaning of the parameters or the order in which certain methods should be called. This information is not included in the IDL description and therefore requires interpretation by the programmer of the client object. Hopefully, this interpretation leads to the same result as the ideas that the programmers of that object had in mind when they programmed the service.

6.8.2 Evaluation of CORBA

CORBA was the next logical step after RPC. It provided OOP programmers with a standardized way to develop distributed applications, allowing them smoothly to integrate their software projects with existing or bought infrastructure. There are many ORBs offered today by various providers including ORBIX from Iona [Orbi04], VisiBroker from

Borland [Visi04], or the Open Source OpenORB Project [OOrb04]. They all offer basic ORB functionality but differ in the variety of built-in services.

From a component-oriented point of view, CORBA was the first important system that realized the idea of component-oriented software. It fulfilled all the requirements we defined for components: self-containment, programming language independence, and location transparency.

Services offered in CORBA are self-contained. From a developer's point of view, invoking a remote method is all that has to be done to make use of a service. Of course the service may consist of a whole collection of classes and structures, but this is hidden from the developer and therefore the process of calling a service seems closed to the developer.

Programming language independence is achieved by the variety of IDL compilers. From the programmers' point of view, it does not make any difference if the service they call was originally written in C++, Java, Eiffel, C, or any other language. All they see is the IDL description, which is language independent. The languages do not even have to be object oriented, as can be seen from the IDL compiler for languages like C. This is true not only for the server side, but also for the client side. Therefore, regardless of the language the developers write in, they can make use of any service offered by the CORBA framework. Thus is it possible, for example, that a C program makes use of a service that was written in Java.

Also, location transparency is provided by the ORB. For the client application, all the necessary overhead for remote invocation is hidden behind a local method invocation to the stub method. The ORB takes care of passing all the necessary data to the server object and returning the result value. Finally, the ORB represents a concept closely related to a container. Registering an object with a CORBA registry can be compared to deploying the object in a CORBA container. All this makes the CORBA middleware system a component-oriented framework according to our understanding. It is possible to create and use software components facilitating features provided by CORBA.

6.9 Sun's Enterprise Java Beans (J2EE)

The J2EE platform [J2EE04] supports the development of enterprise applications with multiple tiers. It is called a platform because it combines three technologies: components, services, and communication. We have already introduced the concept of a software component. Services correspond to the auxiliary services provided by the container or the middleware framework. Finally, communication refers to providing easy means of communication between parts of the applications, much like an ORB does in CORBA.

6.9.1 Architecture of J2EE Web Applications

To give the reader an idea of how the parts of the framework cooperate, we will explain how the layers of the four-tier architecture are mapped onto J2EE server components (Fig. 6.3).

On the client machine, there is a regular Web browser running. In J2EE nomenclature, the Web tier and the application server are combined into a so-called J2EE server. It consists of at least two containers, a Web container and an EJB container. The Web server tier, as defined in Chap. 3, corresponds to the Web container of the J2EE server. The application server tier corresponds to the EJB container. The software components

that reside inside the application server are called Enterprise Java Beans (EJB). Finally, there may be back-end systems like a database. We will now discuss these architectural elements step by step.

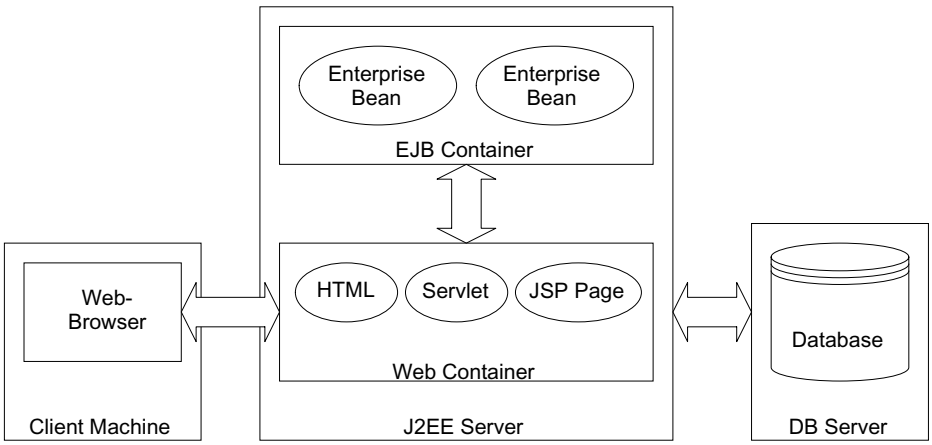


Fig. 6.3. The J2EE framework

The Web container receives requests from the clients and processes them. The Web container can perceive from looking at the requested URL what kind of document is requested. It manages the software components lifecycle, as it is responsible for the creation and deletion of the EJB. It dispatches service requests by mapping them into method invocations for the EJB residing in the EJB container. And finally, it provides standard interfaces to context data, such as session state or information about the current request, which allows the EJB to query the Web container about cookies or which type of browser the request was initiated by. The Web container can handle three kinds of documents: static HTML files, Java servlets, and JSP.

Static HTML files are the simplest case. They merely have to be read from the file system and be delivered to the client via HTTP. The Web container can also deliver Java servlets. As mentioned in Sect. 5.6.2, Java servlets are Java classes that generate as an output the desired HTML page. Whenever a servlet is requested, an instance is generated, the invocation parameters are passed to it, and its output is sent back to the requester. Finally, the Web container also supports JSP. Upon request, the instructions inside the JSP are evaluated and the result is delivered back to the requester.

6.9.2 EJB

We will now move on to investigate the EJB container, the environment for the software components inside a J2EE server. These components are called EJB and represent tokens of application-specific business logic. The container provides important services like lifecycle management, transaction management, security, concurrency, and many more. This allows the developer to focus on solving business problems, as the platform handles complex system-level issues.

Before going into detail, let us think for a moment about what the designers of the J2EE platform had in mind when they defined the EJB. They are supposed to resemble

business objects and business logic. The idea behind these concepts is to look at a business the way you look at an application in an object-oriented way. Just as an application is made up from a set of cooperating programming language objects, so a business is made up from a set of cooperating business objects. In an application, these objects might be lists, arrays, and queues, whereas in a business we talk about employees, managers, divisions and customers. All these business objects have to cooperate in a certain way, called business logic.

The goal of the developers of the J2EE platform was to provide a system where a developer can easily recreate this scenario. To do so, they isolated certain aspects of the business objects:

- Retention of state – business objects often need to maintain their state. This state can be either conversational or persistent.
- Concurrent access on data – as business objects often operate on the same data, there should be a way to realize this concurrent access without causing anomalies.
- Transactional behavior – business objects require transactional behavior. This term describes the principle of "all or nothing". Either the whole action takes place or there are means to change the world back to the state at the beginning of the action with all traces of the half-executed action being treated.
- Scalability – a business object should have the capability to provide its services to a large number of clients simultaneously. This implies the necessity for an algorithm to give each client the impression that it is served by an individual dedicated object. On the other hand, there should not necessarily be a new instance of a business object created for every client request.
- Access control – business processes often realize the concept of identity which allows the definition of roles and users. This in turn allows an access control system to be established, where interactions with certain business objects can be restricted by an access control directive.

As we have shown, business objects need to provide some generic services to clients. Examples of these services include security issues, remote access, and support for transactional behavior. These requirements are complex and beyond the domain of business logic required to implement an application. To support the programmer in developing enterprise applications, a standardized infrastructure on the server side is needed. Thus the programmer can focus on business logic issues.

The J2EE architecture provides such a solution in the form of EJB in the bean container. Experts provide a framework for delivering this system-level service so that application domain experts can concentrate on solving the problems of the enterprise instead of expending their efforts on system-level issues. In the J2EE architecture, these components are called Enterprise Beans. It seems hard to find a solution that meets all the requirements equally well. That is why the designers of the EJB specification decided to define three different kinds of implementations for business objects: session beans, entity beans, and message-driven beans.

Session beans are intended to be private resources used only by the client that creates them. Normally, their lifecycle coincides with the lifecycle of the session between a client and a server, which is where their name is derived from.

Entity beans are an object-oriented representation of some entities that are stored in persistent storage, such as a database. Compared to session beans, every entity bean can

be uniquely identified by a primary key. The lifecycle of an entity bean is independent of the duration of a session between the client and server.

Message-driven beans are a mechanism to process asynchronous messages. We will not describe them too deeply here but focus on the first two kinds of beans. We will discuss the specifics of these three flavors of beans in more detail later on. Firstly, we will take a look at their similarities.

As we have mentioned before, beans live inside a bean container. This container provides many important and useful services to the beans and manages invocations from the outside to the bean. Although it seems like the call is passed directly to the bean, the container must intercept the call and provide useful additional functionality transparently behind the scenes.

6.9.3 The Three Kinds of Beans

We will now take a specific look at the three kinds of beans. An entity bean represents an object view of business data stored in persistent storage or an existing application. The bean provides an object wrapper around the data to simplify access and manipulation of that data (much like the helper classes in Java that wrap primitive data types). An entity bean allows shared access from multiple clients and lives beyond the duration of the session between the client and server. If the state of an entity bean is updated by a transaction at the time of a server crash, the state is automatically reset to the state of the last committed transaction. Thus entity beans show the transactional behavior we required business objects to have.

The protocol for transferring the state of an entity bean between the bean's instance and the underlying persistent storage is called the object persistence. The are two possible ways to realize this protocol: bean-managed persistence and container-managed persistence.

When using bean-managed persistence, the programmer also writes the database access calls into the code of the bean. This may cause problems when trying to adapt the entity component to work with a database that is using a different schema or that is of a different type (relational vs. object relational). Encapsulating these calls into data access objects makes it easier to adapt to the requirements of these changes, but still requires regeneration of the data access objects in the case of a change. Thus, container-managed persistence should be used whenever possible.

When using container-managed persistence, the programmer relies on the container to manage access to the database. Thus the programmer does not have to write any code concerned with database access. This simplifies tremendously the task of writing entity beans because the container takes responsibility for the tedious job of generating the database relevant code. Using this approach accelerates the development of applications by relieving the programmer of this task.

However, in some cases, using bean-managed persistence is unavoidable. If the application is very performance sensitive, then it makes sense to use fine-tuned database access code that is handcrafted and provides maximum performance. It is also unavoidable if the persistent storage is not supported by the container; for example, it is not a database but some other kind of archive that can only be accessed by proprietary code. If the container has no notion of accessing this archive, the programmer must provide instructions on how to retrieve and store information from or in this storage system.

Session beans are used to implement business objects that hold client-specific business logic. The state of such a business logic represents the interaction with a specific cli-

ent and is not intended for access by other clients. A session bean typically executes on behalf of a single client and cannot be shared among multiple clients. A session bean is a logical extension of the client program that runs on the server and contains information specific to the client. In contrast to entity beans, session beans do not directly represent shared data in the database, although they can access and update such data in persistent storage. The state of a session object is non-persistent and does not have to be written to the database. The J2EE architecture specifies two flavors of session beans: stateful and stateless.

A stateful session bean stores the conversational state on behalf of its client. This state is defined as the bean's fields values and all objects reachable from the bean's fields. Stateful beans do not directly represent data in a persistent data store, but they can access and update data on behalf of the client. The lifetime of a stateful session bean typically is that of the client.

A shopping cart is a good example of the use of a stateful session bean. The content of the cart is specific to a particular customer and does not have to be saved unless the customer is willing to place the order and thus purchase all the items in the cart. So the shopping cart object lives as long as the ordering session by the client. The data should not be shared, since it represents a specific interaction with a specific customer and is alive only for the customer's session with the server.

Stateless session beans, on the other hand, are designed strictly to provide server side behavior. The term stateless means that the session beans do not maintain any state information for a specific client. This implies that all stateless session bean instances are equivalent when they are not involved in serving a client-invoked method. Thus they are ideal to create reusable service objects. They also have the benefit of providing high performance. As they are not bound to a specific client, the server can generate a pool of these objects and use one of these instances whenever a client issues a call. Thus the server only has to keep as many instances of these beans as are used at the same time, not overall, as it would have to do with stateful session beans. This minimizes the resources needed. A bean that searches a product catalogue for information on a specific product is a good example of the use of a stateless session bean. It is invoked with a product ID, retrieves the desired information and returns it to the client. Then this bean is ready to serve another request by another client.

Message-driven beans are the latest addition to the family of beans. They allow applications to receive messages asynchronously. These messages allow components to communicate with other pieces of software by exchanging messages in such a way that the senders are independent of the receivers. The client sends its message and does not have to wait for the receiver to receive and process the message. Thus message-driven beans receive inbound messages from the Java Messaging Service that takes care of the technical realization of the communication, allowing for independence from a specific communication protocol.

From a programmer's point of view, message-driven beans behave much like stateless session beans, but are simpler. The bean typically examines the message and executes the actions necessary to process it. This may of course result in the invocation of other components. Like session beans, message-driven beans may be used to drive workflow processes. In this case, however, it is the arrival of a special message that causes the workflow to be started, not the initialization of a session with a client. A shopping system which processes e-mails that contain orders is a good example of the use of message-driven beans. If an e-mail containing an order in a well-known format arrives at a shop, it

can be passed on to a message-driven bean which interacts with other business objects in the server to place the order, just as if a customer had initiated a session with the server.

6.9.4 Evaluation of J2EE

Combining all the technologies provided by the J2EE framework gives you the power to create mature well-engineered Web applications. To realize the user interface, servlets and JSP can be used. As these are active elements, they can dynamically react to program flow and focus on presentation. The EJB allow the programmer to structure the application in the form of well-defined and reusable business objects. Easy access to persistent storage is provided by entity beans. Thus the J2EE framework provides you with all you need to create good applications.

Of course there will always be special domain-specific requirements. But J2EE provides a good structure to create the backbone of an application, not having to worry about typical common issues such as scalability, database connectivity, lifecycle management, etc. Further requirements can easily be added to an application later on, as the Java concepts of extensibility and modularization are completely available.

6.10 The Microsoft .NET Framework

There is a second framework available for the development of Web applications, offered by Microsoft: the .NET framework. It is a collection of various technologies which "enables the creation and use of XML-based applications, processes, and Websites as services that share and combine information and functionality with each other by design, on any platform or smart device, to provide tailored solutions for organizations and individual people" [Net04]. Before we go into detail on how .NET works, we will have to take a look at its historical predecessors.

6.10.1 (D)COM(+)

In the beginning, there was COM [Kirt98]. This acronym stands for "Component Object Model" and describes the way Microsoft manages software components. It allows programmer to develop a piece of software and register it along with an interface description, much like the RPC IDL. After registering this new component, it is possible to refer to it by a global unique ID. Any process on the local machine can obtain a reference to the component, create an instance of it, and invoke a method on that instance. Everyone who has ever inserted an Excel spreadsheet into a WinWord document has used this mechanism. The embedded spreadsheet is an instance of a register component. That is why you can make use of all the Excel features inside the WinWord document, because at that moment you are talking to an instance of the Excel component.

COM works fine, but has a big drawback: it works only locally. Invocations across host borders are not possible using COM. So Microsoft extended COM to become DCOM, "Distributed Component Object Model" [DCOM04]. The mechanisms used are very similar to the ones used in RPC. Stub components are created on the client and server side to hide the communication from the invoking party. Loosely speaking, DCOM is the network protocol through which COM components can interoperate within a network. With all these capabilities, DCOM could be called a middleware system.

But there were certain important services missing that were requested by programmers, e.g. support for transactions, persistence, and asynchronous messaging. So Micro-

soft provided the "Microsoft Transaction Server" and "Microsoft Message Queuing". The MTS can be referred to as a container for COM components. It contains an object broker, provides means for live cycle control and security, and also is a TP monitor. MMQ offered asynchronous communication mechanisms for COM components and copes with all the problems that might arise from this semantic.

With all these extensions, it became obvious that the COM architecture had to be extended further in order to support all the new technologies. The component model was redesigned to support features like just-in-time activation, early deactivation, object pooling, load balancing, and many more. This new schema was called COM+ [COM+04].

To sum up, the current version of (D)COM+ is a component model that allows one to define software components and their interfaces in such a way that they can be used remotely in an easy way. They benefit from the services provided by the container they live in. The only drawback is that this framework is currently only available in the Windows world.

6.10.2 Components of .NET

The .NET framework consists of several packages, not all of which are necessary for the development of Web applications. We will mention only those necessary in this context and not talk about the Common Language Runtime or C#. To develop Web applications using .NET technology, the programmer can use the services of ADO.NET, ASP.NET, and Web services.

ADO.NET contains all the classes necessary to realize access to data sources. It is the successor of Active Data Objects (ADO). It offers support for various kinds of database systems due to its flexible driver management, and even allows access to XML files.

ASP.NET is responsible for creating user interfaces in the form of active Web pages. Compared to the earlier ASP, ASP.NET has evolved into an object-oriented paradigm. It offers many widgets and tools to create impressive user interfaces, making it possible to create dialogs that look almost like local applications.

Finally, Web services could be described as the new generation of RPC. They allow global service invocation. As they are so important, the whole of the next chapter will be dedicated to them.

6.10.3 Architecture of .NET Web Applications

To understand how all these packages work together to create a Web application, we will take a look at a typical scenario (Fig. 6.4).

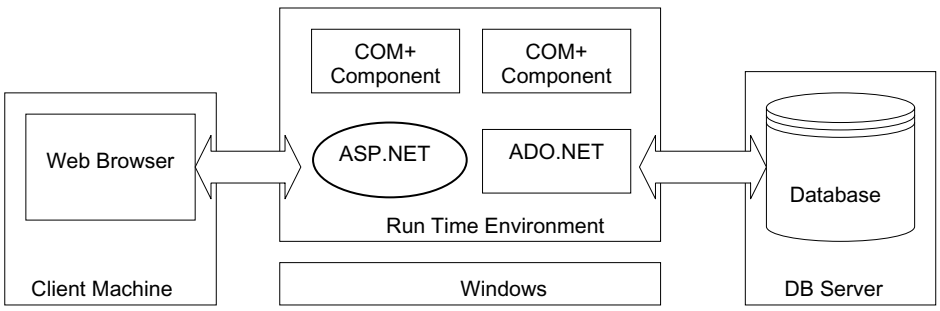


Fig. 6.4. Web applications with .NET

As with J2EE technology, active pages (in this case ASP.NET) are evaluated and then delivered to the Web browser on the client side. The active pages can make use of COM+ software components which are registered inside the run-time environment. ADO.NET allows components to access external data sources like database systems. This is only a short dip into the world of .NET, but the basic idea should be understandable. For the many other features it has to offer, we refer to the continuing literature.

6.11 CORBA Component Model

There is a third family of component software frameworks to be mentioned here, the CORBA Component Model (CCM). Like the original CORBA, it is a framework to support programming language independent distributed programming. But compared to its ancestor, which merely handles distributed objects, it adopts the concept of components and containers.

The CCM is a framework defined by the OMG. CCM was first introduced as part of the CORBA 3.0 specification. By definition [CCM02] the CCM is a set of concepts, notations, and programming interfaces for the design, implementation, packaging, deployment, and execution of distributed, heterogeneous, secure, transactional, scalable, CORBA component-based applications. Many of these properties can easily be realized using the container concept. We will not go into too much detail and give a precise definition of what CCM is, but will rather describe its extended capabilities by a comparison with the already introduced concepts of EJB and (D)COM(+).

One of the biggest benefits of CCM is the combination of a component-oriented container framework with the degrees of independence of CORBA. IDL descriptions of methods to be invoked can be connected to existing programming languages by using IDL compilers for the specific language. In CCM this idea is taken to the world of software components and containers. Thus, capabilities of a container are no longer accessible by using a programming specific method like a Java Method call, but are accessible exclusively by stubs generated by the corresponding IDL description. The byte code representation of an EJB, for example, is programming language independent as well. But as there is just one language available for producing such byte code (namely Java), it is not as flexible as CORBA and IDL. Using the CCM it is possible to write a CORBA component that runs inside a CCM container using the languages C, COBOL, or Pascal. Although these are no object-oriented programming languages, the level of abstraction introduced by IDL allows to specify components even in those languages. This independence could also be realized using EJB as wrappers for calls to binary implementations in any programming language on server side. But this approach would abuse the idea behind EJB.

If software is to be integrated that is written in an (almost) arbitrary programming language, using the CORBA IDL is the right choice. The wide range of languages that are supported by IDL compilers make it easy to integrate software of different origin.

CCM is very similar to the concept of EJB. However, there are some differences. They arise mainly because the CCM specification is newer and could respond to the challenges that have arisen recently.

- Language independence – CCM components are more language independent.

- More sophisticated component interfaces – CCM offers means to define several interfaces per component which allow to more exactly define the interaction between the component and its container.
- Introspection – CCM offers more sophisticated means to examine components during runtime.
- Deployment – CCM supports advanced deployment by offering better means to package components.

CCM is an interesting new approach for realizing distributed component-oriented programming. Its main achievement is the combination of the benefits of traditional CORBA programming using distributed, independent objects with the component–container concept. In the future it could become the standard for integrating existing software components, especially as it can easily be integrated with EJB.

6.12 When to Use What – the Dilemma

Now that we have introduced all these technologies to the reader, we want to provide a short guide on when to use them. Every approach has its own advantages and drawbacks and is therefore qualified for a specific scenario.

6.12.1 RPC

RPC is the oldest technology in use. It must be used if the programmer has to work in an pre-OOP environment. Either the clients only support pre-OOP languages or the service the developer wants to make use of is available only by RPC services and out of reach for reimplementation in an OOP manner.

An example of such a scenario might be an industrial installation like a printing press for newspapers. Client programs can query by RPC how much ink is left inside the machine. It is probably a hard task to obtain a Java environment for such an embedded system to realize an RMI, so you have to make use of the mechanism provided by the vendor. In general, you should not use RPC, if you can avoid it.

6.12.2 RMI

When programming in a Java environment, then RMI might be the right solution. It is suitable if one has a single centralized server that provides services that hardly ever change. Clients can just query the information they need by calling well-known objects on the server.

RMI is applicable in such a scenario: the interface hardly ever changes, there is one single server, and many clients. Client applications can run on different platforms, as the Java technology provides hardware independence. Many clients can query a single server object, as the RMI registry will take care of session management for each connection. Compared to a whole CORBA framework, the RMI registry is lightweight, as it merely manages the process of remote invocation and the session bindings.

One drawback of RMI already mentioned is that the interface of the remote objects has to be known a priori. Thus, if you change the interface on the server side, you have to distribute the new interface description to every client.

Thus, RMI is a good candidate for relatively simple client/server scenarios and if you are using Java as your language of choice. Furthermore, the RMI registry has just moderate performance requirements on the hosting server.

6.12.3 CORBA

The biggest benefits of CORBA are programming language independence and the services that come with the framework. A fundamental drawback of RMI is the requirement to run the Java-specific RMI registry on the server. With the CORBA ORB, the underlying architecture of the server is of no concern to the clients.

Imagine a central bank server that manages money transfers. This host is probably a high-performance mainframe system with special hardware and a specialized software environment, like a mainframe operating system. If the clients use CORBA to access the server, they now have to worry about those issues. Clients themselves can be written in any programming language and run on any architecture. It does not make any difference whether the clients' home banking software runs on an Intel CPU and a Windows operating system or on a SPARC CPU and a Solaris operating system.

Another advantage is the built-in services that come with the framework. In distributed applications, the synchronization of processes can be a problem. In a network of hosts, where every host has its own clock, it is hard to make sure that things happen in the right order. Instead of having to worry about how to synchronize all the machines in a network, a CORBA service can be used to set all nodes to roughly the same time.

On the other hand, CORBA is losing ground against new technologies like J2EE and Microsoft .NET. If you want to build applications with user interaction, these are the technologies of choice, as they provide better and easier means to model user interfaces.

This means that CORBA is always an appropriate solution in heterogeneous environments, especially if the developer wants to realize communication between parts of applications, e.g. to crosslink back-end servers for load balancing.

6.12.4 (D)COM(+)

The Microsoft protocol for communication between software components should no longer be used directly. Although it may be a powerful tool to realize integration of the various Microsoft products, it is too tedious to write an application from scratch just using this environment. If you write applications in the .NET framework, you will use this protocol without even realizing it. The complex semantics are hidden behind simpler programming language concepts so that the developer does not have to worry about them.

In summary, if you want to access internal components of Microsoft applications like Word or Excel, then (D)Com(+) is the right choice. But if you want to develop applications, than you should use either J2EE or the .NET framework.

6.12.5 J2EE

The J2EE framework focuses primarily on the development of thin-client applications, where the client consists only of a standard Web browser. Most of the activity happens on the server side. A typical scenario is a Web shop [PetS04]. The user interacts with the application by using dynamically generated HTML pages. The business logic of the application is coded on the server side by the appropriate means. The integration of back-end servers, like database systems, happens without the user noticing.

In conclusion, it can be said that the J2EE framework is the adequate choice for developing HTML-based Web applications.

6.12.6 .NET

The .NET framework has the same focus as the J2EE framework. With its means, it is possible to write applications that can be easily accessed and interact with the user via a standard Web browser. But the .NET framework has even more to offer. It comes with a whole collection of services and useful libraries.

First of all, there is the programming language C#, which comes with its run-time environment, the Common Language Runtime. The idea is basically the same as with Java: Instead of compiling source code into binary instructions, it is transferred into an intermediate language, which can then be processed by a run-time platform. This allows programs to run on various platforms. Secondly, there is ASP.NET. This technology allows for the dynamic generation of Web pages, much like JSP and servlets. And finally, there is substantial support for Web services. We will not go into too much detail here, as these will be explained in the next chapter.

You cannot compare J2EE and .NET directly, as J2EE is much more specific. But it does make sense to compare the .NET framework with J2EE including all the other Java technology. Then you might say that they both seem to be equally powerful. The Microsoft world has finally something to offer which provides equal benefits as the Java platform. As we have stated before, we will not participate in arguments about which technology is better. Suffice it to say that .NET is newer and therefore has better support and better integration for new technologies like Web services. But the support for this technology is also available in the J2EE framework and evolving. Time will show which technology will finally prove to be superior. Perhaps they will both find their niche, where they can benefit from their advantages over their competitor.

6.12.7 CCM

We mention CCM here just for reasons of completeness. There is currently an implementation called Open CCM [OCCM04]. It is currently in Version 0.7, meaning it has not yet reached a production state. Thus, the real impact of CCM cannot be evaluated yet.

6.13 Conclusion

In this chapter you have learned what software components are. Structuring software in a component-oriented way makes it easy to develop distributed applications. Thus, this programming paradigm is always the right choice when functionality has to spread across host borders. It allows the programmer to easily set the balance where execution takes place. In a classical client/server scenario, thin clients can be realized by putting only visualizing components on the client side and all computational functionality on the server side. Rich clients can be realized using almost the same components; they are just distributed in a different way, i.e. more on the client side. And to complete the picture, a server does not necessarily have to be a single host, but can be split into separate backend server farms that cooperate using component technology. This means that application designers can focus on structuring functionality and do not have to worry about partitioning. Segmenting the components onto clients and servers can be done afterwards.

The message of this chapter is that structuring an application in a component-oriented way is always beneficial, even if it might seem to cause unnecessary extra work at first. We will now move on to a new technology, called Web services, which is used for realizing the communication between software components.

7 Web Services and Web Applications

In this chapter the emerging Web service technology is discussed and the ways it complements traditional Web applications are explored. The chapter is organized as follows. In the next section Web services are motivated; the relationship between Web services and Web applications is explored; some definitions of Web services are discussed; and the components of the service-oriented architecture (SOA) are defined.

Sections 7.2, 7.3, and 7.4 define a basic set of standard technologies required to implement a Web service, namely WSDL, SOAP, and UDDI. Some advanced concepts (such as Web service security, transactions, and semantics) are discussed in Sect. 7.5. Section 7.6 is dedicated to composing Web services into higher order entities and describing Web service flow languages.

7.1 Introduction and Motivation

Web services are commonly viewed as a representative of middleware technology [ACKM03] [Newc02] [KaBu03]. On their emergence they were classified as a platform for distributed computing. Web services, however, may be utilized to complement Web applications. For this reason a slightly different motivation will be provided in this section. The discussion will attempt to show how and why Web services fit in the architecture of Web applications, while putting the broader middleware framework aside for a moment.

The field of application for Web services is the Web. In contrast to traditional enterprise computing technologies (Chap. 6), Web services are exclusively based on Web technologies. Software programs, for example, interact with a Web service functionality by exchanging messages realized as XML formatted documents and HTTP is typically chosen as transport protocol

Web services introduce RPC-based interactions to the Web, which is a new interaction style for the Web. Typical Web interactions are based on the exchange of HTML documents carrying presentation information. Web services introduce a way to carry out programmatic-type communication to the Web, combining both the document exchange and RPC styles.

7.1.1 Web Applications and Web Services

Enterprise applications are typically designed to have client/server architecture. The client part (usually a rich client) handles the GUI logic. The business logic is located on the server and is shared among multiple clients. The client applications perform RPC-style calls to invoke methods on the server. In an ideal case the public interface of the server part of a well-designed application would not only facilitate efficient communication with the clients, but also allow other applications to reuse it. With the increasing complexity of the applications nowadays and the steady trend towards providing services, reusability is becoming a factor of growing importance.

The majority of existing Web applications (Sect. 2.5) use HTML documents to format their user interface. As a result of the thin-client paradigm, the only pieces of information sent by the server part to the client are HTML documents, which encode the GUI

(Fig. 7.1, 1). While such an approach has numerous advantages, it also makes Web applications difficult to reuse in other applications (Fig. 7.1, 2). In other words, reusing or customizing an existing Web application for building third-party applications is becoming counterproductive. The reason is that HTML documents contain a mixture of presentation information (e.g. layout instructions) and real information, i.e. the data that is needed. Therefore, extracting the latter and providing it to the reusing Web application in an appropriate form is a major issue.

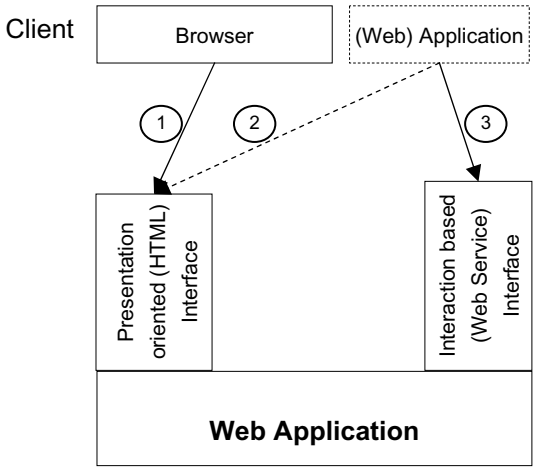


Fig. 7.1. Web applications and Web services

There are multiple solutions to this problem employing different approaches. For example, the Web application can be slightly modified to send the usable data in XML documents, which are passed by the reusing Web application. Alternatively a special HTML wrapper may be built (Fig. 7.1, 2). It will extract the information directly from the HTML documents (without changing the original Web applications) and provide it in a structured form to the rest of the client application. In this case, some serious problems related to the quality of the extracted data may arise.

An interaction-based interface, supported by the Web application, would greatly simplify the task (Fig. 7.1, 3), which allows an RPC-style interaction with the Web application's business logic. Based on this interface, third-party applications can directly use the business logic (avoiding the presentation layer) for purposes of building custom applications or compound applications (using the existing Web application as a source of components).

Web services solve the problem elegantly. Application designers can easily expose the desired part of the business logic and let other application developers use it as a distributed application. This approach leverages with existing technologies, requires less processing overhead, adds less platform modules, and simplifies the programming task.

7.1.2 Definitions

The term Web services has appeared a couple of times in the previous sections. But what are Web services? There are a number of competing definitions, varying from very sim-

ple to relatively complex, and having a different degree of precision. A relatively short definition, which can be found in [Mane01], defines Web services as "a unit of business, application, or system functionality that can be accessed over the Web". This definition, however, does not specify the type of interaction and how it is performed. In our opinion, it is too simple as it does not allow us to distinguish between a Web service and a CGI program – both are accessed over the Web (HTTP, URL, etc.) and both represent a unit of business functionality.

The W3C [WeSA03] offers this definition: "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." Its previous definition [WeSA03] insisted that a Web service must be "identified by a URI". This definition is taken as a working definition throughout the current chapter.

7.1.3 Service Oriented Architecture

The service-oriented architecture (SOA) offers a conceptual view of the Web service technology. In the next subsection we will firstly introduce SOA and briefly describe the way it functions. A description of how Web services implement SOA and what standards and technologies are involved will also be given in the following section. The SOA consists of three major parties (Fig. 7.2): service provider, service requester, and service registry.

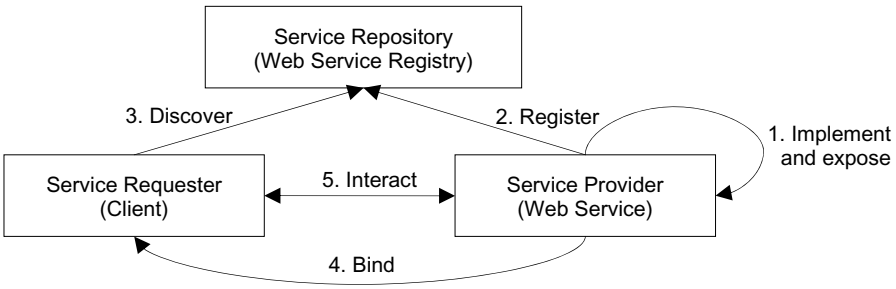


Fig. 7.2. SOA – a simplified view

The service provider is the party that implements concrete services with a predefined interface and exposes them to potential service requesters (Fig. 7.2, step 1). The service provider must register (publish) the services in a service repository after exposing them.

The service repository contains information about the interface implemented by the published services, the provider, and the service itself. Using this information, the service repository allows service requesters to search and "discover" exposed and available services (Fig. 7.2, step 3). The repository contains just a pointer (URI) to the services; they reside physically at the service provider.

The service requester is the third party in the SOA. Having first discovered a service a requester must bind it, i.e. establish the proper infrastructure and programming con-

structs, such as stubs for example (Fig. 7.2, step 4). Then the service requester can begin the actual interaction with the service by calling its operations (Fig. 7.2, step 5).

Generally speaking, the roles of a provider and requester are relative. Service requesters can expose their business logic (publish it in the service repository) and act as service providers in the context of a different interaction. In the most general case it can be assumed that all three parties are implemented as Web services.

The Web service paradigm is defined in terms of a set of standards. As will be further pointed out, the goal pursued with this standardization is twofold. On the one hand, it enables interoperability. If any implementation follows the set of standards then any service can talk to any other one. On the other hand, the standardization of the crucial parts of a service leads to the fact that implementation details of the Web services' business functionality can be almost completely discarded.

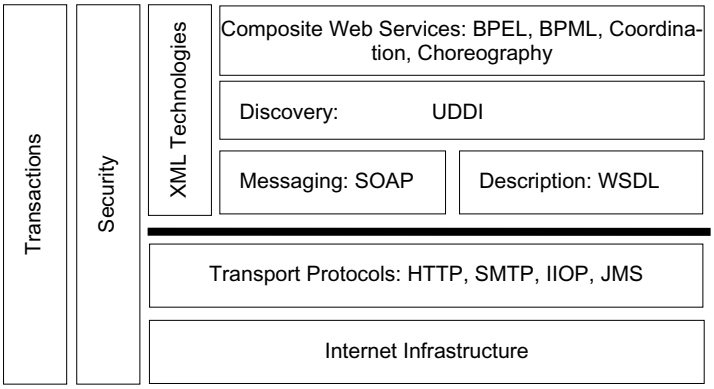


Fig. 7.3. Web service related technologies

The basic set of Web service standards comprises WSDL, SOAP, and UDDI. The Web Service Description Language (WSDL) [WSDL03] is the standard used to define the Web service interface. The Simple Object Access Protocol (SOAP) [SOAP00] is the standard for defining the format of the messages used within interactions. Finally, the standard for Universal Description, Discovery, and Integration (UDDI) of Web Services [UDDI03] is used to define service repositories (Fig. 7.3).

The combination of the standards represents a sufficient solution. However, the minimal solution can be reduced just to WSDL; SOAP and UDDI are highly desirable but still optional. While SOAP is important to assure interoperable communication on any Internet protocol to any device, it is not crucial since TCP/IP can also be used successfully in the case where a Java specific implementation is enforced as a business constraint. Actually, WSDL defines a special binding to SOAP. Similarly a motivation as to why UDDI may not be considered is absolutely necessary. Figure 7.3 presents a bigger picture of the Web service technology including advanced features. These will be introduced in the remainder of this chapter.

7.1.4 Why Use Web Services?

It is not obvious why Web services should be used. Web services are a new promising technology which has several competitive advantages over other established technologies.

- Web services enable interoperability. They comprise a set of standardized technologies. As long as service providers and clients follow them then interoperability is ensured. This is not the case with other technologies, e.g. component technologies.
- Web services leverage existing technologies. Web services encompass basically only the call infrastructure. How the Web service functionality is implemented is not a matter of concern. Therefore the business functionality many be implemented using almost any of the technologies discussed in the previous chapters (e.g. PERL script, EJB, COM components, simple Java class).
- Web services offer programmatic access to components of Web applications. Programmatic access in contrast to presentation document-based access in terms of HTML documents facilitates reuse. Therefore large parts of the Web application functionality can be efficiently reused in other applications. This approach leverages all the existing Web technologies.

Broad industry acceptance of the Web service technology, however, still depends on features such as powerful discovery or transactional support, which are still not available at a satisfactory level.

7.2 WSDL – Web Services Description Language

WSDL [WSDL03] is the standard used to define Web service interfaces. The definitions are XML documents, using the elements of the XML Infoset [XMLI01], [KeCh03]. WSDL defines a nomenclature of its own. Before we describe the concrete syntactical constructs in more detail, let us take a quick look at the big picture (Fig. 7.4).

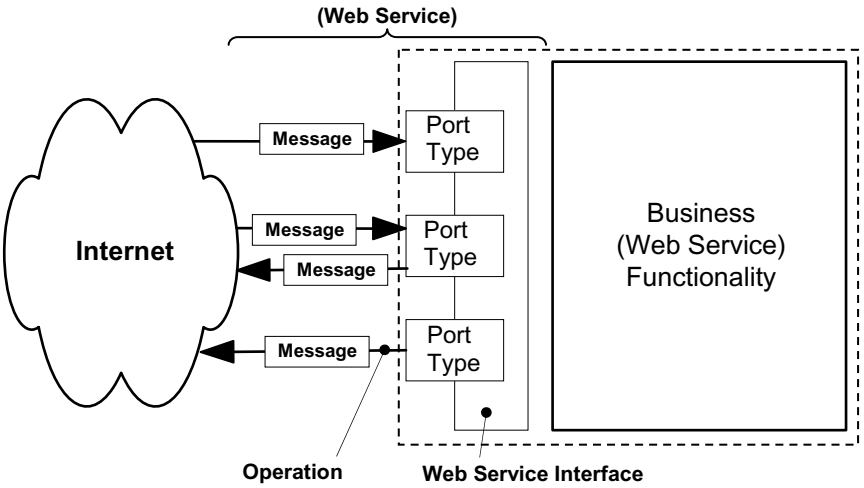


Fig. 7.4. Web service and terminology

From the client's point of view, a Web service can be compared to some sort of component. Therefore we are going to draw parallels with component terminology (Chap. 6). The readers need to always bear in mind that Web services do not involve business functionality (Fig. 7.4), i.e. the implementation of the Web service interface. A Web service has only one interface. The interface consists of one or more port types. Port types can be associated with the interfaces of components (Fig. 7.5). Each porttype contains a set of operations. An operation can be compared to the methods of an interface for a component. Each operation involves a number of messages. The operation call is realized as a request containing an input message with the actual parameters for the call. The second kind of message is an output message representing the response (usually the return value). A fault output message represents an exceptional case.

WSDL concept	**Component-specific concept**
Interface	Set of all implemented interfaces
Port type	Interface
Operation	Method
Message(s)	Method invocation-related parameters (e.g. formal parameters and return value) Method's faults (exceptions)

Fig. 7.5. Terminologies compared

7.2.1 The Structure of WSDL Documents

WSDL documents consist of two parts (Fig. 7.6): an abstract part and a concrete part. The abstract part contains interface elements' definitions independent of the concrete specifics of the underlying transport protocol or the provider. The concrete part defines how abstract part definitions, e.g. operations or messages, are mapped to the concrete transport protocols (e.g. SMTP, FTP, IIOP, JMS) or how exactly messages are serialized in XML. The goal is to separate the general Web service interface definition from the transport protocol specifics and to be able to define abstract service interfaces, which will then be implemented by many service providers and run over different transport protocols. Among other things this separation helps to achieve reuse of definitions in multiple Web services and to promote compatibility, because all services comply with the same abstract definitions.

Both parts may be split into different WSDL documents. Abstract part definitions document can be "imported" into many concrete part documents using the import statement (Fig. 7.6), which is quite a well-known technique in programming languages. Thus it can be ensured that different concrete Web services realize the same interface, which is an important feature when guaranteeing interface compatibility. By importing the abstract part definitions the importer actually references the abstract definition file, which helps to avoid inconsistencies once a change occurs. The abstract and the concrete definitions are registered in separate constructs in UDDI (Sect. 7.4)

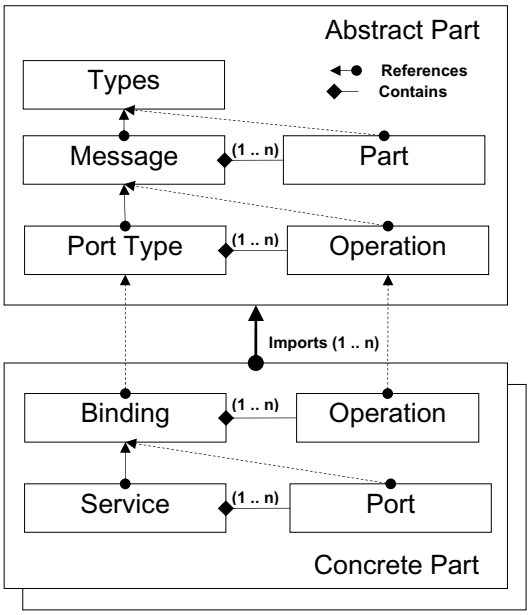


Fig. 7.6. WSDL document structure

7.2.2 Abstract Part Definitions

Abstract part definitions (Fig. 7.7) describe a Web service interface in a protocol-independent manner. An abstract interface definition defines one or more port types (Fig. 7.6), each having unique names. Port types comprise a set of operations. A porttype may extend other port types, with their operations being implicitly included in the extended (more) specific port type. This technique is known as inheritance in OOP.

Each abstract operation (Fig. 7.6) within a port type comprises (references) a set of messages. A message is treated as an `input`, `output`, or optionally as `fault` message. Depending on its type, each operation has up to two messages labeled as either `input` or `output`; optionally it may have many `fault` messages. `Input` messages represent the input parameters of an operation call. `Output` messages hold the return value of an operation and the `fault` messages stand for exceptional operation termination. In other words, the call, return, and error values the operation may return are defined as messages of different message kinds. For example, consider the operation `pendingOEntryList` in Fig. 7.7. It contains an input message `pendingOEntryList` representing the formal parameter list, and an output message `pendingOEntryListResponse` containing the return value.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
(1)   targetNamespace="http:// www.oe.de:8080/axis/services/OrdEntry"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:apachesoap="http://xml.apache.org/xml-soap"
      xmlns:impl="http://www.oe.de:8080/axis/services/OrdEntry"
      xmlns:intf="http://www.oe.de:8080/axis/services/OrdEntry"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Abstract Part Definitions

```
<wsdl:types/>
<wsdl:message name="pendingOEntryListResponse">
(2)   <wsdl:part name="pendingOEntryListReturn" type="soapenc:Array"/>
</wsdl:message>
<wsdl:message name="pendingOEntryListRequest">
    <wsdl:part name="in0" type="xsd:string"/>
    <wsdl:part name="in1" type="xsd:int"/>
</wsdl:message>
<wsdl:portType name="OrdEntry">
(3)  <wsdl:operation name="pendingOEntryList" parameterOrder="in0 in1">
(4)      <wsdl:input message="impl:pendingOEntryListRequest"/>
         <wsdl:output message="impl:pendingOEntryListResponse"/>
     </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="OrdEntrySoapBinding" type="impl:OrdEntry">
(5)  <wsdlsoap:binding style="rpc"
                       transport="http://schemas.xmlsoap.org/soap/http"/>
     <wsdl:operation name="pendingOEntryList">
         <wsdlsoap:operation soapAction="" style="rpc"/>
         <wsdl:input name="pendingOEntryListRequest">
             <wsdlsoap:body
(6)        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
           namespace="http://www.oe.de:8080/axis/services/OrdEntry"
           use="encoded"/>
          </wsdl:input>
          <wsdl:output name="pendingOEntryListResponse">
              <wsdlsoap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
           namespace="http://www.oe.de:8080/axis/services/OrdEntry"
           use="encoded"/>
          </wsdl:output>
     </wsdl:operation>
</wsdl:binding>
<wsdl:service name="OrdEntryService">
(7)  <wsdl:port binding="impl:OrdEntrySoapBinding" name="OrdEntry">
(8)      <wsdlsoap:address
             location="http://www.oe.de:8080/axis/services/OrdEntry"/>
     </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Concrete Part Definitions

Fig. 7.7. WSDL interface description

Generally speaking, four different types of operations can be distinguished:

- One-way (input or inbound operation) – a message is sent (an operation is called) and no reply is expected. Consider an operation setting the value of a variable.

- Request/response, the "normal" RPC-style interaction – clients send a call request; the Web service returns a return value as the response. Support of sessions and guarantee of reply must be supplied by the communication protocol.
- Solicit/response – the Web service sends a message (invokes a method) on a client and then the client invokes an operation on the service. This strategy is used to realize call-back functions. The WSDL specification does not specify SOAP for solicit/response operations.
- Notification (output or outbound operation) – a service sends a single message to a client (calling a method there), but no response is expected. This operation type allows the service to inform its clients of a change in state on the server side, for example.

Message definitions are independent of the operation definitions. This allows message definitions to be reused (referenced from different operations), which helps to avoid definition duplication. Each message comprises one or more arguments, called parts, having a corresponding (exactly one) data type.

Last but not least, a WSDL interface includes type definitions (Fig. 7.6). XML Schema-defined data types are used as base (default) set of data types in WSDL. Additionally, the WSDL specification allows complex data types to be defined very much like structures or arrays.

To recapitulate, all these definitions are abstract because they do not contain any concrete information about the service, the service's location, the transport protocol used, or the message's encoding. This information is specified in the concrete part.

7.2.3 Concrete Part Definitions

The two major definitions in the concrete part are binding and service (Fig. 7.6, Fig. 7.7). Bindings define what transport protocol is used to access a service of an abstract port-type, how the operation calls are performed and how the messages required for each operation are encoded. A service definition specifies one or more ports, i.e. URIs, where the service can be accessed and its operations called.

7.2.3.1 Bindings

Each port type can be implemented by several bindings. OOP gives a comprehensive example of this issue. An abstract data type or simply an interface must be implemented concretely by a class in order to be able to call methods. Each port type is like an abstract data type. In order to call the operations it defines, it needs to be "implemented" by a binding. In other words, transport protocols, encoding schemata, calling styles, and other parameters need to be specified.

Each binding contains a set of concrete operations. In addition, the WSDL definition makes use of the SOAP bindings. A SOAP binding defines which SOAP protocol format will be used, the concrete transport protocol, and the default style for each operation (Fig. 7.7, 5).

Every concrete operation (in a binding) corresponds to exactly one port type operation. Depending on the style, concrete operations define different sets of elements for each message kind (input, output, fault) implemented by the abstract operation. For an RPC style (Fig. 7.10), this would be:

- The namespace of the abstract (operation) definition – the namespace attribute.
- If the message is encoded in its transport protocol (e.g. HTTP) representation – the use attribute.
- And according to what encoding schema (i.e. how) it is encoded – the encoding style attribute.

Style is a property of bindings and concrete operations. It is of particular practical interest, because it defines the interaction/invocation style. WSDL specifies two types of interaction styles: RPC or document. In RPC style, the format of each message is strictly defined. An RPC-style interaction requires the use of SOAP RPC representation. In document-style interactions, the message format is not prescribed by the interface WSDL definition. Although SOAP is the common XML-based message format, document-style Web service interactions are not required to use SOAP formatted documents. In addition, it is broadly assumed that RPC-style interactions are synchronous whereas document-style interactions represent asynchronous communication.

Although RPC style is widely supported and leverages better the existing enterprise computing technologies, document style offers a higher degree of flexibility. The RPC style is very useful to leverage existing distributed applications, and shift the focus to Web services. In contrast the document style fosters the use of Web services in the field of electronic collaboration where the different parties exchange documents. For example, the order entry system of an automobile product may be configured to accept the order of a client formatted as a plain unstructured text document written by the respective dealer. Some of the reasons for this are:

- Documents have a more flexible format – it is easier to specify optional or new elements. The messages in RPC-style interactions exhibit a much firmer format. Therefore document-based Web services can evolve more easily.
- Validation and versioning of document schema – many document formats (the most famous example is XML) have document schema, defining the syntax of the document. The contents of the documents transmitted as SOAP messages can always be checked for validity against the schema. While versioning RPC-style messages is an issue, versioning documents and document schemata is a much more fault-tolerant solution.
- Granularity – Document-based interactions have proven to be efficient for high-level interactions. RPC-style interaction can become a bottleneck if low-level objects are exposed as services.

7.2.3.2 Services and Ports

Services and ports are the last missing piece in the WSDL puzzle. A service is a concrete service offered by a Web service provider. The service element (Fig. 7.6) is the actual "Web service". It is this service which is later registered at the service registry. The service contains (is an aggregate of) one or more ports.

A port is simply "an individual end-point for a binding". A port is associated with exactly one network location, i.e. a URI where the concrete operations defined by a binding can be accessed. One binding is associated with many ports; however, each port is associated with exactly one binding. Remember that one abstract port type is implemented by many bindings. The goal is to have an access point (port) for any binding and through the binding to any port type of the Web service interface. Ports related to the same port type but having different bindings or addresses must show semantically equivalent behavior.

It other words, the client may arbitrarily choose on which port it will communicate according to some criterion such as availability or communication performance. For example, the same Web service may be replicated on different servers. Each of them will have a different URL, which will correspond to a separate port. Another example is the use of different protocols. If the same Web service (abstract definitions) is implemented on different transport protocols (e.g. HTTP or SMTP) the separate binding will exist for each protocol. Consequently at least one port definition will exist for each of the bindings, where the location will be the location for the respective protocol – a URL and an e-mail address.

7.2.3.3 WSDL Overview

In this section we will summarize the key elements of WSDL interface definitions, which were discussed above. Figure 7.7 shows the structure of a WSDL interface definition document of the sample order entry Web service. WSDL documents start with an element definition (Fig. 7.7, 1) which contains the namespace definitions used in the rest of the document. The order entry Web service interface uses only the default XML Schema data types and does not define any composite types, therefore the element types is empty.

The first major element of the abstract part definitions is the message definitions (Fig. 7.7, 2). Two messages are defined: `pendingOEntryListRequest` and `pendingOEntryListResponse`. Each message has a set of arguments called WSDL message parts. So, for example, the message `pendingOEntryListRequest` has two parts: `in0` of XML Schema type string and `in1` of type integer.

The second major element of the WSDL abstract part definition is port type, which contains a set of operation elements. The order entry service has a port type called `OrdEntry` (Fig. 7.7, 3), defining one operation `pendingOEntryList`, which uses the above two messages as input and output messages respectively (Fig. 7.7, 4).

The concrete definitions are two major WSDL elements – binding and service. The order entry service contains one binding `OrdEntrySoapBinding` (Fig. 7.7, 5) implementing the port type `OrdEntry`. The `OrdEntrySoapBinding` is an RPC-style binding using HTTP as the transport protocol. A binding contains a list of operation elements (Fig. 7.7, 6) defining the concrete parameters of the implemented port type's operations. These parameters include style, message encoding, or preprocessing action (SOAP action).

The service element (Fig. 7.7, 7) defines a set of port elements (Fig. 7.7, 8), which point to a location (URL) where the Web service can be accessed. If multiple port elements are specified the respective binding can be accessed on any of them.

7.2.4 Lifecycle

In this section, we outline the canonical process (Fig. 7.8) or the classical sequence of steps for creating a Web service. In practice, however, there are multiple possible paths; this is why the process described here should be considered merely as "best practice".

Initially, the interface of the Web service must be defined. To do so, the designers create WSDL definitions comprising one or more WSDL documents. These are used at the provider's side to generate the Web service skeleton, on top of which the Web service business functionality is developed. To emphasize it, this is done by using tools such as WSDL2Java. WSDL interface documents are registered with (pointed by) the UDDI registry. When a service requester discovers the Web service, it retrieves the WSDL in-

terface definitions and uses them to generate the necessary stubs. This process is called Web service binding. At both sides (provider and requester) a number of platform modules, called SAOP infrastructure (Fig. 7.8), need to be installed and configured. In the most general case these modules consist of a SOAP router (also called an engine) running on an application server and a transport protocol server, for example an HTTP or an SMTP server.

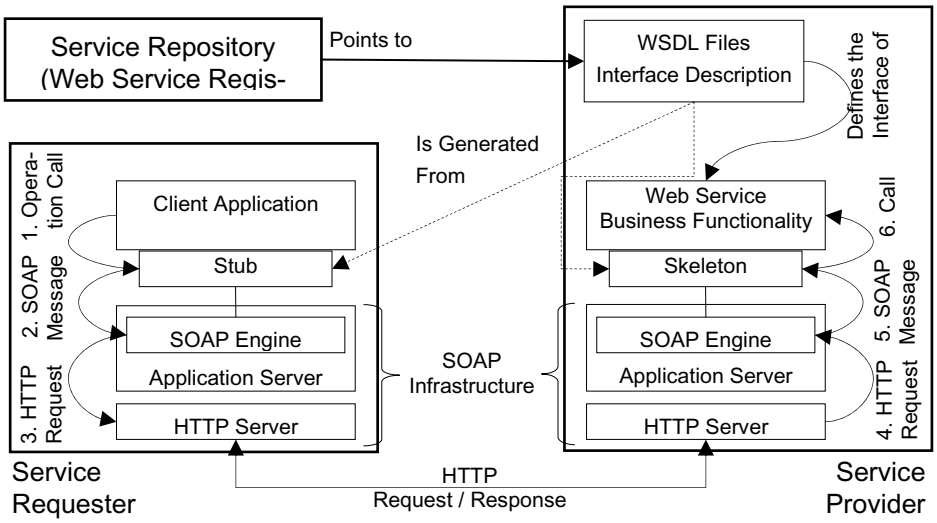


Fig. 7.8. A Web service scenario

While the stub and the skeleton cover the marshalling and demarshalling of SOAP messages (creating SOAP messages from operation calls and transforming SOAP messages back to calls), the SOAP infrastructure handles, among other things, transformation of messages to and from the target protocol message format. By target message format we mean the format of an HTTP request/response, if HTTP is specified as the transport protocol. Of course, beforehand SOAP message must be serialized into XML. The serialization rules are defined in documents referred to by encoding elements (as mentioned earlier). Steps 1 through 6 (Fig. 7.8) refer to the actions performed by the different modules involved in a Web service invocation.

7.3 SOAP – Simple Object Access Protocol

SOAP is the universal technology serving to transfer data messages in the realm of Web services. The first version of SOAP was proposed in 1999, and was entirely HTTP based. Since then, there have been a number of intermediary versions. Version 1.1 introduced the possibility to have bindings to different transport protocols (Version 1.0) such as SMTP or IIOP. The current version is 1.2 [SOAP00].

7.3.1 Why SOAP?

SOAP defines an interoperable way of transmitting messages in a system language- and protocol-independent manner. These properties are ensured by the extensive use of XML. All SOAP messages are serialized in XML. Effectively, SOAP messages can be processed by any system able to process XML documents. The fact that SOAP messages can be transferred over multiple Internet protocols leads to protocol independence. The way SOAP messages and protocol data are combined together is called binding. Technically the protocol independence is ensured by the fact that there are standardized bindings for a number of Internet protocols such as HTTP, SMTP, IIOP, or JMS. Theoretically a TCP/IP binding is also possible.

SOAP is not a self-sufficient protocol. The existence of a binding to a transport protocol is required to transmit a SOAP message. SOAP messages are self-describing due to their XML serialization. SOAP transmits messages without any knowledge of their semantics.

SOAP messages are said to be "one-way" transmissions. In other words, messages are regarded as simple packages of data sent either from the service requester to the service provider or vice versa. The notion of operation types (Sect. 7.2.2), i.e. the way of coupling SOAP messages in communication patterns, is defined at the interface definition level (WSDL). More complex communication patterns, called conversations, can be defined as well, but they are beyond the scope of this section.

SOAP is designed to be stateless. This means that the protocol neither provides constructs nor requires an infrastructure to record the current communication state (conversational state). In the general case, this influences the way Web services are invoked and interfaces are designed. No conversational state means no changes of the internal state of the Web service are preserved across successive operation calls. Therefore, all the data a Web service operation needs must be in the formal parameter list. There are vendor-specific APIs supporting the conversational state – for example, when stateful EJB are exported as Web services using HTTP binding.

7.3.2 SOAP Message Format

A SOAP message contains a SOAP envelope comprising optional SOAP headers, consisting of one or more header blocks, and a mandatory SOAP body, consisting of one or more body elements (Fig. 7.9).

The SOAP specification assumes that every message is sent by a service requester (sender) to a service provider (ultimate receiver) and processed by a number of intermediaries called nodes.

The intermediaries process information available in the message headers. The SOAP envelope encloses the SOAP message to be transported. A SOAP envelope element specifies the concrete encoding schema for the rest of the document (SOAP-ENV).

7.3.3 The SOAP Header

The SOAP header is optional and contains information orthogonal to the message body. For example, when a session-based connection between the service client and the Web service is opened, the header of each message will carry information to this session.

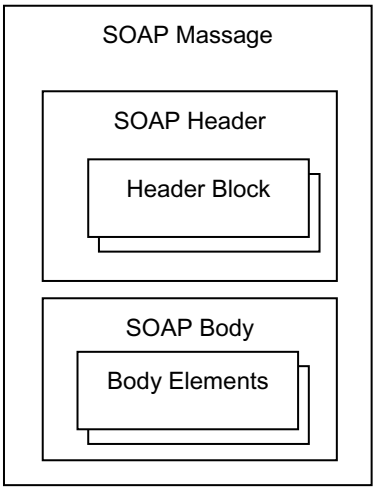


Fig. 7.9. Structure of a SOAP message

The SOAP header contains one or more header blocks. Every header block may be assigned a role indicating how the message has to be processed by the nodes. There are three types of roles: none, next, and ultimateReceiver. If a header block is assigned the role "none", then none of the receiving SOAP nodes must process the message. If the role is set to "next", then any node receiving the message can process it (including the "ultimateReceiver"). If a header block is assigned the "ultimateReceiver" role, the block is to be processed by the ultimate message recipient (the Web service). The way the message is forwarded to the intermediary nodes capable of performing the "next" role. The way the message is passed to, processed, and forwarded by the different nodes is called routing.

Assigning roles to the different header blocks defines how the soap message passes through the chain of nodes from the sender to the receiver. Another relevant question is how to indicate whether the message must be processed. This is done through the optional "mustUnderstand" attribute (Fig. 7.10), which is of type Boolean. If a header block contains the "mustUnderstand" and its value is set to "true", the respective node must process the message. If for some reason this cannot be done, a fault must be returned. Alternatively, if the value of the "mustUnderstand" attribute is false, the node may choose whether to process the message or not. Finally, the SOAP message body must be processed by the "ultimateReceiver".

7.3.4 The SOAP Body

The SOAP body contains the actual message. Messages are structured differently depending on the operation style. As mentioned in Sect. 7.2.3, there are two operation styles: document based and RPC based. In RPC-style operations, messages will include operation invocations (the operation's name and a list of factual parameters), return values, and operation errors (Fig. 7.10). RPC-style messages exhibit clear and predefined structure prescribed by the SOAP specification. Document-style operations are based on document exchange. They therefore have a structure, which is for the most part unde-

fined by the SOAP standard. RPC style must be used to expose efficiently existing client/server applications as Web services. If, however, an existing electronic collaboration application, e.g. an EDI-based application, is extended to use Web services then it is recommended to use document-style communication.

Another factor influencing the structure (syntax) of the SOAP is the concrete way SOAP messages are represented (serialized) in XML – called SOAP encoding. Concretely, the SOAP standard defines the ways parts of the SOAP message are represented in XML based on the "encodingStyle" attribute. It influences not only the way data types (e.g. integer, Boolean, and complex arrays) are represented, but also the concrete syntax of structure such as the SOAP header and any SOAP body subelement (but the fault element).

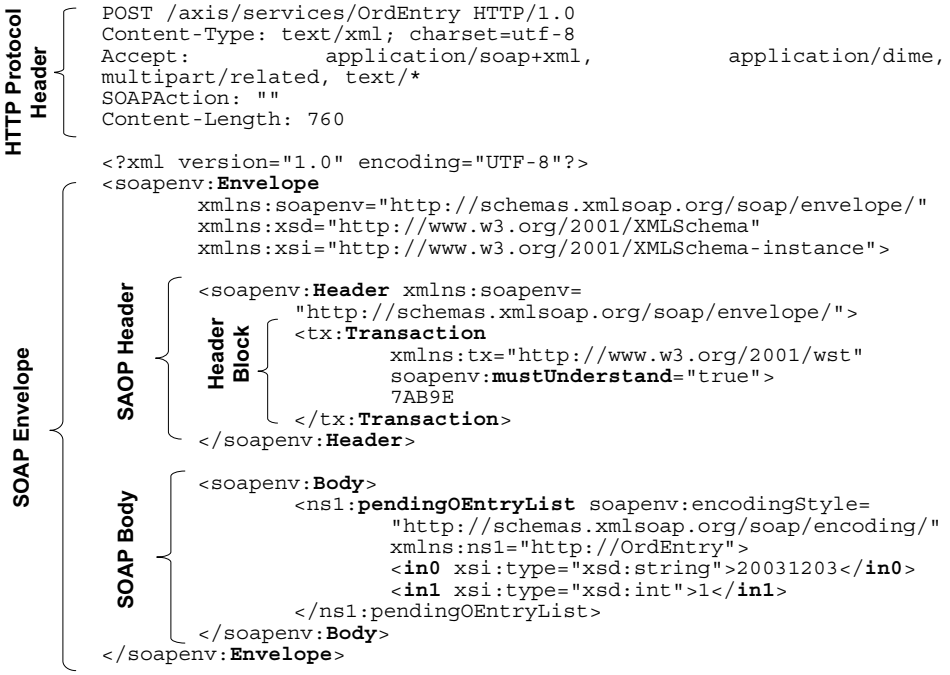


Fig. 7.10. Example of a SOAP message representing a request

The SOAP specification defines RPC-style interaction messages. These will be discussed in the rest of the section. Figure 7.10 shows a SOAP message representing an RPC-style invocation of the order entry Web service. The body contains an invocation element containing the operation name `pendingOEntryList` and the list of actual parameters (the pending date and the range). In this example there is also a header, carrying transaction information. Both the body and the header are represented as subelements of the SOAP envelope element.

The response will contain a header and a body. The body will be formatted as the operation `out` message containing the actual return values (or set of return values).

7.3.5 Binding to a Transport Protocol

SOAP messages can be transmitted over many transport protocols. The way the SOAP messages are mapped to the protocol-specific message format is called binding. The ability to define many mappings is the characteristic that ensures the ubiquity of SOAP as a protocol.

At the time of writing this book, there are bindings defined for the protocols HTTP, SMTP, IIOP, JMS, etc. The most popular and widely used binding is that to HTTP. Figure 7.10 is an example of it – the SOAP message is transmitted as an XML document included in a POST request. Although the use of POST requests is preferred, many SOAP engines tolerate the use of GET requests.

An example of a SOAP message embedded in an HTTP request is shown in Fig. 7.10. The XML serialized SOAP message is in the body of the HTTP POST request. If a GET request is used the invocation will be encoded in the URL. It may look for example like this: `http://www.orderentry-example.de/axis/services/OrdEntry?method=pendingOEntryList&in0=20031203&in1=1`. Using this invocation a user may invoke the operation `pendingOEntryList` on Web service `OrdEntry`, offered by the provider `http://www.orderentry-example.de/`, where the parameter `in0` has the value of 20031203 and the parameter `in1` has the value of 1.

7.4 UDDI – Universal Description, Discovery and Integration

UDDI [UDDI03] is the technology used to implement the service registry/repository. The first version of the specification appeared in September 2000 and was authored by IBM, Microsoft, and Ariba. The current version of the specification is 3.0.1 [UDDI03] which is standardized by the OASIS standardization committee.

The goal pursued with UDDI in the context of Web services is to serve as a technology enabling the discovery of all available Web services. Prior to being discovered, a newly exposed Web service must be published (Fig. 7.2), i.e. the necessary pieces of information must be put into the registry.

The UDDI registry is organized in a logically centralized, physically distributed manner. It consists of a set of UDDI registry nodes, which are synchronized with each other on a regular time basis (generally every 24 hours). The information for a service registered against one node is available on all other nodes after synchronization. There are many UDDI nodes: for example, the Microsoft UDDI node [Micr04] or the IBM UDDI node [IBMU04]. This is why we speak of a UDDI registry rather than a set of UDDI registry nodes. Such organization is typical of registries or directory services. It provides for robustness and extensibility.

The UDDI registry is itself accessible (exposed) as Web services. The UDDI interface (UDDI API) has a WSDL interface definition and is accessible over SOAP. The UDDI registry is itself registered as a Web service thus yielding self-description. Exposing the UDDI registry this way simplifies the interaction with the other nodes and makes access to it uniform (Web service coverage).

The scope of the UDDI registry is another relevant issue. Two types of UDDI registry are distinguished: public and private. All UDDI registry nodes are an example of a public registry. However, some enterprises might install a “private” UDDI node serving only their intra-enterprise services. In this case the synchronization mode, if any, needs to be defined.

The UDDI registry is quite often described in terms of its logical structure. The UDDI specification, however, defines a set of data structures and a data model. In the following the logical organization and the set of data structures will be discussed.

7.4.1 Organization of the UDDI Registry

The various types of information published in the UDDI registry about a Web service are logically organized into different categories called pages – just like a phone directory. The different pages are hierarchically organized. Generally speaking three kinds may be distinguished.

White pages are the top level of the hierarchy. These contain information about the service provider, the type of business, contact person information, and categorization. The general idea is that one company may provide many services. This is why many yellow pages can be associated with a white page. The data structure the white pages are related to is called a businessEntity.

Yellow pages are associated with the Web services a company provides. The contain an indirect description of what the service does in terms of taxonomies. A yellow page is associated with one or more green pages. The data structure a yellow page relates to is called a businessService. A yellow page groups concrete Web services with similar characteristics.

Green pages describe the way a concrete Web service can be invoked. Classification categories apply here as well. Green pages contain a URI referring to the interface description document. The data structure associated with the green pages is called a bindingTemplate.

The UDDI standard organizes the data physically in four data structures: businessEntity, businessService, bindingTemplate, and tModel. All of them are described below and are depicted in Fig. 7.11.

All UDDI data structures are assigned unique IDs. These represent an identification mechanism and are used as references. The UDDI keys follow the UUID (Universally Unique IDentifer) scheme firstly introduced in DCE RPC and used in many technologies subsequently.

The businessEntity contains information about the service provider such as its name, a short description, and a number of contact persons with their names and addresses. Any business organization may be identified by standard identification systems such as D-U-N-S (short for Data Universal Numbering System from Dun & Bradstreet [DUNS04]). These are referenced in the identifierBag construct. A business entity may also be categorized according to multiple schemata such as NAICS (North American Industry Classification System [NAIC03]) or UNSPSC (United Nation Standard Products and Services Code [UNSP04]). These are referenced in the categoryBag element. The goal is to provide additional information to the provider in terms of standard classification of its business, tax numbering, etc. All businessEntities, businessServices, bindingTemplates, and tModels may be assigned attributes from standard classification schemes. To do so they contain an element called categoryBag. The element businessServices is a collection of elements for all services implemented by the service provider represented by businessEntity. A businessEntity is uniquely identified by a businessKey.

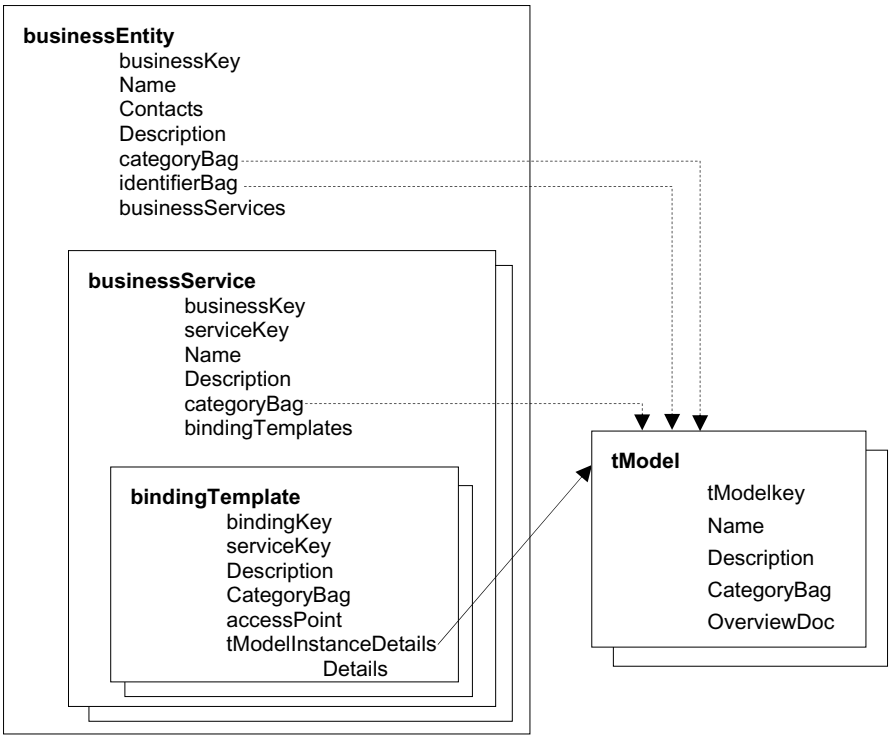


Fig. 7.11. UDDI registry data stuctures

The businessService element represents a Web service implemented by a service provider. A businessService actually contains general and descriptive information about a collection of concrete services offered by a provider. For example, the businessService element for the order entry service will contain data like name, description common to all concrete order entry services associated with (bound to) the transport protocol, and given concrete assess points. These concrete services are described by the bindingTemplate element. A businessService elemente is associated with the yellow pages from the logical organization. Each businessService element is assigned a unique serviceKey and may contain a reference (businessKey) to its businessEntity. Every businessService has a name and a description. It can also be classified under multiple categories and classification schemes. The IDs of each classification category are stored in the categoryBag element. The bindingTemplates element is a collection containing a set of the bindingTemplate elements. Each bindingTemplate is associated with a green page.

The bindingTemplate element represents the concrete part of a Web service. In other words, a bindingTemplate elemente relates roughly to the information contained in service and port elements of the WSDL interface definition (Fig. 7.7). A businessTemplate elemente contains a unique bindingKey and a serviceKey reference to the businessService. Each bindingTemplate element contains the URI where the service can be accessed, a reference to the WSDL document containing binding information, and further detailed

information. Additionally, the tInstanceModelInfo element contains a reference to the tModel.

The tModel is the fourth of the most essential UDDI data model structures. “tModel” is an abbreviation for technical model. Generally speaking, a tModel is “a reusable concept, such as a Web service type, a protocol used by Web services or a category system” [UDDI03]. The most essential role of tModels is to register the abstract part of a Web service interface definition as a global “Web service type” – independent of the respective provider. By registering abstract Web service interface definitions separately standardized interfaces can be defined – for example, a standard set of Web service interfaces for order entry systems. These can subsequently be implemented by a number of order entry systems whose services are registered in the UDDI registry as being of the type of the standard Web service interface.

The type–implementation relationship between the tModel and the bindingTemplate (the concrete service) is crucial to the integrity of the UDDI registry. Introducing types of services in the way the tModel does, would not be possible otherwise because the businessServices are nested in the businessEntites. A similar idea is already widely used in field component-oriented programming, where a component implements one or more interfaces, which are registered and treated separately. The use of tModels to express generic types can be extended. One can define tModels for classification or identification schemes. The goal is to make these definitions independent of a service, and reuse them while registering multiple Web services. Therefore the primary goal of tModels is said “to provide a reference system based on abstraction” [UDDI03].

The tModel definition contains an element called “overviewDoc”, whose value is a URI pointing to the actual service description, which is a document meant to be human readable. In the case of Web services, it must be the WSDL interface definition. However, since UDDI is not tailored exclusively to Web services, it can be any kind of document, e.g. ebXML or RosettaNet.

There are two UDDI data structures defined by the specification, which have not yet been described. They are called the publisherAssertion and the subscription. The publisherAssertion is used to define organizational structures (i.e. define relationships among) businessEntities in the UDDI registry. The subscription structure is used as a notification request when certain parts of the registry (for which the subscription is done) change.

Both UDDI data structures and WSDL interface definitions describe Web services in a complementary way. Therefore it is quite elegant to be able to establish some mapping between the two structures. Although a UDDI registry is independent of a concrete language or document format, there exist some recommendations (“best practices”) as to how this can be done [CuER02].

The idea is quite straightforward and depicted in Fig. 7.12. The concrete part definitions (Sect. 7.2.3) are mapped to businessService and bindingTemplate, whereas the abstract part definitions are mapped onto tModels.

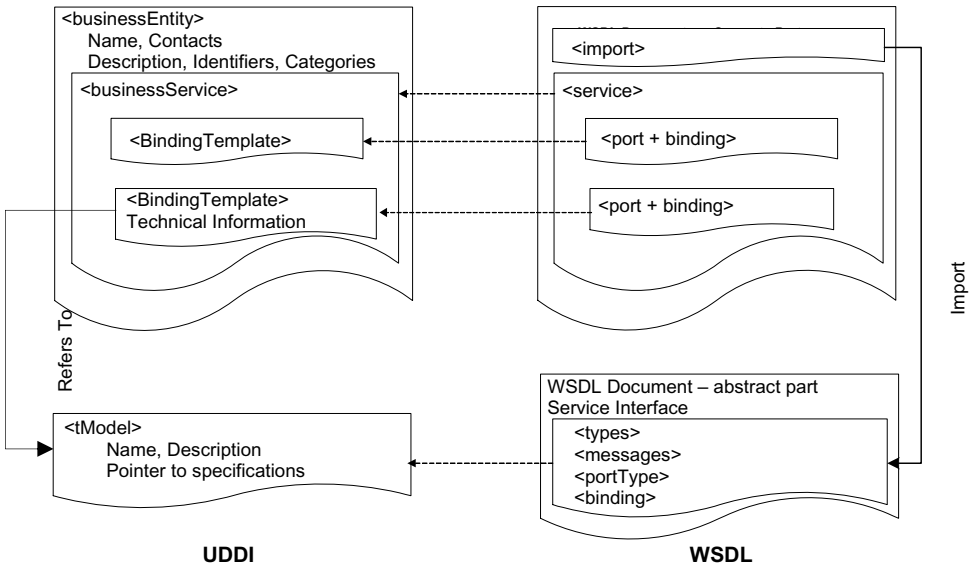


Fig. 7.12. Mapping WSDL document constructs to UDDI data structures

A URI to abstract WSDL definition is stored in the overviewDoc element of tModel. The service and binding WSDL elements are generally mapped to the bindingTemplate UDDI structure. The service access address as specified in the port element is stored in the bindingTemplate's accessPoint subelement.

7.4.2 UDDI APIs

The UDDI registry is accessible via a set of operations covering various functional areas such as publishing, querying, replication, etc. They are commonly referred to as UDDI APIs. The current version of the UDDI specification distinguishes six UDDI APIs (Fig. 7.13):

- Inquiry API
- Publication API
- Security API
- Custody and Ownership Transfer API
- Subscription API
- Replication API.

All these APIs (Fig. 7.13) have different WSDL interface definition files and are registered in the UDDI registry with different tModels. From the users' point of view, the Inquiry and the Publication APIs are the most import ones.

The UDDI Inquiry API provides operations facilitating the search inside a UDDI registry. This is the API used by the standard UDDI registry browsers or custom UDDI registry search tools. It involves two distinct subsets of operations: for posting queries and for retrieving information. Programmers can query UDDI registries using operations such as find_business(), find_service(), find_binding(), find_tmodel(). Once the result set is enumerated, the programmer can retrieve the information about each entry using the

retrieval subset of operations. It includes the operations get_businessDetail(), get_serviceDetail(), get_bindingDetail(), and get_tModelDetail().

The UDDI Publication API offers a set of operations enabling the manipulation of UDDI registry entries (create, alter, delete). On the one hand, the Publication API offers operations for creating or altering UDDI entries such as save_business(), save_service(), save_binding(), or save_tModel(). On the other hand, it provides operations to delete entries such as delete_business(), delete_service(), delete_binding(), or delete_tModel().

One of the disadvantages of the UDDI registry's Inquiry and Publication APIs affecting the capabilities for search and discovery is the fact that the UDDI standard does not provide possibilities to specify custom characteristics/properties for the published entities. Therefore the users can perform searches only with a predefined set of criteria, e.g. for a tModel or a service name. These kinds of queries are called technical searches. Non- technical searches (i.e. searches for user-defined characteristics like QoS attributes) are not possible. This seriously hampers the dynamic discovery and binding of Web services.

The UDDI Security API offers programmers a set of operations which allows them to obtain the security credentials to work with the UDDI registry. It involves operations such as discard_authToken() and get_authToken().

The UDDI Custody and Ownership Transfer API is used in the inter-registry communication to enable transferring "ownership of one businessEntities or tModels from one publisher to another" [UDDI03]. It involves operations such as get_transferToken() and transfer_entities().

The UDDI Subscription API is optional and is used in the inter-registry communication. It specifies a simple signaling (publish/subscribe) mechanism, notifying the subscriber if the piece of information it has subscribed for changes. This interface involves operations such as save_subscription(), delete_subscription(), get_subscriptions(), or get_notification(). The complementing interface of call-back operations is called the UDDI Subscription Listener API. The UDDI Replication API is another inter-registry communication API facilitating the task of keeping the set of UDDI registries' contents coherent.

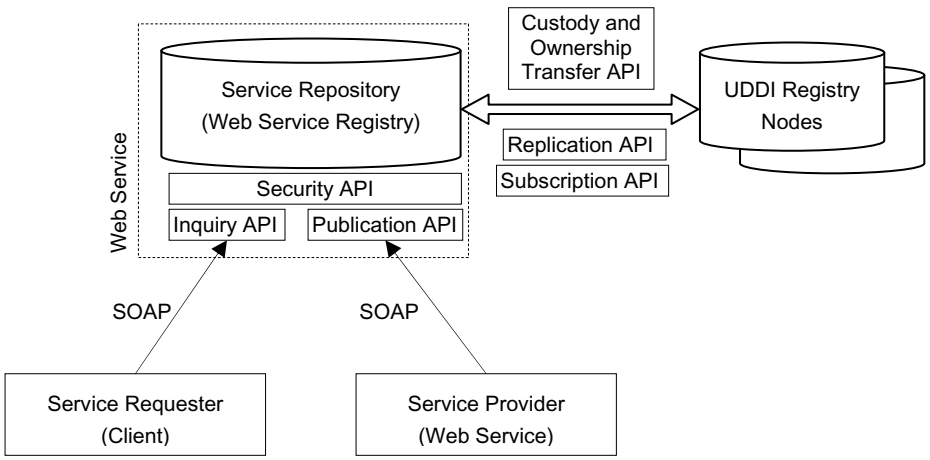


Fig. 7.13. UDDI APIs

7.5 Advanced Concepts

In order to turn Web services into an enterprise technology and let them gain ready acceptance, some features from enterprise computing and middleware technologies need to be present, such as security, transactions, etc.

Providing support for session-oriented interactions is critical to implementing complex Web service patterns. Session orientation and stateful interactions are not provided by default in the standard Web service framework, therefore they have to be implemented by custom APIs. Software vendors provide vendor-specific APIs. For instance, in the realm of Java, two APIs exist providing stateful communication: JAX-RPC [JAXR03] and JAXM [JAXM03]. Moreover, a large majority of the currently existing APIs provide stateful communication only for certain SOAP bindings – more precisely, HTTP binding. Standardization and interoperability efforts need to be made in order to provide better communication support.

Providing security for Web services is another issue representing a dynamic field of development. There are three major aspects: providing declarative security, reusing the XML security standards, and Internet security technologies. Declarative security means implementing Web service security orthogonal to the Web service application itself and delegating all the complex security tasks to the platform, i.e. Web service infrastructure. This approach has already been successfully implemented in component orientation [SiSJ02] [EdEd99]. All existing Web service specific security specifications are built on top of XML security standards and are aligned with the existing Internet security standards. These specifications include WS-Security [WSSe02], WS-Privacy [SiWS02], WS-Trust [WSTL02], WS-Authorization [SiWS02], and WS-Policy [WSPo03]. WS-Security is for the time being the one with the widest acceptance. The underlying XML security has three relevant standards. The XKMS (XML Key Management Specification [XKMS01]) is defined on top of two other standards, namely XML Encryption [XMLE02] and XML Signature [XMLS04]. Of course, existing Web and Internet security standards such as HTTPS or SSL may be used in conjunction with the above standards to guarantee secure connections (communication channels).

Providing transactional support for Web services is a crucial milestone along the way to broad industry acceptance and a key feature to guarantee reliable business interactions. Just like security, the transactional support may be implemented in a declarative manner and delegated to the platform, which in general greatly simplifies the application programming. The cornerstone in the field of Web services is the fact that the platform must be kept relatively simple (at least at the service requesters' side) so that it fits on any device. There are two existing specifications: WS-Transaction [WSTr02] and WS-Coordination [WSCO03].

7.6 Web Service Composition and Web Service Flow Languages

Having defined what Web services are, we can ask a quite logical question: Can a Web service be implemented in such a way that it calls other Web services? Such Web services are called composite Web services. They are regular Web services as discussed above, but as will be shown in this section, they may have some special properties.

As shown in Fig. 7.3, composite Web services form a layer of Web services on top of already existing ones. Such services use the functionality of the underlying ones, abstracting from them, and combining them to offer higher level functionality. It will com-

bine the order entry service, various supplier and delivery Web services, as well as at least one bank Web service. As can be easily inferred, the interfaces of the composite Web service are much more general than the interfaces of the combined ones. Composite Web services also involve lots of coordination between all the composed services.

Every composite Web service is a Web service itself and thus has a WSDL interface. If SOAP binding is supported, then the interaction with it can be done using SOAP. Every composite Web service may be published in the UDDI registry. These facts may appear to be obvious at first glance, but have profound consequences that will be discussed in the next sections.

Composite Web services are, generally speaking, written in special purpose languages termed Web service flow languages; see further in Sect. 3.6.2. The goal is to have a simple language making the task of composition easy and supporting as many of its aspects as possible. The major advantage of having such a special propose language is that composite services and their interactions can be modeled, which greatly simplifies the task of creating them. Before discussing Web service flow languages in detail, let us consider the platform for composite Web services, which will provide an insight into how they function.

7.6.1 Platform for Composite Web Services

In this section it is assumed that a composite Web service is written in a Web service flow language. When doing so, an accompanying WSDL document must be created. This document contains only the abstract definitions of the composite Web service interface.

After the composite Web services are created, they are deployed in the execution environment (Fig. 7.14). The composite service itself is translated into executable code. Additionally, the concrete part of the WSDL document is generated. The binding and the service WSDL definitions depend in general on the specific execution environment. The composite Web service may be registered with the UDDI registry as a next step.

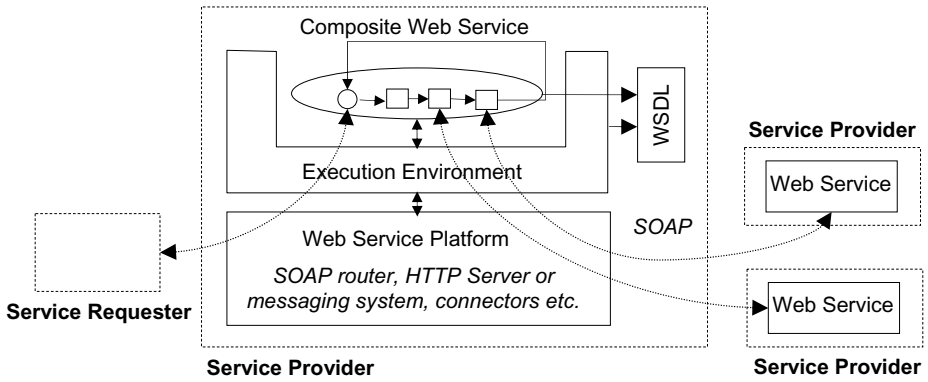


Fig. 7.14. Platform for composite Web services

The interaction with the composite Web service is not different from the interaction with any other Web service. During its execution it will of course invoke other Web services. This fact may be generalized: Web services represent a uniform invocation and in-

teraction mechanism. The inherent lack of technology bridges reveals the high integration potation of the Web service technology.

7.6.2 Web Service Flow Languages

Since 1999 there have been a number of proposals for Web service flow languages (Fig. 7.15). Microsoft developed XLANG [XLANG01] which was preceded by XAML [XAML04]. XLANG is used in Microsoft BizTalk Server [BizT04]. In 2001 IBM created the "Web Service Flow Language" [WSFL01]. The joint efforts of IBM and Microsoft combined XLANG and WSFL in 2002 in a new language BPEL [BPEL03]. Another trend of development was marked by the consortium bpmi.org with the development of BPML [BPML04] in 2001, which is complemented by WSCI [WSCI02].

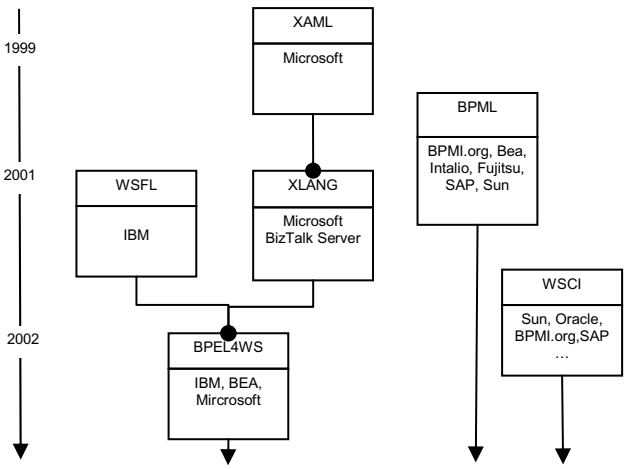


Fig. 7.15. Evolution of the Web service flow standards

XLANG and XAML were developed by Microsoft. Although these two languages are closely related they have a somewhat different focus. While XAML was designed to handle transactional support in Microsoft's e-Business solutions, XLANG was designed to be a full-fledged flow language supporting business process design (Fig. 7.15). XLANG does not target Web services explicitly. XLANG together with the BizTalk server are rather to be positioned in the field of workflow or electronic collaboration. WSFL is a flow language from IBM supporting Web service composition. To the best of our knowledge no real products supporting WSFL exist. The joint efforts of IBM and Microsoft led to a combined flow language, called BPEL4WS – for short BPEL – which replaces WSFL and XLANG. BPEL4WS supports the typical control flow constructs such as looping, branching, etc.; in addition it supports activities for parallel execution. BPEL4WS also supports exception handling and compensating transactions. The structure of BPEL processes is flat, i.e. subprocesses cannot be defined. The biggest rival to BPEL4WS is a flow language called BPML created from the consortium BPMI.org. BPML supports some additional constructs, making it more flexible in comparison to BPEL, such as subprocesses, dynamic partners, etc. There is, however, less industry support for BPML in comparison to BPEL.

In the previous section it was assumed that composite Web services are written in a special purpose Web service flow language. Although this is generally true, it is not necessarily the only plausible alternative. The developers may choose to implement a composite as a normal Web service. Such an approach might have a number of disadvantages. Firstly, composite services tend to be created by people with knowledge of process modeling and not necessarily by programmers. Secondly, creating complex process and coordination logic in a conventional programming language (e.g. Java) produces long and complex programs. Therefore it is generally perceived that the combination of a standardized Web service flow language and an execution environment may lead to more efficient results.

The goal pursued with Web service flow languages is to have a simple language, making the task of composition easy and supporting as many of its aspects as possible. Simplicity is also implied by the fact that the actual flows can be modeled instead of programmed. An additional advantage of having a standardized language is portability. Such a composite service can be deployed on any engine implementing that standard.

A common characteristic of Web service flow languages is that they are process oriented. As such they have some common constructs and implement some common aspects regardless of some differences.

All languages support control flow constructs. These are expressed in the form of activities supporting branching (if–then–else, switch), cycles (while), invocation, sequential or concurrent execution, etc. Data flow constructs are expressed through definition of types and data variables such as input or output parameters or global or local variables and activities for assignment or querying. Defining transactions is the next characteristic. Transactions are not just used in the sense of ACID operations. ACID properties of operations are considered too strict, however, transactions are an issue, therefore compensating transactions are employed as a mechanism to reverse an action. Last but not least, there is the need for exception handling – a mechanism defining exception business-critical process sections and handling erroneous behavior.

Figure 7.16 shows an example of a Web service flow, which does nothing more than serve as a simple intermediary. It receives a request from a client and passes it to the order entry Web services, which process the request. After the invocation the process collects the result and sends it back to the client.

The process `OrderEntryProcessExample` gets a request (call to the process's `pendingOEntry` operation) from a client and consumes it in the receive activity. In an assign activity the message parts of the incoming message are copied to another message (container) used further to invoke the order entry Web service (Fig. 7.7).

The invoke activity (Fig. 7.16) calls the order entry Web service and stores the result in the `invocationresponse` container (variable). The returned list of order entries is then copied to another container response in the assign activity. In the reply activity the returned result is sent back to the client.

The use of processes provides certain advantages over programming in the context of composite Web services. Firstly, the experts need to know only the Web service flow language. In contrast, Web service developers need to have in-depth knowledge of multiple technologies. Secondly, the Web service flow is modeled. Modeling is in principle much simpler in comparison to programming. It requires less technological skill and more business understanding. Last but not least, the Web service flow language constructs are high level and special purpose, i.e. process oriented. Programming a composite Web service in a programming language such as Java will be more time consuming,

and the program itself will be longer and more complex. Features such as concurrent process execution and partner selection will be more complex to implement. In general, features of Web service flow languages such as simplicity and independence of the underlying platform help developers to develop complex Web service compositions efficiently and in less time.

```
<process name="OrderEntryProcessExample"
         targetNamespace="http://www.oe.de:8080/axis/services/
                                                    OrdEntryProcess"
         xmlns:tns="http://www.oe.de:8080/axis/services/OrdEntryProcess"
         xmlns:tnsProcess="http://www.oe.de:8080/axis/services/
                                                    OrdEntryProcess"
         xmlns:tnsProvider="http://www.oe.de:8080/axis/services/
                                                    OrdEntry"
         xmlns="http://schemas.xmlsoap.org/ws/2002/07/
                                                    business-process/">

   <containers>
      <container name="request" messageType="tns:request"/>
      <container name="response" messageType="tns:response"/>
      <container name="invocationrequest" messageType=
                              "tnsProvider:pendingOEntryListRequest"/>
      <container name="invocationresponse" messageType=
                              "tnsProvider:pendingOEntryListResponse"/>
   </containers>
   <partners>
      <partner name="caller" serviceLinkType="tns:OrderEntrySLT"/>
      <partner name="provider" serviceLinkType=
                                    "tnsProcess:OrderEntrySLT"/>
   </partners>
   <sequence name="sequence">
      <receive name="receive" partner="caller"
         portType="tns:OrderEntrySLT" operation="pendingOEntry"
         container="request" createInstance="yes"/>
      <assign>
         <copy>
            <from container="request" part="in0"/>
            <to container="invocationrequest" part="in0"/>
         </copy>
         <copy>
            <from container="request" part="in1"/>
            <to container="invocationrequest" part="in1"/>
         </copy>
      </assign>
      <invoke name="invoke" partner="provider" portType="tnsProvider:
         OrdEntry" operation="pendingOEntryList" inputContainer=
         "invocationrequest" outputContainer="invocationresponse"/>
      <assign>
         <copy>
            <from container="invocationresponse" part=
                                    "pendingOEntryListReturn"/>
            <to container="response" part="pendingOEntryListReturn"/>
         </copy>
      </assign>
      <reply name="reply" partner="caller" portType="tns:OrderEntrySLT"
         operation="pendingOEntry" container="response"/>
   </sequence>
</process>
```

Fig. 7.16. Example of a BPEL4WS process

7.7 Assessment

So are Web services a revolutionary technology? The answer is simple – no. None of the SOA components is genuinely new. These ideas have existed for a long time. For example, the origin of WSDL as an interface description language dates back to DCE RPC IDL. UDDI originated from the middleware interface repositories, e.g. the CORBA interface repository. Besides, any middleware or component-oriented framework has a custom communication protocol: DCOM has ORPC, CORBA has IIOP, RMI uses IIOP as well.

However, the potential scope and environment are what distinguish Web Services from other SOAs. Web services have no analogue in a Web environment. The potential scope of Web services is the potential scope of the Web itself, which means different device types ranging from handheld devices to powerful servers, and various heterogeneous software platforms.

Web services ensure interoperability. These services are based on standardized technologies. As long as the developer community follows the standards, the developed Web services will be able to cooperate with each other. Web services also leverage existing and widespread technologies.

8 Web Site Engineering and Web Content Management

So far we have discussed different programming languages, platforms, and concepts for designing Web applications and accessing data. In this chapter we present a different perspective on the development of Web applications: we shift from a view on the programming languages to a view on the documents to be published via Web technology and how they can be created with respect to efficiency and reuse.

8.1 History of Web Site Engineering – from Engineering in the Small to Engineering in the Large

At the dawn of the Internet the Web was a collection of static HTML pages, which had to be manually created and maintained. These static pages were mainly written and maintained by using text editors, often with HTML extensions to highlight and display the hypertext in a comfortable way. The HTML "programmer" had to write the HTML "code" mostly by hand. Text-based approaches are applicable to simple and static pages such as personal homepages where the creator can easily have complete control over the produced page. Later on, graphical Web site creating and editing tools emerged. Their goal was to support experienced Web designers on the one hand and to enable non-experienced editors without HTML knowledge to create and edit Web sites, on the other hand. These tools spread rapidly and are widely used despite problems with the correct generation of corresponding HTML code. The result of this approach is often non-standard and non-minimal HTML code. But their application domain is at least the field of small Web pages.

Over time the requirements for Web sites changed. Content had to be published from databases and the need for more comfortable editing arose. A solution reflecting this requirement was provided by the dynamic generation of Web sites, which allowed for the publication of whole product catalogues, for example, from the content stored in databases. For instance, server side scripting languages can be used to fill HTML templates with different content by substituting parts of the pages by server side code generating the content dynamically. Similar results can be achieved by other server side application programming concepts often referred to as template concepts. With XML gaining broad acceptance a new transformation-based approach resting on the separation of structure, content, and layout arose. This concept is by far not new. There are much older approaches, such as GenCODE, ODA, or SGML [SGML04]. With this concept enormous flexibility arises [RoRi02], so we will discuss the concepts and advantages in the next section.

In this chapter we describe how Internet publications can be generated based on the separation of structure, content, layout, logic, and semantic description. Then we present the concepts and architectures of Web content management systems, which are broadly used to create and maintain medium and large-size Web sites.

8.2 Separation Aspects

This section provides an approach for administrating and handling the separation of structure, content, and layout in an orthogonal way, which can be gradually extended by logic and semantic description (Fig. 8.1). In contrast to many other presentations of this topic we did not choose a standard approach, but we are focusing on the orthogonal handling of these aspects as shown further on. An overall document generation process chain is depicted in Fig. 8.2. Its content will become clear and successively reconstructed in the next sections, starting with the structure and its mapping to assets, the inclusion of program logic, and ending up with the transformation in the final layout.

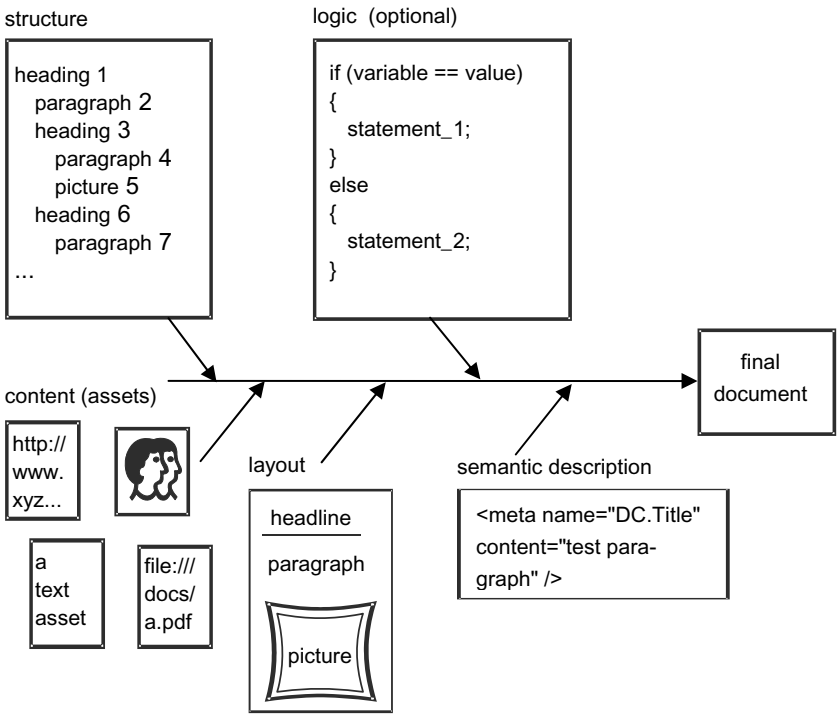


Fig. 8.1. Separation of structure, content, layout, logic, and semantic description

One may ask why such an orthogonal handling is necessary; the answer is more than simple and pragmatic. Is allows for the flexible handling of documents in such a way that the same content can easily be published in different output formats, often called the single-source–multiple-media principle. Let us consider a Web page that offers different news articles about an enterprise published in an HTML document. Now a user wants to print several news articles published on the portal site. Obviously the user prefers a print version without navigation and other site elements, but requires the content in a print style. According to the separation concept discussed here, this can be done by changing the structure of the site (only then the news heading, the text, and author are no longer of interest) and using a different layout description. And if the user wants to download a

PDF document this is also no problem: just the layout description has to be changed and replaced by a stylesheet allowing for PDF output.

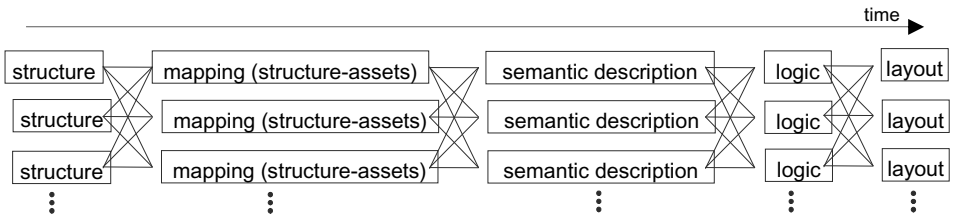


Fig. 8.2. Document generation process chain

8.2.1 Structure

How the parts of a Web publication are organized into pieces is called structure. The sequence of several paragraphs, the separation into headings and normal text, or the inclusion of references to pictures are some examples. Such structural characteristics can be described hierarchically; therefore markup languages (e.g. XML) are the most logical mapping choice. A structural description requires careful identification of content data depending on the desired granularity. In an optimal case the structure does not directly name elements of the content; it will rather contain indirect and abstract references called placeholders. This allows for the creation of a set of structure definitions completely independent of the content. How the related content is mapped onto the structure is a question that will be answered in a consecutive step.

In the initial phases of the publishing process many approaches tend to use templates, which are already and partially populated with content and only have a few placeholders for dynamically filled assets. These approaches led to a high number of templates, even though they do not differ much in structure. Hence the approach presented here uses a clean definition of structure, replacing all connected content parts by placeholders (special tags), and allowing the inclusion of different contents into one and the same structure. Conversely, a single piece of content can be mapped onto different structures. Figure 8.3 (upper part) visualizes such a structure description in which the placeholders are represented by the tags `<element orderid=...>`. Two structures are depicted, both consisting of four placeholders structured in different styles. In structure 1 one placeholder is grouped as a heading, another as a paragraph, and two placeholders are subsumed as paragraphs. In contrast to this, structure 2 on the right contains four placeholders all tagged as paragraphs.

8.2.2 Content

In a consecutive step the selected content must by inserted into the structure at the places marked by a placeholder. Therefore it is necessary to “disintegrate” the content down to its atomic parts, the digital assets. The granularity of these assets can vary from a few words (e.g. a product name) to longer texts with several paragraphs, and to pictures. These assets must now be mapped to the corresponding placeholders (mapping), which

can be identified by a unique number. By using this number it is possible to position a concrete asset at a certain location within the structure (Fig. 8.3). This operation can be carried out by an editor, who has to map one placeholder to one asset in a stepwise manner. This enables content reuse across different structures.

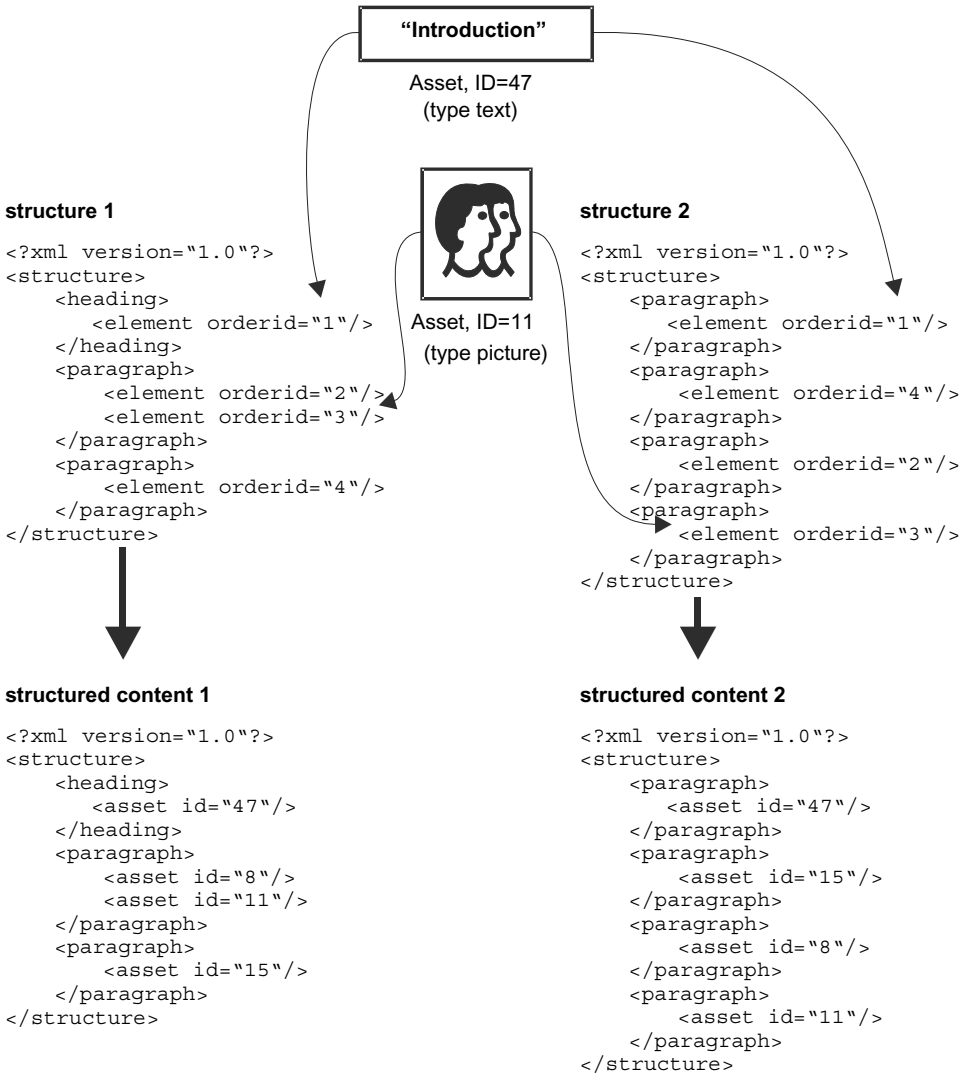


Fig. 8.3. Bringing structure and content together

Consider this example: two different structures are visualized containing four placeholders each. The asset "introduction" (ID=47) of type text is mapped to placeholder 1 in both cases. The asset of type picture (ID=11) is mapped to placeholders 3 and 5. An analogue mapping operation is performed on assets 8 and 15, but they are not depicted in the example.

With the structure being defined in XML in our approach, it is much easier to do the mapping between placeholders and assets by utilizing XSLT [XSLT04]. XSLT processors are available in most XML-enabled systems. By using special rules the XML documents are transformed in such a way that the placeholders are substituted by assets or at least by references to assets. The corresponding XSL stylesheet is shown in Fig. 8.4. It contains four matching rules, where each matching rule does one replacement of a placeholder (e.g. `<element [@orderid='1']>`) by a concrete asset (e.g. `<asset id="47"/>`). To store the mapping between placeholders and assets explicitly in a stylesheet is not essential: if necessary, the XSL stylesheet can be dynamically created from the mapping data in a database hiding the stylesheet generation from the user. The mapping, assisted by a graphical interface allowing drag and drop between placeholders and assets, is done by the user. This approach allows for context-specific data combinations. In a shopping system, for example, different products with different product data schemata can be mapped to one structure. When describing a book in the shop, the number of pages may be of interest, when describing a CD this may be the playing time. Both can be mapped to the same structure by defining orthogonal structural templates.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="element[@orderid='1']">
    <asset id="47"/>
</xsl:template>
<xsl:template match="element[@orderid='2']">
          <asset id="8"/>
    </xsl:template>
    <xsl:template match="element[@orderid='3']">
          <asset id="11"/>
    </xsl:template>
    <xsl:template match="element[@orderid='4']">
          <asset id="15"/>
    </xsl:template>
</xsl:stylesheet>
```

Fig. 8.4. XSL stylesheet for the mapping in Fig. 8.3

8.2.3 Layout

What we have described so far is structured content, or, to be more precise, content mapped to a structure. Now the layout of a document can be defined by using an XSL template defining the rules for the transformation in the desired output format. For instance, it can be determined that a title, marked by a <heading> tag, should be published as a first-order heading tag in HTML. By using different layout templates, different destination formats can be addressed (e.g. XML, HTML, or PDF).

The single components of the structured content are transformed into the selected presentation format by using a step-by-step method. The result need not be in the final format. The process chain of transformations can by extended by as many steps as necessary and thus further XSL transformations can be added. This allows for independent development of content and layout and gives the possibility to present the same content ad hoc in different output media – what is often called the single-source–multiple-media principle. The transformations in the process chain can be done in parallel as long as possible but must be split in a later step to achieve different output media formats. Therefore it is possible to generate an on- and off-line version of a product catalogue or two Web pages with the same content but different personalized layout with less additional effort.

8.2.4 Program Logic

So far we have discussed the separation of structure, content, and layout and its application in a very orthogonal way. But often the discussion cannot be restricted to these three parts. It is becoming increasingly important to add special instructions (program logic) to the content, which are used for interaction with the user. If program logic were treated implicitly the developer would be forced to process the program logic together with the layout. This would contradict the orthogonal treatment of the different aspects. The creation and maintenance of the XSL stylesheets – containing layout and logic – needs more knowledge of both the layout and the programming language. From this point of view it is necessary to treat logic and layout independently, which allows for a supplementary step in integrating logic in content-oriented Web applications.

The most important advantage arising from this approach is the ability to add to a publication different elements of program logic just modifying one step in the transformation process chain of the publication process. This allows, for example, for embedding different JavaScript functions dependent on the user's browser. One implementation of such an approach is realized in the open source XML publishing framework Cocoon [Coco04] of the Apache Software Foundation. By using the concept of Extensible Server Pages (XSP, [XSP04]) one approach of treating content and logic separately can be shown. The XSL stylesheets used to embed Java program code in a XML document are called logic sheets. Figure 8.5 illustrates this method of separating content and logic in the publishing process chain. Each logic paragraph is explicitly encapsulated by a special XML tag (xsp:logic), which is the enabler for the subsequent embedding. To mark the location where logic has to be inserted, so-called Taglibs [Jaka04] are provided. They are a kind of function library defining various XML tags, which are to be substituted by the corresponding Java source code afterwards. Attributes can be successfully utilized to implement different kinds of parameters.

Figure 8.5 depicts a structure (top left), which is extended further on by a piece of XSP code. This code checks a variable containing the number of incorrect authentication attempts. After three failed attempts a notification is issued and page access is denied. The XSL stylesheet inserting the XSP code into the structure is depicted at the top right. The stylesheet contains a matching rule that searches for the last occurrence of the paragraph tag and fills in the logic described in the stylesheet. The result is an XML document with the same structure as the source document, but now containing the XSP script.

Fig. 8.5. Orthogonal embedding of logic in a XML document using XSP

The inclusion of logic can in general be done in a separate step before transforming the document into the destination format. While Cocoon focuses on XSP and server side logic, we want to point out that this concept can be used more broadly. The approach is able to treat both client and server side scripting extensions. The system must only be instructed whether or not to interpret the scripts before transferring them to the client. We can see a promising application domain especially in the field of client side scripting. It is important to consider the specifics of the different browsers and often a version without client side scripting code must be provided for users having disabled, say, the JavaScript support option.

8.2.5 Semantic Description

So far we have considered the aspects of structure, content, layout, and logic. Semantic description can be easily identified as a missing component. Semantic Web concepts allow for the semantic markup of content which consequently allows for better search capabilities and automatic interpretation of content by computers. Especially, the better search capabilities are an important factor when talking about content in the Web or in company intranets.

To illustrate this let us assume an enterprise having a number of employees. These are listed and described on a Web page. Let us further assume that one is looking for a contact person of this enterprise. When using a traditional search engine one can use as key-

words the “contact person” and the name of the enterprise. What one gets back as a result is most probably not very satisfactory, since the search engine displays all sites containing the named keywords. Such an inquiry can be enhanced by semantic description. More precise queries can be constructed (i.e. name all contact persons of a specific enterprise X). This is only possible if the semantic description on the Web site identifies the organization as an enterprise with a name and if all contact persons are semantically described as such. By doing this search capabilities can be enhanced.

Therefore we are convinced that it is necessary to integrate a notion of semantic description into Internet publications and to provide concepts for this. Before having a closer look at the three different approaches for semantic description in Web pages, we want to introduce briefly the basic Semantic Web concepts.

8.2.5.1 RDF and RDF Schema

The Semantic Web is a W3C initiative [SemW04]. The goal is the content-oriented description of Web resources, which allows for automatic processing, based on logic connections and conclusions. The Semantic Web utilizes several standards and technologies depicted in Fig. 8.6. In the bottom layer one can find Unicode as the internationally accepted encoding standard for symbols. The URI is a superset of the URL and is used as an identification mechanism. In the second layer XML and XML Schema are used as a common syntactical basis for the semantic description. Based on XML, RDF can be used for the semantic description of resources in the Web and RDF Schema to extend the grammar used by RDF. In the next layer several ontology languages (e.g. DAML+OIL or OWL) are depicted, which allow more complex ontologies to be defined based on the layers above. They are not important for the semantic description approaches we introduce in the following; more information can be found in [WebO04]. At least a query interface is necessary to use the semantic description. This can be done directly or via an inference engine. Such an engine should be able to conclude new information based on the existing description (Fig. 8.6).

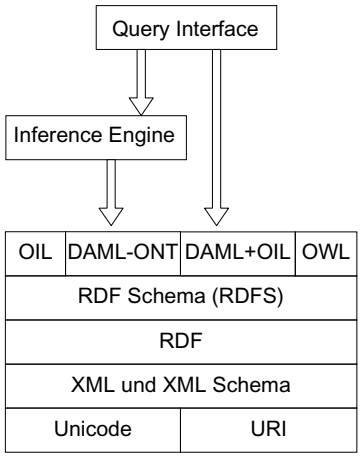


Fig. 8.6. Semantic Web

After this brief introduction to the layers we can now concentrate on RDF and RDF Schema for semantic description of resources. In this context the term descriptive metadata is often stressed: "The RDF data model provides an abstract, conceptual framework for defining and using metadata" [RDFC04]. At present, metadata in relational databases is mainly used to describe the structure of tables. The term metadata in the context of the Semantic Web focuses mainly on the description of objects and their meaning. From a technical point of view this description is used to reconstruct missing schemata of resources. The term resource is of great importance in the field of the Semantic Web, whereas all parts of an Internet publication, e.g. text paragraphs, pictures, or links, are subsumed under this term. Resources can be referenced uniquely by using a URI.

The description of resources is done via triples (called RDF statements): subject, predicate, objects. The subject is the resource to be described. The predicate represents a special attribute, a property, or a relation. The corresponding value is represented by the object. Both subject and predicate are resources. An object can be a resource or a literal, i.e. a value of string type or an XML fragment. Since objects can be reused as subjects in other statements, the data model is a directed labeled graph.

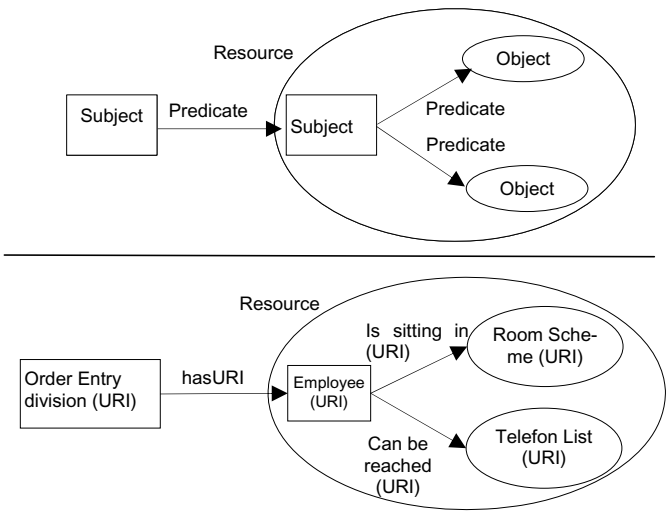


Fig. 8.7. RDF triple: subject, predicate, object and example

Figure 8.7 (upper part) shows an example of an RDF graph where resources are depicted as ovals, literals as rectangles, and predicates as directed and labeled graphs. The lower graph can be interpreted as follows. A division dealing with order entry has employees. One employee is sitting in a room, which is described on the Web site as a room scheme and can be reached via a telephone number described in a telephone list. Due to the fact that the RDF can only describe properties of resources, it is not possible to define new classes or attributes or rather rule for their usage. RDF Schema has been developed for this purpose. It uses the syntax of RDF but allows for additional elements.

8.2.5.2 Different Approaches for Integration of Semantic Description

After having introduced the basics of the Semantic Web, we will now elaborate on how to integrate Semantic Web descriptions into Web pages. To be able to show this in a very concrete and precise way we use the introduced separation of structure, content, layout, and program logic as an approach for creating Internet pages under optimum conditions with regard to reuse and flexibility.

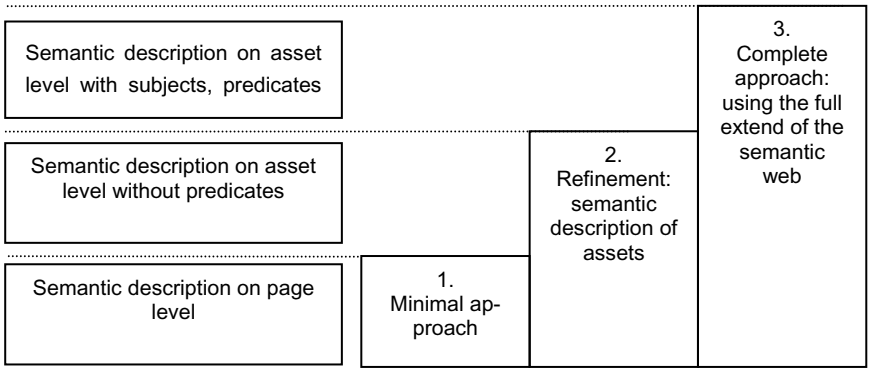


Fig. 8.8. Three different approaches for semantic description

The simplest approach (Fig. 8.8, 1) for enriching internet publications with a semantic description is to use descriptions at a page level, i.e. the HTML meta-tags. In this way certain characteristics of a Web site can be defined – for example, the author, the title of the page, or the creation date. As can be easily seen, it is impossible to describe a singular asset within the page. This leads to coarse granularity but has to be considered when HTML is the output format, which restricts the use of meta-information in Web sites to page-level descriptions [XHTM04]. This approach is bound to the HTML standard. The position between the <HEAD> tags of an HTML page is the appropriate place for this description. In RDF one should use Dublin Core [Dubl04] for defining the properties (Fig. 8.9). With Web sites being the lowest granularity supported for semantic description meta-information has only to be produced and maintained at this level, which reduces the effort. In the overall approach this is similar to using HTML meta-tags and not entirely satisfactory in the context of the Semantic Web.

This page-level granularity allows for a better description of whole Web pages. At a higher level granularity it is obviously useful to describe not only pages but the assets (Fig. 8.8, 2) within the pages enabling a more detailed search. For this the assets must be enriched with attributes containing the semantic description (Fig. 8.10). So this approach rests on subjects and objects and does not use predicates in comparison to the full Semantic Web concepts. Thus it is not possible to express statements like “The Chair of Computer Science 6 has employees”, but objects can be statically described without regard to their context. Let us consider a picture of an employee. A semantic description about the employee’s first and last names can be assigned to this picture. Those are statically connected afterwards to the asset and independent from the concrete use of the picture within several Web sites. In many application areas such a description is satisfactory because the semantics do not change across the different usages.

```
<HTML>
<head>
<rdf:RDF xmlns:rdf=http://www.w3.org/1999/02/22.rdf-syntax-ns#
 xmlns:dc="http://purl.org/dc/elements/1.1/">
<rdf:Description rdf:about="http://www.xyz.org/example.doc"
       dc:creator="WCMS"
       dc:contributor="John Example"
       dc:title="test pages"
       dc:description="A page containing semantic description."
       dc:date="2002.10-02" />
</rdf:RDF>
</head>
<body>
 ...
</body>
</HTML>
```

Fig. 8.9. Example – meta information for Web pages

The third and complete approach (Fig. 8.8, 3) enables the full capabilities of the Semantic Web. Assets can be described not only by static attributes (subjects and objects) but also by using predicates for logical assignments. This approach is an extension of the previous approach which allows for context-specific descriptions instead of static ones. Context-specific semantic descriptions vary with the context the asset is used in. Imagine an assignment between the pictures of persons and their projects. It is not useful to express the projects of persons as a static attribute of their pictures. Because of this it is necessary to express this assignment on a page containing both the pictures and the project by a predicate. This approach enables on the one hand the full capabilities of the semantic but leads on the other hand to more expense in maintaining and creating the descriptions. When deciding on one of the three solutions one has to consider the trade-off between precise and detailed semantic descriptions and the corresponding expense in maintaining and creating the descriptions.

```
<p id="Test">
<meta name="DC.Title" content="test paragraph" />
<meta name="DC.Subject" content="example, Dublin Core" />
<meta name="DC.Producer" content="WCMS" />
This is a text paragraph.
</p>
```

Fig. 8.10. Example for semantic description with <meta> tags in XHTML

8.3 Web Content Management Systems

So far we have talked about structure, content, layout, logic, and semantic description from a conceptual point of view. Now we want to take a closer look at a class of systems dealing with Web content management and the aspects of such systems.

At present, it is almost obligatory for an enterprise to be present on the Web, so the effort to keep content and structures of this representation up to date has often been underestimated. Web content management systems have been established as a tool for collecting, creating, editing, administering, and publishing content on the Web. In this section we give a definition of Web content management systems, and consider important aspects, management processes, and architectures. To be more precise, Web content management is the systematic and structured procurement, creation, preparation, administration, processing, publication, and reuse of content [RoRi02].

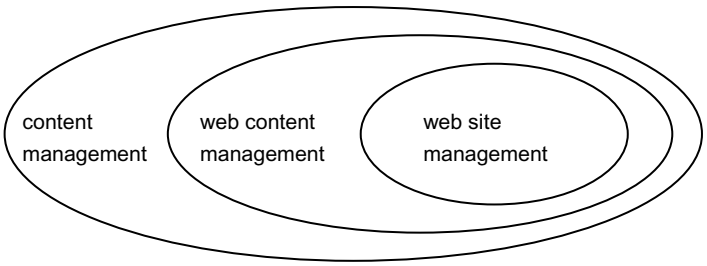


Fig. 8.11. Demarcation of Web content management

In Fig. 8.11 one can see a definition of the term Web site management, Web content management, and content management. Web content management is a part of the domain content management focusing primarily on Web content. Web content is information, documents, and data published over Internet technology. In contrast to this, Web site management as part of Web content management deals with the technical control of Web sites, e.g. link checking or data and file storage and access. The term Web content management can be described more precise by its constituents: Web, content, and management. Publication is done over Internet technology, exactly over the so-named i*nets: Internet, intranet, and extranet. Under content any information can be subsumed. Content consists of digital assets, e.g. pieces of text, pictures, or tables. The third term, management, implies the process character and subsumes different processes that must be enabled and supported. In this section we want to explain how Web content management systems can support the publication of Web pages by coordinating the tasks to develop and maintain a Web site.

The main goals when introducing a Web content management system can be grouped into two classes. The first class deals with process improvement concerning the publication process. By mapping and realizing the business process into workflow schemata within the Web content management system the activities of Web content management can be supported, recorded, supervised, and afterwards analyzed. This leads to an improvement in quality and to a reduced period between investigation and publication of the content. Optimized supported process lead besides the reduced time and improved

quality to cost reduction. The second class deals with more efficient content creation and administration. This is achieved using the concept of separation of content, layout and structure, logic, and semantic description. This concept is the key to enhanced reuse in Web content management. The same content can easily be published in different output formats, e.g HTML or PDF, by using different stylesheets. But it also allows users with few or no HTML skills to edit the content separate from the other aspects.

8.3.1 Characteristic Aspects

Web content management systems have to support different processes involved with creating and maintaining Web sites. The content management lifecycle is a high-level abstraction of the most important process. Different users and roles are involved in this process; they have to deal with different classes of content. Characteristic use cases are shown further on; they help to understand the need for process support.

8.3.1.1 Content Lifecycle

Lifecycles are an established visualization of process-oriented facts. A typical Web content management lifecycle [ZsTZ01] – the core of each Web content management system – is shown in Fig. 8.12. At the beginning the content must be investigated and created by an editor. Depending on the assets different tools are needed, e.g. a text or image processor. Afterwards the content must be supervised by one or several other persons (e.g. a chief editor) before it can be released. Sometimes important documents must be supervised by a content manager, by a lawyer, and by a managing director. Often the process is more complicated in practice.

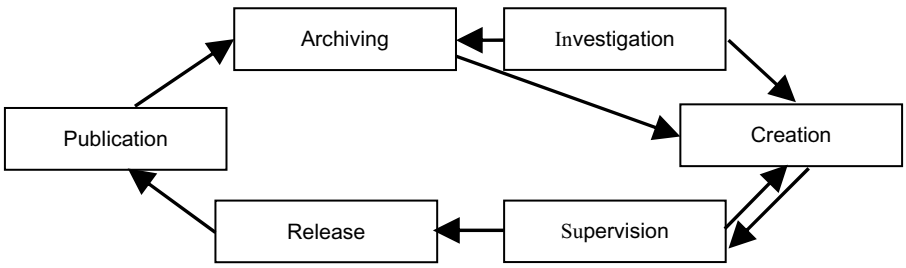


Fig. 8.12. Web content management lifecycle

If the document is not released it must be sent back to the author often supplemented with some additional information or instructions. After a successful supervision the document can be released and the content can be published. When the content is no longer needed or becomes outdated it can or has to be archived. This is sometimes needed to document what content was published at which time (e.g. as proof in a lawsuit) and also often in the context of reuse. The archive can be a public archive on the Web site or an internal archive.

8.3.1.2 Users and Roles

The high-level description of the content lifecycle already provides a good overview of the different tasks to be performed in Web content management. These different tasks are less of a problem in small companies and organizations than in bigger ones where many different constituents in different places work together.

Figure 8.13 illustrates a situation where Web site management is done without a content management system. The editors are creating content, consult with partners without process support, and send the results to an employee responsible for HTML coding and programming within the organization or possibly to an external service provider. Prior to this the content must be released.

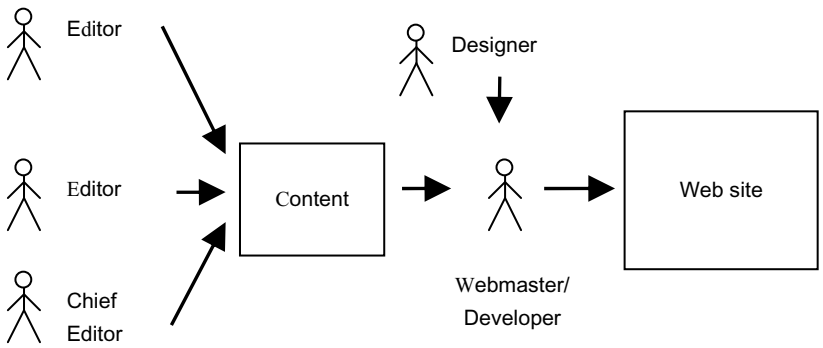


Fig. 8.13. Publishing process without a Web content management system

The Webmaster, often supported by a designer, creates the pages and publishes them on the Web server. This process does not run under efficient conditions: the Webmaster is a bottleneck within the system. Quality management is also a problem; in most organizations the release is more or less defined and seldom supervised. An editor can easily publish something without asking the chief editor, perhaps because the chief editor is not available. Archiving is often neglected and more or less structured by a Webmaster.

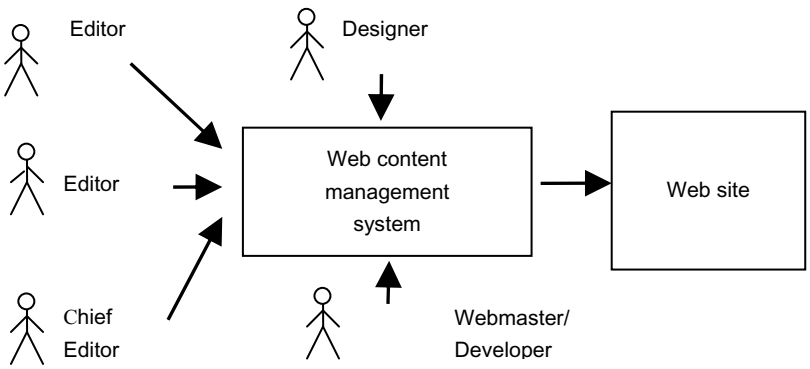


Fig. 8.14. Publishing process using a Web content management system

Figure 8.14 shows the improved situation when using a Web content management system, whereby the conceptual difference becomes clear. The Webmaster is no longer the bottleneck in the publishing process. All participants work together with different rights but in the same system, which coordinates the release process and allows editors to publish or maintain content independent from the capabilities of a Webmaster.

8.3.1.3 Characteristic Use Cases

So far we have discussed the content lifecycle and user and roles in Web content management. To illustrate the different complexity and types of publication processes we illustrate two use cases in this section, both based on the organizational structure in Fig. 8.15.

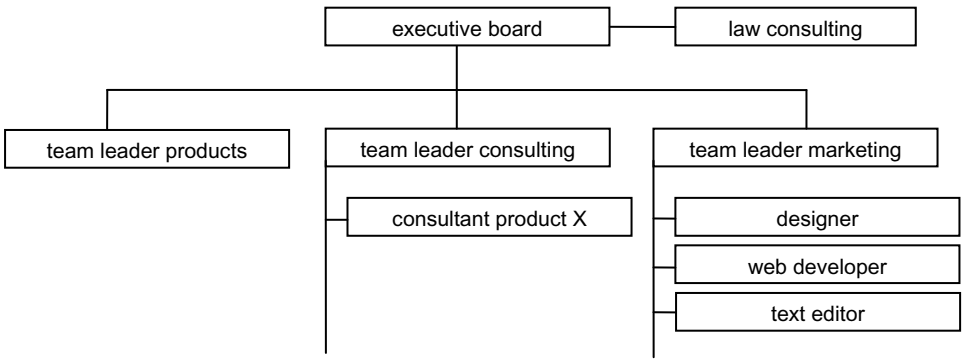


Fig. 8.15. Organizational structure

The use cases or example processes are based on the assumption that the publishing process is based on a division of labor and supported by a Web content management system. The organization is headed by an executive board assisted by consulting lawyers. The divisions are product, consulting, and marketing all having a division leader. Within the consulting division there is a consultant for product X and within the marketing division there is a designer, a Web developer, and a text editor.

In Fig. 8.16 two sample processes are depicted, differing in their complexity. The process example shown on the left shows the creation of a description for a new product X. Several persons with several tasks are involved. The concepts and their graphics must be entered, revised by a text editor and by a designer, and a template must be created by the Web developer. Afterwards the team leader and a law consultant must release it. Different roles must be integrated in a predefined way, e.g. not to forget the release of the law consultant. The process example depicted on the right of the figure is by far less complex. This is due to the fact that cyclic content is entered, which must be published regularly, and the same template or stylesheet is always used. The content does not include a picture so no revision is needed for it. It must only be entered and revised by a text editor before publication. These two examples show the need for a flexible and adaptive process support in Web content management systems. A workflow management component is needed to achieve the support of different processes. Hard-coded processes cannot fit complex requirements and the necessity for flexible process support often in-

creases with process complexity. In the context of a concrete project this means that the real-world processes must be analyzed and the possible systems must be tested whether they can map these processes or not.

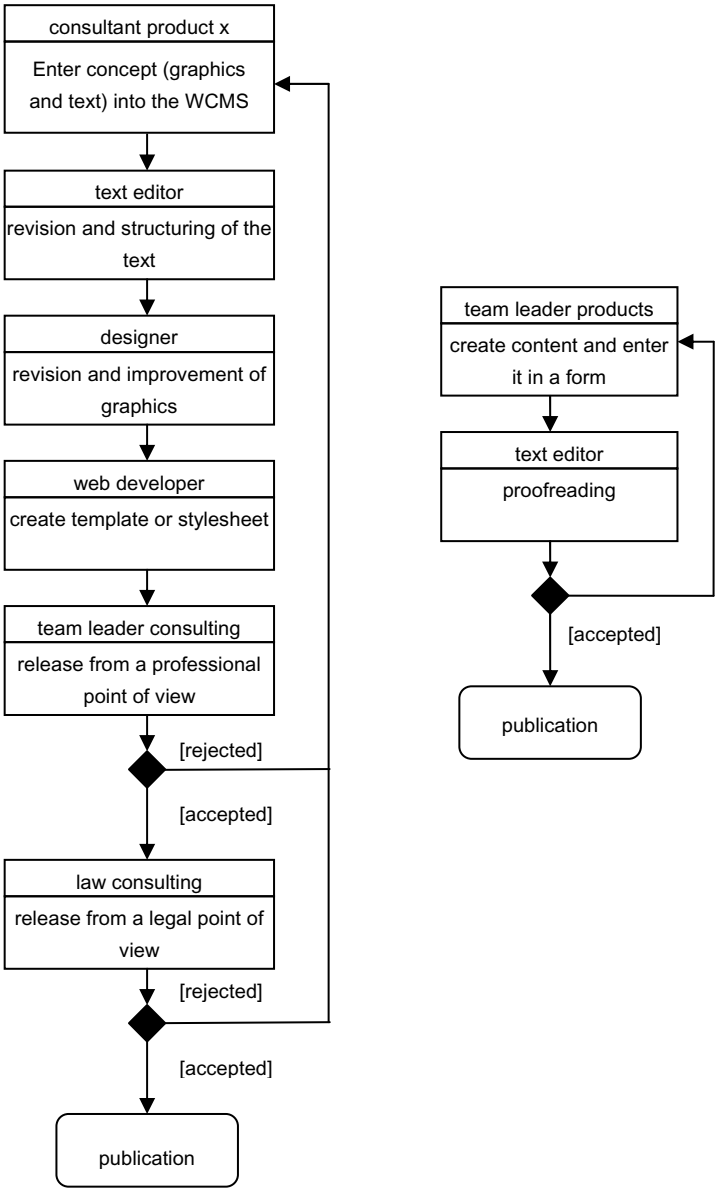


Fig. 8.16. Two example processes – different in complexity

8.3.2 Functional Architecture

In this section we introduce a high-level functional architecture for Web content management systems. Not all real-world systems follow this architecture. Sometimes functionality is missing or the building blocks are grouped in another way. But the architecture helps us to understand the functionalities of different systems and to compare them. Fig. 8.17 gives an architectural overview and the following paragraphs elaborate briefly on the different components.

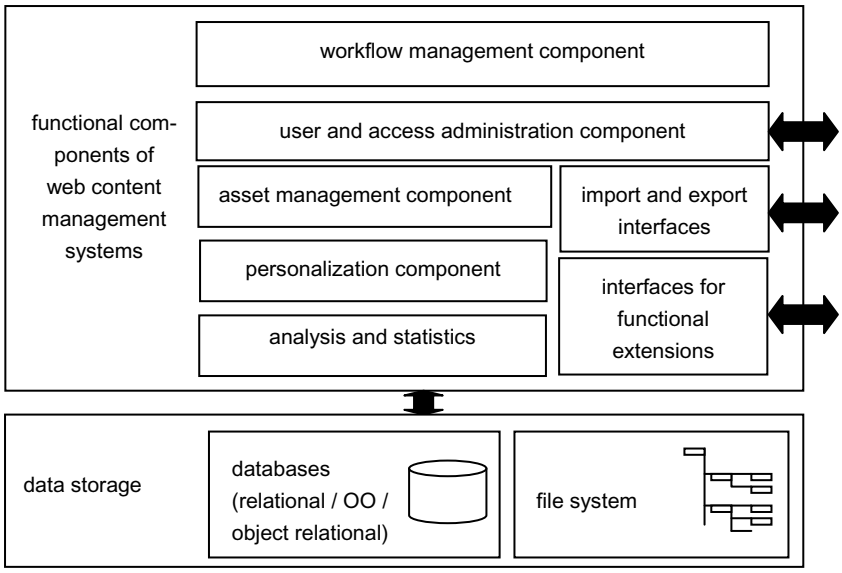


Fig. 8.17. Architecture of Web content management system

The asset management covers all functions to publish, administer, structure, and present the content. Digital assets are the components of an Internet publication, e.g. text, pictures, or videos available in a defined type. The greatest part of the systems available on the market is based on relational databases, other systems on object-oriented databases. Some systems are only based on the file system, which enables content management without a database license. There are also hybrid systems, where the database is used to store text or tables and the file system is used to store pictures or large multimedia elements. This approach rests on the fact that binary large objects (BLOBS) lead to poor performance when stored in a database. The goal of asset management is to store not only content but also descriptive metadata for the content (Sect. 8.2.5).

The workflow management component supports the handling of the processes on which Web content management is based. From a workflow management point of view [SPKH01] there are no really new requirements, but Web content management must be consequently implemented or integrated in Web content management systems. One prerequisite here is the complete description of the organization including the tasks to perform. Based on this organizational perspective the workflow management component is able to assign tasks to users. The second prerequisite is to map and transform the Web

content management processes, which differ in the real world from company to company and from application to application, into workflow schemata enabling the execution. Finally, the designed workflow schemata have to be executed. The workflow management must offer functionality to notify the user about tasks (perhaps in a work list or by e-mail), to set priorities and filters, to visualize the distribution and allocation of tasks, and to log tasks and analyze them (history function). Often it is useful to give users the possibility to define rules for assigning tasks automatically to a deputy.

The user and access management has mainly to perform two tasks within Web content management. The first task is to protect the system from unauthorized access. The second task can be characterized by the question "Who is allowed and has to do something?" This second tasks is similar to the organizational perspective within the workflow management component and will be described in Chap. 12.

Import and export interfaces play an important role within Web content management systems. When such a system is introduced into an organization, content formerly often manually maintained has to be imported into the system. Export interfaces are needed, for example, to produce a CD with static content for off-line distribution. When up-to-date content must be imported or exported, automatic content exchange functionality is required besides manual content import and export. This process of multiple reuse by several publishers is called content syndication. The goal of the publisher is not to create the content in-house but to buy and import the content. The "information and content exchange protocol (ICE)" developed by the ICE Authoring Group [SPKH01] is an example of such a tool allowing for the periodic and Web-based exchange of content by using a uniform standard. Comparable to the publish/subscribe concept, ICE has two roles: on the one hand the publisher, called the syndicator, and on the other hand the subscriber, receiving content after an agreement about the modalities (price, time, actuality, etc.), called subscription. Other concepts in this context are Open Content Syndication or NewsML.

In dealing with Web applications Web content management plays an important role for mainly content-driven applications. Often content management system must be integrated in other systems (e.g. e-commerce applications) or extended by some functionality (e.g. a guestbook). For this purpose a Web content management system must offer possibilities for the functional extension of integration into other systems. A simple way to add some functionality is to use server side scripting languages as discussed in Chap. 5. Often common languages like JSP, PHP, or ASP are used by the systems or sometimes proprietary languages are provided.

Many content management systems offer support for analyzing access logs. There are several standalone tools for Web log analysis available on the market, but in combination with Web content management systems they have one main disadvantage: traditional log file analysis is based on the URL addresses of the Web pages. In Web content management systems these URLs are often not of a very high expressiveness. Here the IDs used to identify special pages must be referred to the classification of content and pages in the Web content management system. This is why integrated support is needed for the systems. More and more one can find the need for personalized information delivery. Many content-driven applications, e.g. in the field of customer relationship management (CRM) or one-to-one-marketing, require the content to be tailored to the users' preferences.

8.3.3 Server Architectures and Scalability

In the field of Web content management two major architectural approaches concerning the server configuration and scalability are relevant. They differ in the way they create Web sites from their parts of structure, content, layout, logic, and semantic description: when using the staging concept the pages are created once and only static Web pages are sent to user; when using the live server concept each request results in the generation of a Web site on the fly.

In the staging concept all parts of a publication are stored and administer on a publishing server. Triggered by an explicit release or a time-based mechanisms the publishing server generates the static Web pages and transfers them to the Web or better staging server. From there the pages are transferred to a client by request. No generation is done at request time. Fig. 8.18 depicts a staging architecture.

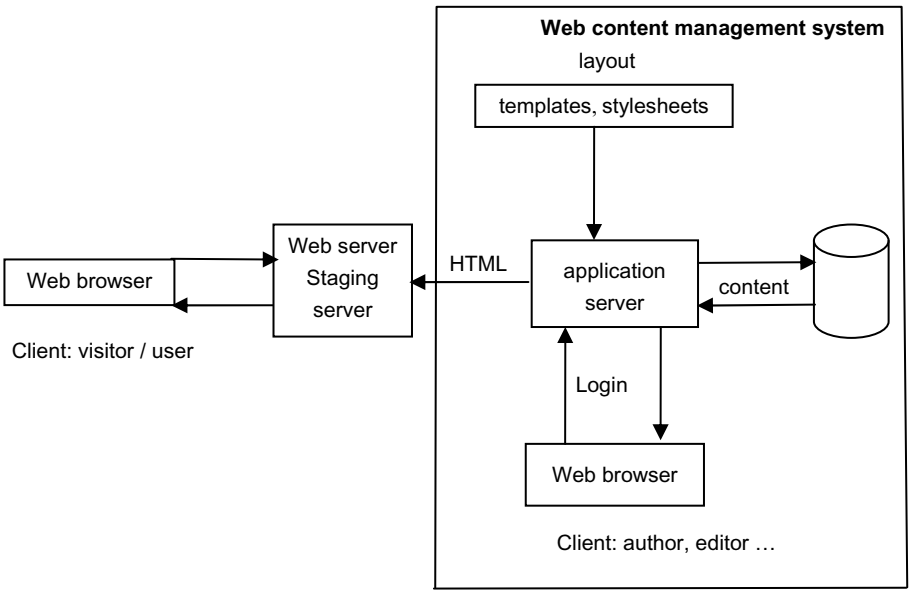


Fig. 8.18. Architecture for the staging concept

From a performance point of view this is a very powerful approach. The generation of Web pages is only done once, which drastically reduces the server's load. There are only limited requirements for the Web server, though it has nothing to do but to transfer static HTML pages. From a conceptual point of view this approach is very limited. It circumvents the dynamic generation of pages at request time and so there is no personalization at all. Content cannot be selected dynamically and as user dependent. This approach is not applicable for innovative personalized Web content and dynamic Web applications.

The live server concept (dynamic publishing approach) does not generate the pages before a user's request (Fig. 8.19). So this concept is well suited for personalized and highly dynamic content, which is often updated and refreshed. At each request the parts

of content, structure, and layout are transformed into the output format. The Web browser sends a request to the Web server which redirects the request to the application server the Web content management system. The application server retrieves the necessary data from the database (or the file system) and generates the output media, e.g. an HTML page, on the fly. Obviously this leads to a poorer performance than the staging approach. One simple way to gain more performance is to separate the application server which generates the output from the server which enables editing and process support. A more far-reaching solution is to distribute the generation over several application and database servers.

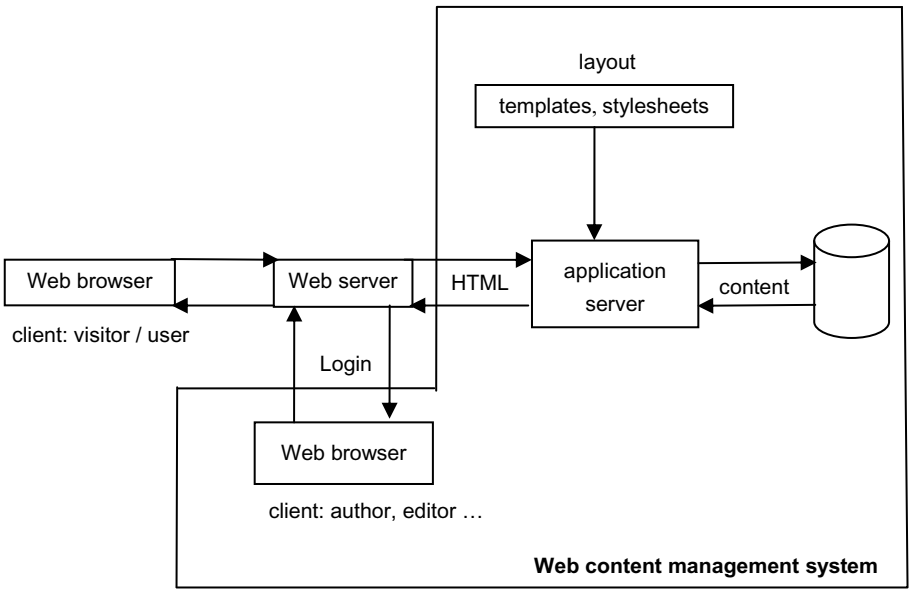


Fig. 8.19. Live server architecture

The arguments in favour of and against the two approaches have already been mentioned, but will be repeated here. Only static HTML pages can be created with a staging architecture, so this approach can only be facilitated if it is possible to create all HTML pages independent of the user's requests. But when the HTML pages must be created with different ingredients (content, layout, etc.) dependent on the user's requests, a live server architecture allowing for dynamic content generation must be used.

Part III:
Complementary Technologies for Web Application Development

The goal of this third part of the book is to introduce technologies and approaches, which enable the "synergy" between all components of the WAA and the WPA. The point of view changes from programming techniques and Web technologies to a broader conceptual view.

First of all we identify why the technologies introduced in Part II of this book are not enough to develop comprehensive Web applications (Chap. 9). One essential requirement identified in this context is the need for documentation and administration support. Registries are a class of systems allowing for this and are introduced from a technical point of view in Chap. 10. Further on we motivate the importance of organizational modeling as a means to model the role of humans in large Web applications (Chap. 11). Organizational modeling helps to define groups of users allowed to do certain tasks. Process technology is considered to provide support for information models and administration issues; it is introduced in Chap. 12. Among other things, it is demonstrated how requirements engineering for Web applications is supported by process technology and how workflow management is most suitable for complex administration tasks.

A central question is how to store information models of large Web applications, i.e. registry data, organizational information, and process data. Repository technology is a perfect candidate for that. Repository technology is described in Chap. 13.

We conclude Part III by discussing general examples of Web applications in Chap. 14. This example illustrates the whole design approach introduced in Part I, applies Internet standards and technologies of Part II, and makes extensive use of the concepts introduced in Part III of the book. This concludes the book by providing an example that integrates all the concepts of this book.

9 Why Technologies and Standards Are Not Enough

In the second part of this book, we introduced the various technologies that can be used to build Web applications. We will now move on to a broader point of view looking into the whole landscape of Web applications an enterprise is running and using. Now the main issue arises of how to keep track of what applications are part of this landscape, how they interact, who is administering them, and what happens if this application landscape has to be changed. Therefore we do not focus on just the implementation of one Web application; rather we are interested in controlling and administering the whole landscape of Web applications.

It is important and valuable that the approaches of the second part of this book focused on the development of single Web applications. However, we postulate that Part III of this book has to change the perspective and must look into the development of comprehensive Web application scenarios. We regret that this is an underdeveloped issue and is considered too little.

We will first analyze the typical characteristics of enterprise scenarios to understand the nature of the problems (Sect. 9.1). Then, we will try to isolate concrete issues that are the result of these characteristics (Sect. 9.2). In a third step, we will provide solution concepts to overcome the issues defined (Sect. 9.3). And finally, we will show how these concepts can be realized (Sect. 9.4).

9.1 Characteristics of Web Applications in Enterprise Scenarios

In the context of real enterprises, Web applications show various characteristics. We have isolated the ones which are in our opinion the most important:

- Heterogeneity of platform – as we have mentioned in the previous chapters, one of the main issues of Web applications is the challenge of the wide variety of platforms present on the Web. The application has to run independent of the client's underlying operating system, hardware architecture, or software installation.
- Heterogeneity of application functionality – Web applications offer a broad range of functionality. There is a wide variation in the services provided by applications and the requirements they have.
- High number of functional units – in enterprises, applications tend to show up not just in small numbers, but there are always lots of them. To fulfill complex tasks, there are often many small steps using separate applications as necessary.
- Distribution of functionality – applications are spread across organizational barriers like departments, businesses, or even countries. Making them work together can cause further problems.
- Autonomy and isolation – the various applications are in general not connected, but autonomous. Thus, there are many applications existing separately from each other.
- Functional overlap – due to the isolation mentioned above, there might be more than one application in use serving the same purpose. They do the

same thing, but at many times the cost, because not just one, but many applications have to be maintained.

The standards and technologies introduced in the second part of this book do not offer solution concepts to the issues compiled above. In contrast, the variety of standards and technologies is the source of many of these issues. What is missing is an integration backbone for the whole landscape of Web applications. This integration acts as a sort of glue between single Web applications that would alleviate some or most of the issues mentioned in the list above.

In this third part of the book we will introduce the means to cope with the problems identified above. We will introduce registries, organizational management, and process technology that functions as the integration and administration backbone for comprehensive Web applications. Before these concepts are presented they are analyzed in order to identify the demands of comprehensive Web application scenarios.

9.2 Issues Arising from these Characteristics

Now that we have identified the characteristics, we will clear up the issues that cause them. Luckily, some of them are caused by the same issues. Also, some of the solutions we provide cover several issues.

In today's business world, enterprises run the risk of losing themselves in a more and more confusing internal structure and have to cope with more comprehensive relationships with their business partners and customers. The organization of the company gets increasingly complicated until no one understands it anymore. One reason for this is the increased frequency of change: new employees get hired, new products are invented, or new departments are established. The question arises on who has to keep track of all these changes. If you hire more and more people, you will at some point need someone to tell you that you need to rent another building, as the company has run out of office space.

Frequent changes also cause problems in the software world. When people leave a company, their knowledge leaves it, too. It is an often encountered problem that software can no longer be maintained since the programmers are no longer available. This would not be too serious if programs were appropriately commented and documented, so that a new employee could learn how to do it. But practice shows that programs are in general not documented well, leading to the chaotic picture we have outlined before.

Tracking changes is not the only problem. If you welcome new employees into the family of your company, you have to introduce them to the rules, hierarchies, and internal structure. This requires that someone knows all the rules, hierarchies, and internal structures. These structures also include all the software tools, ranging from the check clock system to the modeling tool used for product development.

What arises from this scenario is what we call the need for documentation. The term documentation describes in this context the desire to describe how things work. It is the process of creating sufficient information that makes it possible for someone unfamiliar with the domain to understand what is going on.

Solving the need for documentation opens the door to solving other issues. Once the business is formally documented, it can be analyzed, based on this documentation. Inefficient processes like functional overlap or bottlenecks can be detected and dealt with appropriately. An example of a functional overlap is the use of two mail servers, if one would be sufficient to handle all the incoming e-mail. By handling all e-mail accounts

with a single server, administration time and thus costs can be reduced. A bottleneck on the other hand might be a software component that is overloaded. Imagine a Web shop that prints out all orders on paper for the employees who wrap up the order in boxes. If a single printer is used, the employees might unnecessarily have to wait for a new assignment, not being able to work. There are plenty of orders available, but the printer throughput is too low. As every enterprise should be interested in optimizing its performance, we think this need is a common goal for every business.

Analyzing the internal structure of an enterprise provides other benefits as well. For instance, say a new software package has to be introduced. Now the question arises whether to write it in Java or to use the .NET technology. Analyzing the documentation will provide enough information to determine the more appropriate solution. This will be the solution that can be integrated better into the existing software landscape of the enterprise. We call the need to examine documentation the need for analysis.

Another big challenge is the dynamics of structures. As time moves on, the enterprise changes. There might be new features that have to be added. Or there are new laws or regulations that require new steps in certain processes. Systems that were well designed at first can quickly grow into an opaque source of confusion, even if they are well documented. With the company structure sinking into the realms of chaos, just documenting this process might be interesting for the liquidator, but will not avoid the crash. That is why we require a system to treat changes inside the enterprise in an appropriate manner. System engineers have to be prepared to continuously adapt the enterprise's IT systems, i.e. to increase the level of maturity. This is why we state the need for the treatment of changes. With respect to the issue of documentation, changes can be separated into two categories.

Inner changes happen inside a single documentation domain. Thus the consequences implied by a change can be predicted, completely judged, and treated appropriately. Inside the documentation system, there can be rules that define how to cope with changes. Imagine a department supervisor passing on an assignment to a member of the department. The rules might imply that the assignment is printed out and put in the inbox of the corresponding employee. Thus it can never happen that someone gets punished for not completing an assignment, when they never have received it, or because the supervisor simply forgot to tell the employee.

The publication of a new version of a software environment might well be a reason for internal change. If there is a new version of the Java VM, the question arises whether it should be immediately installed or the installation should be postponed. Before the implementation can take place it has to be discovered whether all applications can cope with this new software and whether it fulfills the stability and performance criteria.

External changes on the other hand happen with changes between two documentation domains. An example is the establishment of a business connection with a new partner. As there is no common or global documentation facility, the consequences resulting from this new connection cannot be predicated. To present an example, if one passes an assignment to this new partner, one has no idea to whom you should give it, as one has no insight into the internal structures of the other enterprise.

To sum up, we have discovered three basic needs that an enterprise system should satisfy:

- the need for documentation
- the need for analysis
- the need for the treatment of change.

In the next section we will provide an overview of solution concepts for these issues. The following chapters will then detail these solutions.

9.3 Solution Concepts

Now that we have determined what the causes are for these issues, we present approaches to overcome them. We consider these solutions important enough to dedicate a chapter to each one of them (Chaps. 10 – 13). Before that, however, in this section we will provide a short introduction to the solution concepts.

9.3.1 Registry Technology

Registries are central elements of the IT system architecture of an enterprise [AaLM82]. Their goal is to store information about any hard- and software system running in the enterprise, e.g. printers, users, computers, or operating systems. This information not only is an inventory, but also stores configuration data reflecting the correct functionality; for example, it stores which ports for connections are enabled by a firewall.

A registry is self-describing. This means that a priori knowledge is not required to retrieve information from the registry. Any application can access the registry catalogue using the standard registry API. The information inside the registry must always be in a consistent state. Any application that stores data in a registry is responsible for keeping it up to date.

An important feature of registries is notification. Certain actions might trigger certain other actions that have to be treated to ensure consistency. If an employee is promoted to become a department supervisor, there are several further steps that need to be taken. The employee will probably get a new office, receive a raise of salary, and might even get a personal parking spot in the company's garage. All these actions are triggered by the promotion process. Finally, as registries supply a complete view of the enterprise, they can be used to resolve dependencies and any resulting conflicts. This can also be illustrated with the parking example. The registry can easily discover whether a parking spot is assigned to two employees at the same time. It can also figure out whether the number of parking spots is sufficient for all employees.

Another feature of a registry is to check the consequences of a system upgrade. For instance, the local database system – which is part of the actual WPA – has to be upgraded to a new version. From the registry all application programs can be retrieved which depend on this database system. Now, the administrators can check whether those applications can go with the upgrade or have to stick to the older version of the database. In the latter case, the administrators can resolve the conflict by choosing one out of many solutions. Either they reject the installation of the new database version or they set up a strategy of updating the affected application programs. From the registry they can even get pointers to the persons responsible for these application programs (Sect. 9.3.2). They can use process technology (Sect. 9.3.3) to invite those people to a meeting to discuss the situation.

9.3.2 Organizational Management

The need for documentation and the treatment of change concerns not only technical issues, but also the organizational aspects of an enterprise. Employees, customers, and other people are involved in system development, maintenance, and usage. Links be-

tween the organizational and the technical environment have to be established and documented at several levels. Questions like who is responsible for the maintenance of a certain system can be answered by querying the organizational management. The example at the end of Sect. 9.3.1 demonstrates that organizational information is part of the registry.

Besides the issues mentioned above, personalization is an important aspect of organizational management in the context of Web applications. If many different users want to access a Web application its interface must be kept very general. However, each user might have preferences on how to access this Web application. Personalization fosters this individualization.

In summary, in the context of the development and maintenance of Web applications organizational management is concerned with the following issues:

- Who is responsible for certain modules of the WAA and components of the WPA?
- How and which functionality and content do users want to access (personalization)?
- Which users are allowed to access certain functionality or content?
- How can users be identified in a Web application and, even better, across the whole Web application landscape?

It is important to understand that organizational management also takes a global perspective. Not only does the single application stand at the focus of interest, but also the comprehensive application landscape is under consideration. The following discussion will reveal that organizational management is tightly coupled with a process-oriented perspective on Web applications.

9.3.3 Process Technology

In this chapter we emphasize the global standpoint of the whole third part of this book. This perspective directly leads to process-oriented techniques which are tightly connected to global perspectives [JaBu96]. Process technology looks at applications from a high-level standpoint; it aims at the integration of single applications in order to form a connected comprehensive application system. Process technology is not limited within a single organization's boundaries – it crosses organizational units. Thus, it is related to organizational management (Sect. 9.3.2).

In the context of Web applications process technology fulfills two main purposes. On the one hand, it provides a systematic approach to requirements engineering. The idea is to formulate requirements as application processes across organizational boundaries. From these processes it is derived what (Web) applications have to be developed. This process-oriented view guarantees that (Web) application development is not just triggered by single, isolated demands, but serves global purposes. On the other hand, process technology is used to support administration and management tasks within comprehensive Web application landscapes. In Sect. 9.3.1 an example is described where a concrete activity – the promotion of an employee – triggers a whole bunch of follow-up activities. Process technology in the form of workflow management [JaBu96] is an ideal means to control such a composite scenario. It becomes obvious that such processes are working directly against a registry (Sect. 9.3.1).

9.4 Implementing the Concepts: Repository Technology

Starting with the identification of critical issues of Web applications in comprehensive enterprise scenarios, we have introduced solution concepts for these issues in the preceding section. One observation from this discussion is that all three solution concepts – registry technology, organizational management, and process technology – are closely interrelated. None of these concepts works without strong use of the other. This finding supports the idea of having a common implementation basis for the three concepts. Such an approach will foster the integrated character of these solutions. We strongly recommend the implementation of these three solution concepts on top of repositories [BeDa94].

Repositories provide a very flexible implementation basis. They follow a meta-data approach that makes them easily adjustable to distinguish application scenarios. Through this abstract implementation strategy they can best support the exchange of information between registries, organizational models, and process models. This fosters the integrated character of these solution concepts.

This chapter just provides a high-level introduction to the concepts registry, organizational management, process management, and repositories. The following chapters will discuss them more broadly and will show how they are relevant to integrate Web applications.

10 Registries

In this chapter the concept of enterprise registries will be presented. We believe that a functioning enterprise registry is a crucial step towards automating the process of systems management. Registries must be viewed as a central point of control, containing information about the system. By system we mean large IT systems consisting of many applications which need to cooperate with each other. This issue is most relevant for Web applications with their enormous heterogeneity and complexity.

The next section introduces the notion of enterprise registries and the goals pursued when utilizing registries. Section 10.2 introduces different characteristics of enterprise registries such as global scope, completeness, consistency, notification, self-description and the information model. Section 10.3 presents two scenarios illustrating the concrete application of registry to solve WPA- and WAA-related issues.

10.1 Introduction

Enterprises, nowadays, have a very complex IT infrastructure (Chap. 11). Two major issues define the complexity of Web application structures. On the one hand, the Web applications have complex architectures (WAA and WPA). Not only are complex infrastructures, i.e. many platform modules, introduced and configured to work together, but also there are many application components that must interoperate. On the other hand, complexity is entailed by the fact that there is a myriad of applications which need to be integrated, or at least used cooperatively by many users.

Systems management and support are tedious and time-consuming activities. With the increasing complexity of IT systems the effort system administrators need to put in to keep the system up and running is increasing. The real problem is isolation. Most of the current systems and applications are designed to work on their own. Firewalls represent a typical example in this respect. The firewall configuration data is specific for the concrete firewall used. The rules as to what ports allow connections are configured by the system administrators after an extensive study of what applications the different users need. It is, however, not uncommon that a newly installed application needs to perform connections to ports which are not a priori known and this is why the application does not function. The system administrators must analyze the problem and reconfigure the firewall.

This scenario is intentionally simplified. A realistic scenario will be much more complex. The goal is to have a generic mechanism allowing different applications to talk to each other and to minimize configuration effort. Automated configuration of system components would greatly minimize the system management effort. A successful example of such a mechanism is Plug-and-Play technology. The idea of having self-managing systems is not new. Over time it went through a number of phases and metamorphoses. One of the latest initiatives is “Autonomic computing” from IBM [KeCh03]. It defines the key properties and benefits of having self-aware and self-managing systems. However, these benefits are not free. A lot of time and complete redesign of existing software will be needed to achieve these features. Still, what are the key areas in an enterprise IT system where shared information is needed? Figure 10.1 provides an overview of this issue.

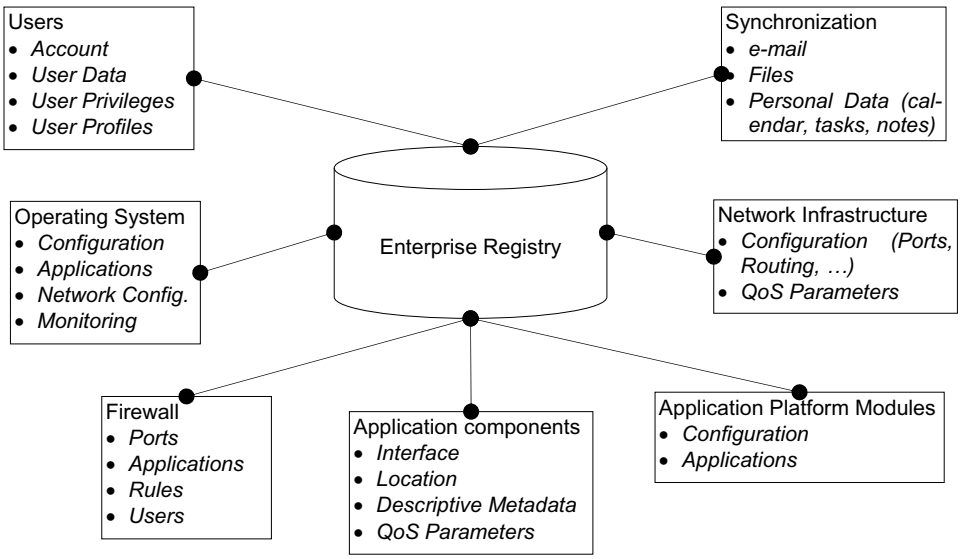


Fig. 10.1. Areas in the enterprise IT landscape cooperating with an enterprise registry

A careful analysis of the entities depicted in Fig. 10.1 would show that an enterprise registry contains various kinds of information, e.g. information about the configuration of the infrastructure, information about applications and their components, and information about the users. It is the ability to handle various kinds of data, establish relationships among them, and interact with the IT system that makes the registries so unique. These features distinguish registers from related technologies such as directory services. Historically the idea of a registry has evolved from multiple technologies:

- Directory service technology – directory services originate from the naming services for name and address resolution. The difference is that directory services (or simply directories) contain multiple properties for each entry. They typically allow for querying. LDAP [JBH+98], X.500 [DCND90], and Active Directory [AcDA04] are just some examples for directories.
- Middleware registries – middleware frameworks such as CORBA have registries about components and their interfaces (CORBA interface and implementation repository; Chap. 6). Compared to directory services facilitating the address resolution and infrastructure, the middleware registries serve for application discovery and dynamic binding purposes.
- User registries – user registries contain information about users and user credentials; they may serve as a central point of authentication. In other words, all client and server operating systems in a network and all applications authenticate users using the credentials stored in the registry. Examples are Active Directory and LDAP.

Registries are active, which is one of the properties distinguishing them from all of the above technologies, which are passive. Registries provide notification services, i.e. registers may provide feedback to the registered modules once an event occurs.

10.1.1 Goals

Before defining the characteristics of the registries and the tasks they must perform, let us state clearly the pursued goals. First and foremost registries are used to facilitate the management and administration of systems. Registries can help to reduce the time-consuming configuration efforts. They can also help to resolve dependencies between configuration parameters of different applications and facilitate the monitoring of the overall system operation and tracking conflicts.

The second goal achieved with registries is keeping a complete and up-to-date information record of the whole system. In other words, registries are used to record information about the system. A client (person, e.g. administrator, or application installer program) with no a priori knowledge of the structure of the recorded information must be able to discover it. Another goal in this context is to achieve a certain degree of integration. If, for instance, an application can discover the settings of another application then it may also be able to modify them without any intervention from the system administrator. On the one hand the whole system is better integrated; on the other hand, components become more autonomous. The registry may contain some quality of service (QoS) information about reusable application components, as well. Querying the registry data is a major requirement.

Last but not least, the goal of enterprise registries is to promote the role of humans, which has a long history of being downplayed. The term denoting such a discussion is called organizational modeling. Consider the following example: for a system administrator it is rarely important to be able to configure application X such that it can access a resource P (e.g. printer, TCP/IP port). What is much more important is to be able to configure the fact that a user U using application X may access the resource P. Organizational modeling is considered in detail in the next chapter.

10.1.2 Why Registries Are Relevant for WPA and WAA

In this section we briefly describe how registries can fit in the Web application framework (Chap. 2). As shown in the scenarios of Sect. 10.3.1 and Sect. 10.3.2 registries find extensive use in both the WAA and WPA. A registry may be used to allow applications to discover their components dynamically and therefore can be used directly in a WAA. A registry may also be used to facilitate the configuration of the platform and its maintenance. This is of enormous interest when the WPA is defined. The WAA includes an optional component called “Search and Discover”. It was introduced in the context of Web services to account for the UDDI search functionality. “Search and Discover” may be extended to support a registry. Consider for example a multimedia content delivery system (Sect. 10.3.2). A multimedia server must dynamically discover from the registry whether a codec capable of delivering the video clip in the desired format is available.

The use of registries in the WPA helps to minimize the administration and maintenance efforts of the different platform components. In the WPA context a registry serves as “glue” among the different platform modules. Once again consider the example with the self-configurable firewall. If an application server uses TCP/IP port 3719 for administration then the installer may request it from the registry, which will subsequently instruct the firewall to enable it for requests to the application server’s admin module.

A significant disadvantage of the registry technology is that no standardized support for registries is available in the context of the WPA. The degree of integration of every

product with a registry varies. Improving it is one of the challenging future tasks. A registry is to be used only when dynamic discovery or automated configuration are required. This condition is certainly true for complex systems where many platform modules or application components are available.

10.2 Characteristics of a Registry

In this section some of the characteristics of registries will be discussed. Registries are quite a versatile technology. They can be applied successfully in many areas and therefore must meet different requirements. The characteristics presented here represent a common set of features, but not necessarily all of them. The characteristics (Fig. 10.2) include: global scope, completeness, consistency, notification, resolution of dependencies and an information model.

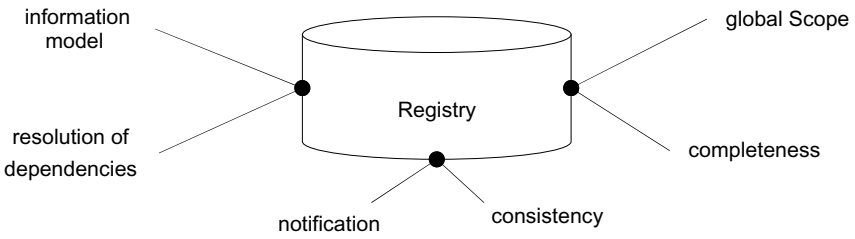


Fig. 10.2. Characteristics of a registry

10.2.1 Global Scope

Registries are meant to have an enterprise-wide scope. The goal is to use the registry as a service, provided by the infrastructure. Therefore the registry services must be tightly integrated with the OS and be available on any OS node (workstation or server connected in different ways to the system). While the advantages of such a tight integration to the applications are clear, the infrastructure itself can draw advantages, as well. These include security, resource identification, and search capabilities. Security and authentication are a good example of integration with the infrastructure. For instance, LADP [JBH+98] is not just a directory/registry where users can search and find contact information for persons; it can also be used as an authentication service storing users' credentials (Fig. 10.3).

The global scope of a registry has another dimension, which turns it into a federated heterogeneous information system. Enterprise systems nowadays store information in multiple local registers. For instance, each organizational unit – department or working group – may have a separate registry. Moreover, the data, which the registry contains, has a specific format, e.g. a database or some configuration files. All this data has to be imported and stored in the global registry. It must have a set of wrappers (transformers, adapters) to different data sources in order to import the data and data identifiers. Aggregation and data transformation functions are required to pull the data into the central registry. Organizing all the data and handling the various schemata consistently are an

essential issue. Contemporary database systems do not offer enough capabilities to handle this problem efficiently. A special kind of metadata-based systems, repository systems (Chap. 13), may be utilized to handle all the different schemata and data in a consistent manner.

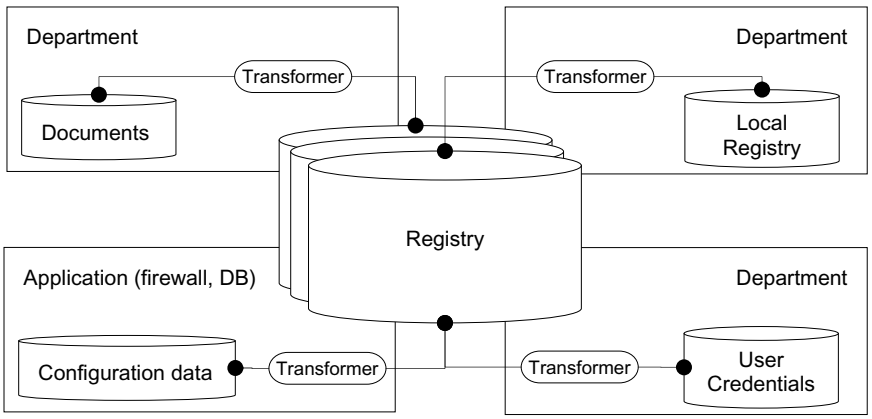


Fig. 10.3. The registry contains a global view on application data and other registries

Enterprise registries typically have a logically unified, but physically distributed architecture. The registry contents are replicated across the enterprise. The advantage is better failure management and recovery. If a node of the registry crashes it is restored from its replica. The UDDI registry (Chap. 7) is a good example of a replicated registry. It consists of multiple nodes, which are synchronized with each other on a regular time basis.

Since the registry is globally "visible" it can store user accounts and thus act as the central point of authentication. This will enable different applications, platform modules, and even whole systems to authenticate users without having separate user management. This will reduce the administration effort and increase integration between systems.

10.2.2 Completeness

Registries must have complete data on the applications, their components, and all platform modules. Generally, these are all parameters/data, which reflect either configuration parameters, or the actual state of the platform module or application component. By configuration parameters we mean all parameters that can be of interest to other applications, e.g. ports on which a firewall allows connections.

Another kind of data is the status data or state of some of the parameters. Consider for example the load of an application component. It is interesting to publish the actual workload of a Web application component in a registry. All other Web application components will then be in a position to adjust their parameters. Furthermore, if some systems are overloaded the registry information will help other applications inform users or choose alternative applications.

A third kind of data, included in the registries, is rules. Typically rules comprise dependencies, relationships, and constraints. Dependencies are special integrity rules enforced by the registry system on the parameters. For example, change the proxy server

configuration parameter of a browser whenever a change in the proxy server configuration is made. A special kind of dependency may be expressed as event–condition–action (ECA) rules. Large IT environments have to cope with many configuration conflicts. Most of them become evident with the introduction of a registry. Therefore, resolution of dependencies and resolution of conflicts are part of the core registry functionality. For example, an installer must cancel the installation of a second Web server on port 80 after being notified by the registry about one already installed as a result of conflict resolution.

Sometimes the actions that need to be taken in the case of conflict may be quite complex. They may involve multiple applications or groups of applications (cross-application boundaries), i.e. a global view of the system may be needed. For example when a new user account is created it must be propagated not only to all OSs but also to the mail server and possibly to a database management system. Similar action has to be taken when deleting an existing user account. This problem of elegantly specifying the complex set of actions and decisions that need to be made when an event occurs may be solved elegantly by workflow. All the activities that have to be performed in this situation are modeled as a process, which is further deployed in a workflow system. The "A" part in ECA simply triggers the respective process. Beyond that workflow management is a technology with a very wide applicability. It is discussed in detail in Chap. 12.

Establishing relationships among entities and enforcing constraints is a task quite relevant to enterprise registries (Fig. 10.4). For example, there is a composition relationship between users' e-mail addresses and the department's domain name. Additionally constraints may be applied on certain data. For example

- the working hours per week can be required to be greater than or equal to the company minimum;
- the maximum and minimum number of different log-ins for a user may be constrained;
- the subscription to different internal news groups or mailing lists may also be effectively controlled this way.

A much better consistency of the registry data can be ensured by using constraints, relationships, and dependencies.

10.2.3 Consistency

The registry must always contain consistent and up-to-date information. Consistency in an enterprise has at least three dimensions/aspects:

- Consistency between the registry and the data sources – the data in the registry must correspond to the state of the different data sources. If a new database user is created in the database system the change must also be propagated to the registry.
- Replication – all registry replicas must be in a coherent state. All clients using different registry nodes must actually see the same information. If a node crashes, it must be made consistent with the others through recovery.
- Consistency across data entries – upon any change the application must update the data entries in the registry and enforce the consistency rules (relationships, dependencies, constraints).

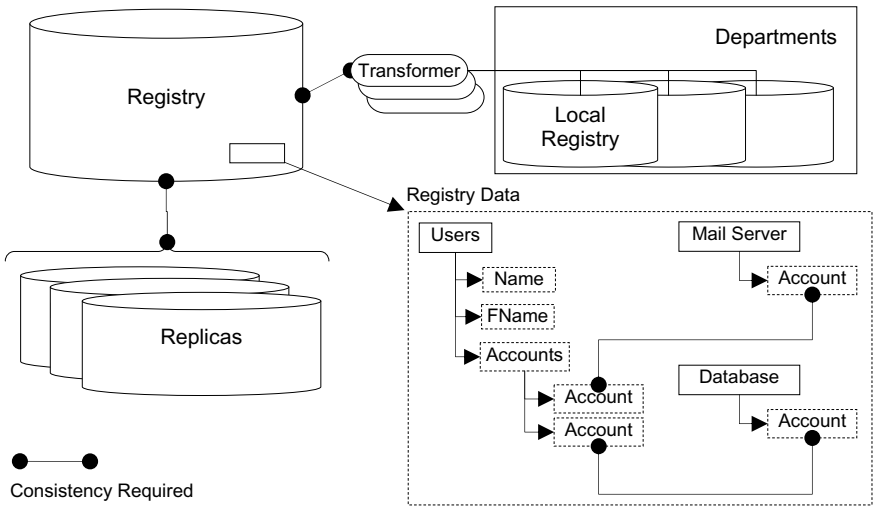


Fig. 10.4. Establishing relationship and enforcing consistency in a registry

Enforcement of consistency is an issue closely related to notification and resolution of dependencies. The registry must be able to track changes in a data source, import the respective data, and enforce consistency. Notification is the basis of change tracking.

10.2.4 Notification

Notification is one of the most challenging features of a registry. Roughly speaking, it is there to notify the registry as a data source changes, in order to import the new data, enforce consistency, and resolve dependencies. However, changes to the registry may lead to changes in other applications' data (e.g. user accounts), which will trigger another round of notifications. The registry may need to notify the data source that the data has changed, or directly write the altered data in the data source.

Implementing a notification mechanism is not trivial. It requires a mechanism to lock portions of data and track changes. Therefore a subscription for a data portion is required. To be able to be notified an implementation of a special interface is required. The interface is registered as part of a subscription. The party registering the interface is notified as the registrar calls the methods.

The principle of operation of the notification mechanism is shown in Fig. 10.5. It illustrates a scenario involving a database system registered with a registry. It associates a database user account with a user account in the enterprise system. The initial stage is represented by step 0. The database registers its notification interface with the registry subscribing for certain database data areas.

The administrator deletes a user from the registry (step 1) and therefore all the accounts the user has must be deleted. One of the accounts the user has is a database account, which also has to be deleted. This fact is expressed through relationship R1 (step 2). The registry notifies the database system to perform the respective delete operations using the registered interface (step 3). The database deletes the user and all database schemata and objects that belong to that user, and updates its configuration data (step 4).

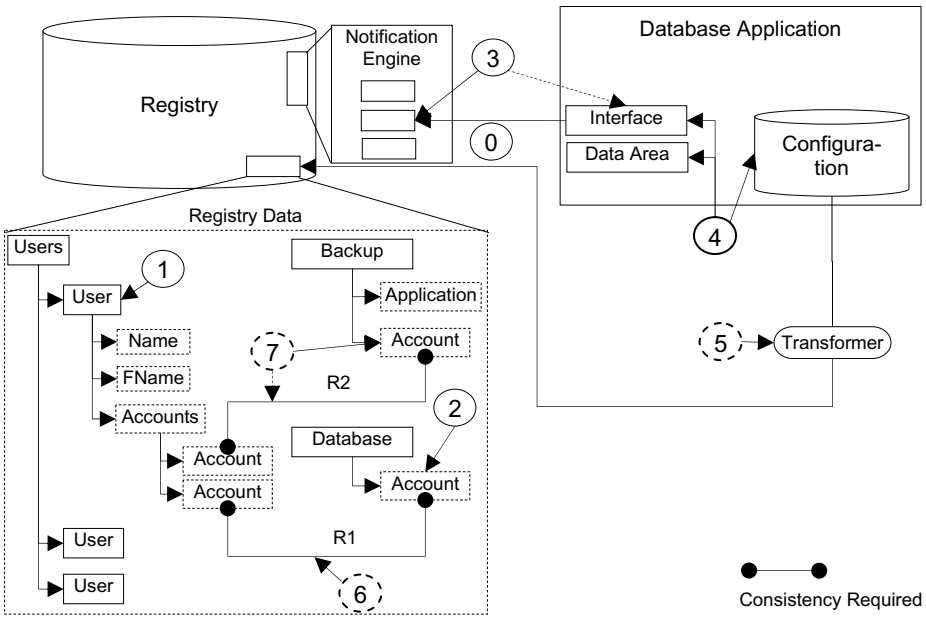


Fig. 10.5. Notification in enterprise registries

Steps 1 through 4 show how the notification mechanism is used to propagate certain operations performed by certain applications on the registry data. For completeness the scenario is extended by steps 5 through 7 (but these steps are optional with respect to notification). After the completion of step 4 the database notifies that the user has been successfully deleted. The registry updates its data by rereading the database configuration (step 5) and consequently deletes the database account. The relationship R1 is no longer valid and can therefore be deleted (step 6). The same procedure is repeated for the second user account (step 7) before the user can eventually be deleted from the registry.

10.2.5 Information Model

Having a general model of all entities "dwelling" in the enterprise registry is one of the registry's most important characteristics. The information model is a kind of all-encompassing data model. The registry's information model covers a number of aspects: data model, identification, locking and notification (subscription), self-description, rules.

All enterprise applications check their configuration data and some of their state data in the registry. Organizing all these different kinds of data into a single model is a tedious task. On the one hand, it is reasonable to assume that every application has a different model and therefore that the registry will be a large collection of many different application models. The registry must somehow cope with them, establish relationships, and enforce consistency. On the other hand, the registry must have a general model which provides a unified and structured view of the whole data in the registry itself. This model is called the enterprise registry information model (Fig. 10.6).

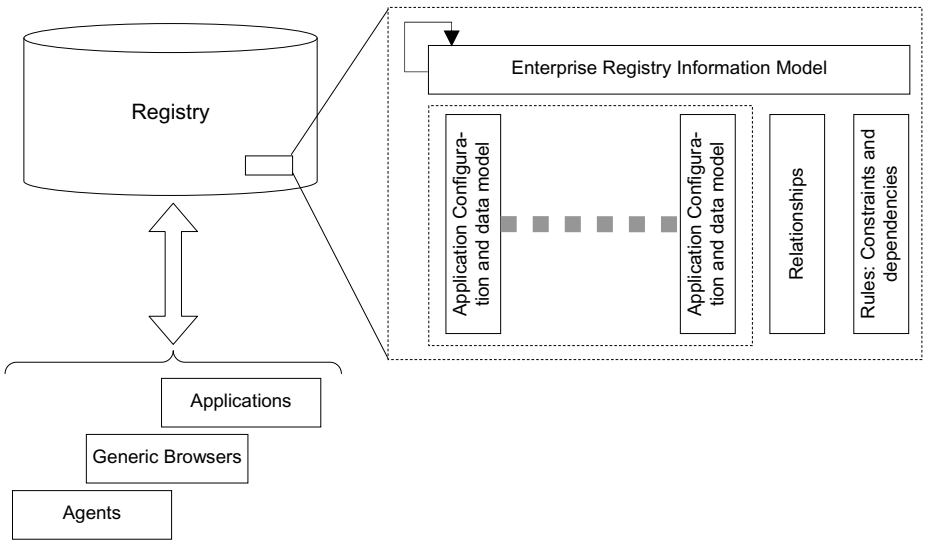


Fig. 10.6. Information model of a registry

There are only a few proposals on how an enterprise registry information model can be designed, what characteristics it should have, what elements it should consist of. Such an attempt has been made by DMTF [DMTF04], who designed a standard called CIM, which aims at modeling all components of the enterprise IT landscape. This standard maps on LDAP and X.500. It is also used in Microsoft Windows as part of the Windows Management Instrumentation technology [WiMI04].

The logical organization of an enterprise registry information model is hierarchical. Consider for example the UDDI (Chap. 7) data structures – businessEntity, businessService, and bindingTemplate for a hierarchy. Although the hierarchical organization is the best structure from a conceptual point of view, it is often quite impractical for large information models. Therefore it is considered to be a directed acyclic graph (DAG). Having well-defined and possibly standardized enterprise ontology is crucial for developing enterprise models. The term ontology (an ontology can loosely be defined as lexicon) has been introduced in Chap. 8. The way the terms of an ontology are selected and the relationships among them predefine the organization of the model.

Any piece of information registered in the registry must be assigned a unique identifier. UUID (Universally Unique Identifier) is a schema for identifiers, guaranteeing their uniqueness across registries. It was first introduced in DCE RPC and used among other technologies also in the UDDI registry.

Any data element existing in the registry must be able to take part in multiple parallel classifications. Therefore each data element will contain “meta” attributes for user-defined properties and categorization.

The registry must also support locking of certain areas. Locking has a lot to do with notification. When an application subscribes for a certain part of the models a (kind of) lock is set so that upon any the change the subscriber is notified.

Self-description is a characteristic that would allow applications to discover the format of the registry data without prior knowledge about it. These data entries would normally be organized as schemata. Self-description means to have a way of describing the registry schemata in a neutral manner, in terms of their structure. Self-description becomes vital when browsing the registry or when analyzing the application's data.

10.3 Application Scenarios

This section aims to illustrate how registries can be utilized in real Web application environments. Two basic scenarios will be distinguished. The first scenario applies to platform modules (Sect. 10.3.1). It shows how a registry can be used to coordinate the configuration of the different modules of the application architecture. The second one (Sect. 10.3.2) is less obvious and intuitive. It describes the case of applications using the registry to select components of the WAA implementing the same interface but having different performance characteristics.

10.3.1 WPA Scenario

The goal we pursue with the introduction of a registry is to make the whole WPA configurable on the fly. The WPA must react to changes in the environment and at the same time be aware of its own configuration. Briefly formulated the ultimate goal is to have a self-aware and self-manageable platform. As was already mentioned in Sect. 10.1, this is an ambitious strategy that will require significant efforts in rewriting how the different products (platform modules) "talk" to the registry. The autonomic computing [KeCh03] strategy has, however, no alternative as the complexity of applications and their platforms grows steadily.

In this section the idea of a dynamically configurable platform will be illustrated by means of an example of a next-generation firewall. Let us make two assumptions. Firstly, the firewall will combine the functionality of a conventional firewall, a proxy server, and a tunnel, thus allowing monitoring, and user detection. The second assumption is that the firewall will upload its configuration (in terms of rules, ports, etc.) into the registry and be able to automatically reconfigure (possibly safely restart) upon some changes or a notification event.

A new software program (an Internet browser) is installed in the system (step 0, Fig. 10.7). During the final phases of the installation procedure its configuration is registered with the registry. The browser's configuration says that it requires outgoing connections on port 80 (step 1). The registry automatically notifies the firewall (step 2) on behalf of the administrator that a program requests opening port 80 for outgoing connections. After being notified the firewall automatically reconfigures itself (step 3), establishing and applying the new rule. The whole action of reconfiguration is actually quite an elaborate one. In a realistic scenario it will easily span multiple programs. Telling/defining the precise sequence of steps is not trivial. Workflow management techniques must be employed to handle the case efficiently. To finalize execution of the notification event the registry updates its configuration to reflect its new set of rules. The registry then establishes automatically the relationship R3 (step 4).

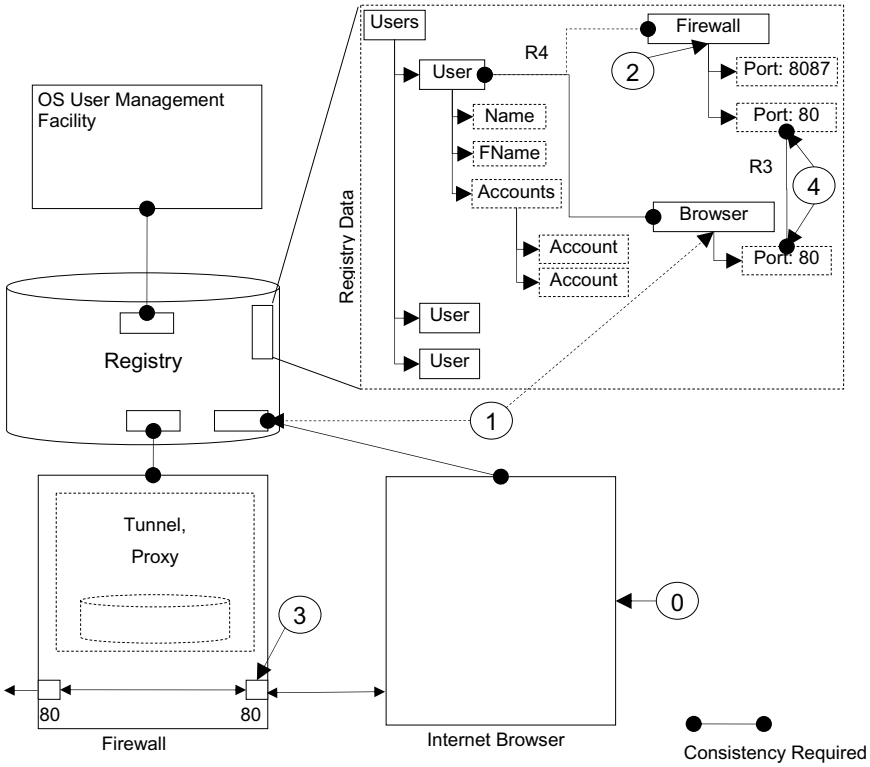


Fig. 10.7. Scenario for automatic platform configuration

What is achieved in this scenario is an intelligent way to make platform components react to certain changes in the environment. Compared to the current situation in which manual intervention from the administrator is required, this scenario represents a significant improvement. Much of it can be seen in the normal operation of the system, performing maintenance tasks and installing or removing other software.

The scenario, as described above, did not touch upon an issue critical for the administrators, i.e. involving users in the process. What administrators really want is to allow everyone to start the browser, grant unlimited browser access to intra-enterprise (the enterprise intranet) sites to all users, but to only grant a limited number of users the right to surf the Internet. Such a task is just a fraction of the much larger topic called organizational modeling, which is considered in detail in Chap. 11. So the firewall must provide intelligent access for the proper software modules on behalf of the proper user to the proper resources. By establishing the association R4 between a user and a firewall the administrators can actually configure Internet access for a user or groups of users. The firewall monitors the incoming requests, filtering only the ones coming from a certain user. This is indirectly how the administrators can implement a “canEstablishInterentConnection” user property.

10.3.2 WAA Scenario

In this section we show how a registry can influence the WAA components. This scenario is less intuitive in comparison to the typical platform configuration scenario discussed in the previous subsection. The reason why it is considered to be uncommon is because the average application would not typically need to connect to modules which are not known in advance. In other words, such a scenario does not fit the traditional architectural style. When creating large (ever) growing Web applications this becomes a necessary technique.

Let us consider the simple example of delivering video content over the Internet. Such a feature is relevant for the Web sites of news agencies, big magazines, or newspapers and therefore potentially part of a Web application.

Let us assume that the video content delivery structure (Fig. 10.8) consists of a streaming server, a pool of converters, and a number of synchronized data stores.

- The converters implement the same interface but have different performance parameters and the data stores contain video content in raw format replicated across them.
- The streaming server does not know the converters in advance.
- What converter is chosen depends solely on the video format requested by the user.

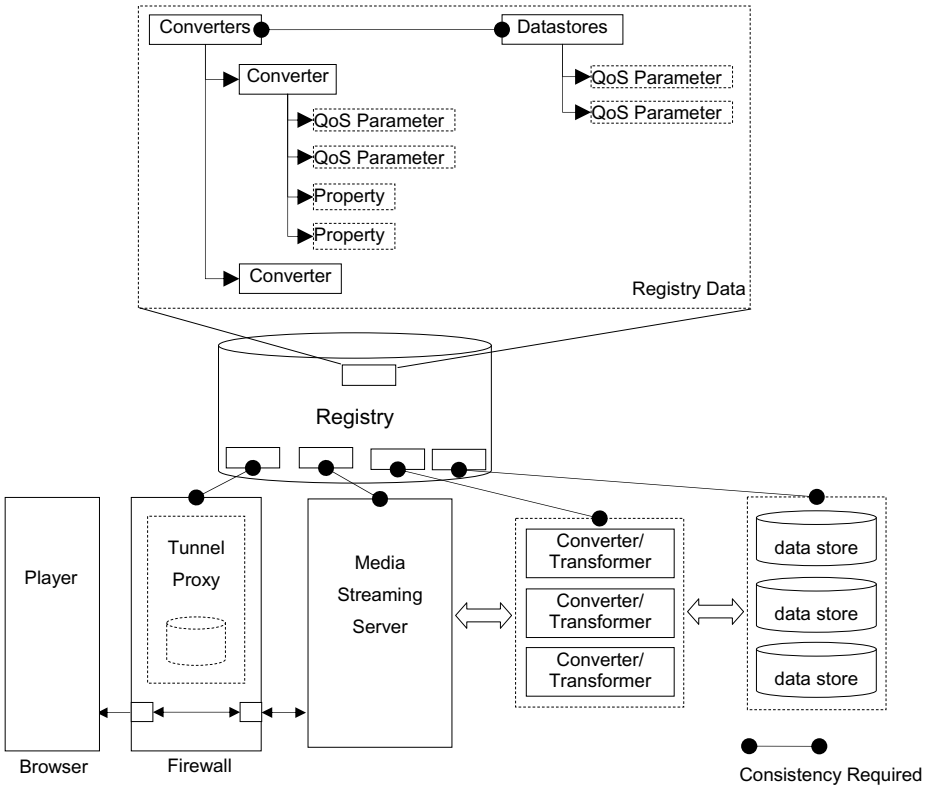


Fig. 10.8. Structure of a streaming application

The registry plays an important role in selecting the proper application component for a given task. The streaming server receives a request from a client to deliver video with certain parameters. A typical parameter set involves the format of the video stream depending on the client side multimedia/video player (e.g. QuickTime or Windows Media Player, Real Player, etc.) and the Internet connection bandwidth (56k modem, DSL, LAN, etc.). The streaming server checks whether the requested video clip can be delivered with the requested parameters and if delivery is possible it responds/confirms this to the client. As the next step it must select a converter. It queries the registry to retrieve the converter which offers the best conversion quality based on the client parameters. All converters store their performance parameters (QoS) or properties in the registry. The streaming server selects the one offering the best conversion quality, and which is also not overloaded (e.g. not currently performing/serving other conversions). Thus the media server also does some load balancing.

The selected converter starts the execution and queries the registry to select a data store not serving other converters. Since all data stores contain the same content the selection depends solely on the QoS parameters of the converters stored in the registry.

11 Organizations and Organizational Structures

In this chapter we motivate the relation of organizational issues and Web applications. Identity management, personalization, and architecture management are identified as motivation application fields. These topics are discussed in the context of the WAA and WPA and the Web application framework from Chap. 2 is extended.

11.1 Web Applications and Organizational Structures

In this book we motivate the complexity of Web applications and their architectures and discuss how to deal with this complexity. In the previous chapter registries were introduced from a technical point of view and were proposed as a possible solution when dealing with complex Web applications. One may ask why organizations and organizational structures play an important role in this context. To motivate this, we will give some examples concerning the organizational aspects of Web applications.

11.1.1 Organization and Authentication: Identity Management

Everybody knows how annoying it is to deal with numerous log-ins and passwords. This problem is not really new, but its importance rose with the spread of Web applications. Authentication is of great concern even in conventional environments within a single organization, as many applications are utilized by one user.

In an example scenario, the user must first be authenticated before interacting with the operating system. In a second step, the user logs into the mail system and then activates the enterprise resource planning software. To avoid the frustration of multiple log-in processes, single sign-in technologies were developed. These are often based on directory service like LDAP or X.500.

Due to the dynamic growth of Web applications, this situation has changed. In conventional environments, single sing-in realms were limited by organizational boundaries. Nowadays users have to create and maintain more and more log-ins and passwords. They use several traditional accounts and at the same time numerous log-ins on Web technology-based applications within their organization. To name just a few: users need log-ins for different marketplaces, both for selling and procuring goods and services. They need log-ins for different airlines and hotel chain to arrange their business trips. And they need different passwords for accessing on-line financial data and services. It summary, it can be said that identity management is one of the big challenges in Web environments.

11.1.2 Organization and Personalization

Another issue of great importance is personalization. With the tremendous increase of information on the Web, information providers must consider how to present relevant data to users and not to drown them in a tide of irrelevant content.

The following use case represents a typical example of personalization in the business-to-consumer application field. Following the purchase of several books in an on-line book store, the shopping system provides several recommendations to the customer, regarding what to buy next. These recommendations are based on the user profile created

by examining past sessions. The system might recommend books on similar topics or books that were bought by other users, who also bought the same book. The first approach is often called information filtering and the latter approach is called collaborative filtering [Pers04].

Personalization is important as well. Let us consider the portal site of a large enterprise. Information concerning not only the enterprise itself, but also competitors, must be communicated. It makes sense to select the relevant information before offering it to the user. Employees working in the sales department require different information for their daily tasks compared to employees working in the production department. Users also have different skills, which results in the need for different ways of presenting the required information. A systems engineer might prefer tables of numbers, whereas a call center agent will prefer graphical statistics.

Personalized information delivery is based on user profiles. These profiles contain various data about the characteristics and the behavior of the users. When developing Web applications, user profiles have to be understood in both an intra- and inter-organizational context.

11.1.3 Organization and Architecture Management

Everything considered so far reflects requirements only during the execution of a Web application. But organizational structures also play an important role during design, development, testing, and maintenance. The following are possible questions that could be asked by organizational management:

- Who creates the design of a module?
- Who is responsible for the implementation of a module?
- Who is in charge of the version control for this module?
- Who is allowed to use a certain module?
- Who must be informed about changes and non-availability?

Answers to these questions are necessary to develop and maintain Web applications in a well-structured, efficient, and quality-assured way. The questions also motivate how closely related registries and organizational structures should be. Information about Web applications and their architecture must be linked to organizational information to facilitate the management and to provide a complete record of such systems.

11.1.4 General Remarks

The three aspects introduced so far motivate why organizational aspects play an important role in a broad range of Web applications. Before having a closer look at the three aspects, we will provide some general remarks.

Most organizations are running more than one Web-based application. Every one of them must be documented in the sense of architecture management, needs authentication, and personalization. Therefore, organizational structures must be modeled and filled with user data, user profiles must be stored, and responsibilities between application and organizational units must be determined.

There is a simple solution to this: to collect the requirements for each Web application and to realize them straightforwardly. But this is not discussed in this chapter. The goal is not the development of isolated solutions for each Web application but concepts and recommendations for integrated solutions offering services for authentication, personalization, and architecture management for many applications throughout an enter-

prise. The advantages are obvious: user profiles are no longer limited to single applications, but they can be used for several applications. Organizational structures for architecture management no longer document only parts of an organization dealing with a concrete application but refer to the whole organization.

To shed some light on this, we will now describe the role of organizational aspects in Web applications. The following section describes the problem domain and gives general solutions by conceptual recommendations. Chapter 13 introduces implementation concepts for these recommendations and requirements based on flexible meta-schemata and repository technology.

11.2 Storing Organizational Structures

Before focusing on the different solutions to realize enterprise-wide single sign-in, user profile handling, and so on, we have to take a closer look at how organizations are structured and how they can be described. As we will see, certain requirements arise from this application domain.

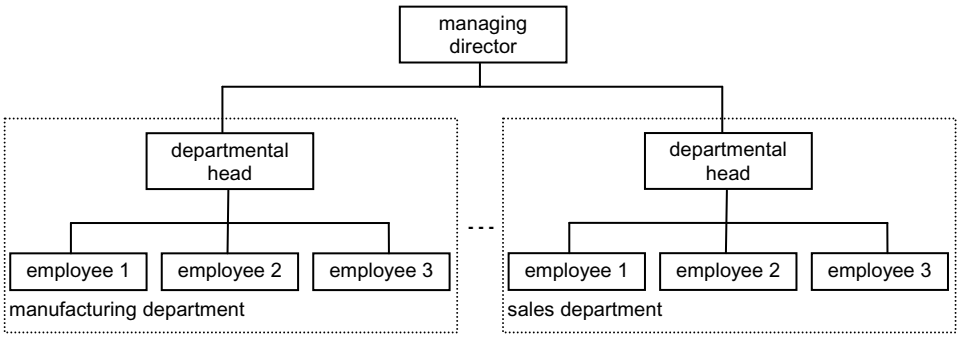


Fig. 11.1. Example for an organizational structure

Figure 11.1 shows an example of an organizational structure. The managing director is head of the departmental board and manages the manufacturing and the sales departments. Both departments have several employees. This is just one possible organizational structure out of many. It follows a simple principle: every employee has just one supervisor. Not all enterprises are organized this way. In organization theory, a wide variety of structures are discussed [Daft03]. We will only provide three common ones. Figure 11.2a presents a multiple line organization. In Fig. 11.2b, staff divisions are supplemented. And finally, a project organization is depicted in Fig. 11.2c. A project organization is an organizational structure which rarely appears on its own. It is often combined or integrated with other organizational structures.

What can we learn from these observations for organizational models in the context of Web applications? The answer is rather simple but important and often neglected. Many systems in many fields support just simple mechanisms to store organizational structures and user data. Many Web content management systems, for example, deal only with notions like users, groups, and roles (Chap. 8). Additional relationships between organizational units cannot be created. Such simplified organizational structures are not

suitable when dealing with Web applications. It is hard and almost impossible to document relationships and responsibilities between applications and organizational units in a meaningful way when the modeled organization structure does not reflect the real-world structure of the organization.

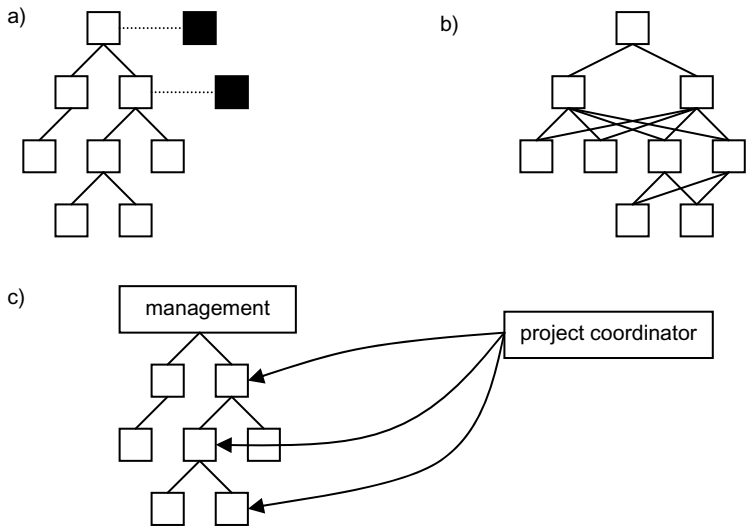


Fig. 11.2. Alternative organizational structures

The need for a flexible schema, allowing the representation of the various structures and relationships between the organizational units, is only one aspect of the problem of describing organizational structures in a flexible and extensible way. The second problem is concerned with the terminology used to describe the organization. Often general terms like user, group, or role are used to describe an organization. All employees must be subsumed under this fixed terminology. Organization-specific terminologies and vocabularies, e.g. team or task force instead of group, cannot be represented in such a restricted system.

Such a simple approach does not provide the desired flexibility of representation. Both flexible terminology and flexible structures [Buss98] [JaBu96] are needed to describe an organization in a realistic way. In Chap. 13 we show how registries can be implemented based on repository technology. Also we describe how the flexibility of schemata can realized allowing for flexible structures and terminology in the context of organizational modeling.

11.3 Dealing with Identity Management

A main issue of organizational structures in Web applications is identity management. It is often described as a collection of user-specific data that servers need to identify a user in an IT system. Therefore, an identity management service is a service that provides and administers on-line identities. Such services are not specific to Web applications. Identity

management plays an important role also in local area networks. Single sign-in there is commonly implemented using directory services. The Lightweight Directory Access Protocol (LDAP) [JBH+98] is a widespread example of such a directory service. LDAP is based on X.500, an ISO standard for directory services, and is accessed via TCP/IP. Entries are composed of attributes, each attribute consisting of a type and one or more values. All attribute values are of data type string.

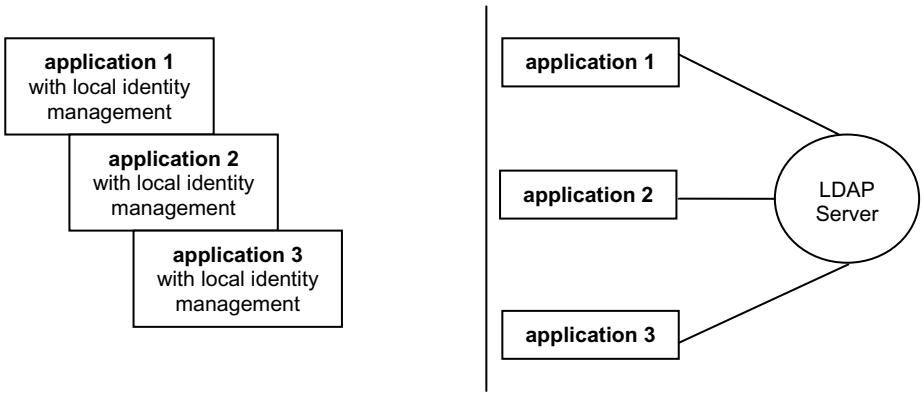


Fig. 11.3. Identity management: local and central implementation

The advantages of such a solution are obvious and illustrated in Fig. 11.3. A directory service can be used as the central point of control for several applications. User management is no longer a separate issue for each application individually. This helps to avoid the entire set of problems that arise with local identity management.

The arguments considered in the context of directory services can be directly transferred to Web applications. What differs is the scope of this application area. Identity management is no longer limited to the boundaries of one enterprise or organization. Furthermore, additional players such as business partners, information suppliers, and service providers need to be considered. Due to this change in perspective new requirements arise.

First of all, global trust authorities are needed to allow identification across organizational boundaries [BaZL03]. More precisely, a trust authority performs the following tasks. It issues or checks security certificates, checks the authenticity of digital signatures and the author of digitally signed documents, identifies other parties based on their security credentials, and cooperates with other trust authorities. In the case of a single global trust authority, synchronization and consistency are straightforward, but a lot of information is centralized. In the case of distributed or federated trust authorities providing single sign-in information, these have to be synchronized. A trust authority must know in advance which other trust authorities are trustworthy. Two different approaches – a centralized and a federated one – will be discussed in Sect. 11.5.

Another requirement deals with the security of the identity management systems. Identity information on the Web is transferred in documents or messages. These documents need to be encrypted, so that no unauthorized user can read and abuse them. Besides critical data like log-ins and passwords, other user-specific data may be present. It

is also important to identify the sender of the document or message. Digital signatures and encryption are a possible solution.

Finally, usability in the sense of ability of integration into Web applications also plays an important role [LibA04]. The techniques for identity management have to leverage existing infrastructure and technologies. Examples of this technologies are SSL encryption, URL encoding, or cookies. By using common standards and transport protocols, interoperability can be achieved. Also legal issues like data security and data privacy arise. This is a widespread field of discussion and will not to treated here.

11.4 Dealing with Personalization

When talking about organizational aspects in the context of Web applications, it is not just identity management that plays an important role: personalization is another hot topic and is the latest buzzword for content or services being delivered to users depending on their preferences. In general, personalization can be divided into three phases [ZsTZ01]. In the first phase, information about the user has to be collected and stored. This data is often called the user profile. In the second phase, this user profile has to be analyzed and information must be filtered to enable the delivery of personalized content or services. In the third phase, the Web application must be tailored to the user's preferences. This could mean putting certain content into an HTML document, changing the layout of a document, or offering specialized services. In the last phase, information available about the user is exploited to customize structure, content, and layout.

In the context of organizational aspects mainly the first phase and the following questions are of interest: What are the preconditions for the collection of user data? Which methods can be applied to collect information for the user profile and how can these profiles be stored and embedded into a Web environment?

11.4.1 User Identification and Session Handling

User data cannot be collected if the system is not able to identify the user. Therefore we consider user identification in this section. There must be a distinction between initial identification (authentication) and the identification during a session as already mentioned and explained in Chap. 5.

Initial identification (authentication) can be done by a permanent HTTP cookie on the user's machine or by a log-in form. The disadvantages of a permanent cookie are that it does not really identify a user, but rather the computer currently used by the user. For this reason other users can gain access to the cookie and the cookies can be stolen. In the case of weak security, everyone could access the cookie if they are able to log in at the computer. Therefore permanent cookies can be utilized in cases in which no security threat is posed to content and application. Alternatively, Web applications may require log-ins, which is more secure. The users must know their log-in details and passwords and, additionally, the connection can be encrypted. As explained in the previous section, this can be achieved by using a central identity management server. Two examples, Microsoft's Passport Service [Pass04] and the Liberty Alliance Project, will elaborate on this in more detail in Sect. 11.5.

The second issue is identification during a session. This is necessary because HTTP is stateless. Such information is necessary to track the users' behavior and to provide them with personalized content. The solution to this problem is simple: a unique session identi-

fier has to be assigned to each user and has to be transferred between requests. There are two widespread alternative solutions to overcome the lack of conversational state: session identifiers in cookies (Fig. 11.4, client 1) and URL extensions (Fig. 11.4, client 2).

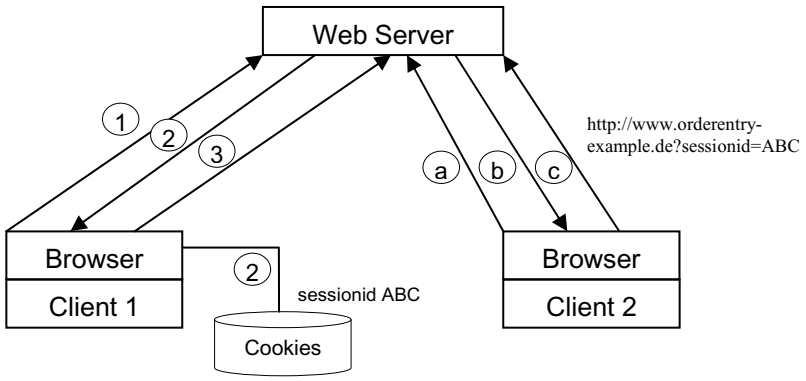


Fig. 11.4. Session handling

For the first solution a unique identifier for each user is generated after the first request (①) and stored on the user's machine in a cookie (②). This cookie is stored temporarily for the session. With each user request (e.g. ③) the cookie is read via HTTP from the user's machine and is transferred to the server. This is only possible if the users have not disabled cookie support in their browsers.

The second approach involves transferring a unique identifier via a URL. It is independent of the first one, i.e. the transfer of cookies. With the user's first request (a) within a session a unique session identifier is generated. This identifier is automatically added to each link used in the Web site or application (http://www.orderentry-example.de?sessionid=ABC) and the requested HTML file containing all the links is transferred to the client browser (b). When the user follows a link the URL is transferred to the server with the session-id extension. The server can retrieve the session-id (c) and can give a personalized response.

11.4.2 Implicit vs. Explicit Data Collection

Up to now, we have learned how to identify user requests. However, collecting data for user profiles has not been discussed so far. Such data can be collected both explicitly and implicitly. Both approaches will be discussed here [Pers04].

Explicit data collection means that users are asked for their data and preferences, e.g. by using a form during the registration process. Using such a form demographic data of the users can be collected together with preferences concerning different topics or products. There are two points in favor for this approach: it is easy to implement and the users know exactly which kind of data is stored in their profiles. But there is also a big disadvantage: the users get quickly frustrated, deny questions, or give knowingly wrong answers. Besides, legal issues have to be considered.

Alternatively, implicit data collection can be applied, where "implicit" means that the users do not actively form their user profiles. Again, two approaches can be distinguished. On the one hand, user profiles can be generated by analyzing Web server log files on a regular basis. On the other hand, the analysis can be done in "real time" just as the uses are interacting with the site. This is called tracking. The *expost* analysis is simple to implement but only little information can be retrieved, e.g. data can be collected about who retrieved which site at what time or after which page the Web site was left. Another disadvantage is that personalization cannot react to the users' behavior in the active session. User tracking, however, offers this feature. Users can be tracked at a high level of granularity, but the complexity of the implementation is higher. Each selected link or entry by the users must be documented and analyzed. This is obviously associated with a higher cost of performance.

11.4.3 Conclusion: Local vs. Global User Profiles

The discussion on user profiles for Web applications is similar to the discussion on identity management. Personalization is nowadays a state-of-the-art approach to enhance the subjective usability for the user. For the provider it allows for diversification, customer loyalty, or cross-selling activities. Above all, personalization is widely spread in e-business and more and more content-driven applications use personalization techniques. However, this applies only to locally managed profiles with all their disadvantages. Up to now, profile data has only been collected by one provider and only stored there. Similar to single sign-in mechanisms in identity management, it would be more comfortable and comprehensive both for the user and provider to undertake personalization based on common profile management. Obviously data security must be addressed when talking about such a central solution. Users must be able to decide whether or not a provider is allowed to view and use the profile or parts of it.

11.5 Solutions: Microsoft Passport and Liberty Alliance

Especially for identity management, single sign-in, and user profile management, two approaches will be briefly introduced here. Firstly, Microsoft .NET Passport [PasR04] and secondly, the Liberty Alliance [LibA04] Project.

11.5.1 Microsoft .NET Passport

.NET Passport is an initiative from Microsoft which offers a set of services for identity and profile management. The approach is based on a central storage where all data of all .NET Passport accounts is stored. Each user is identified by a .NET Passport Unique ID (PUID). The user profile consists of credentials and user profile information. Credentials are stored only within the service, whereas profile data stored in the passport is shared with the participating site, but only if the user allows this. Examples of credentials are the e-mail address, the password, the secret question, and the answer for forgotten passwords and others. The optional profile information holds the following fields: date of birth, country/region, first name, gender, last name, occupation, postal code, preferred language, state, and time zone. All these fields are optional and whether they are used is determined by the site that registers the user.

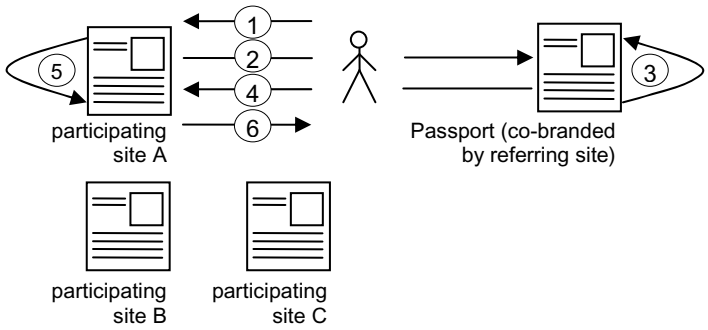


Fig. 11.5. Authentication process [PasR04]

.NET passport is based on standard technologies like cookies, SSL encryption, JavaScript, and HTTP redirects. The authentication process is depicted in Fig. 11.5. In step 1 the user visits a site participating in .NET passport and uses the authentication mechanism. The user is redirected to Passport (step 2), which checks if the user has a "ticket granting cookie" in the user's cookie file. If such a cookie exists the user has already been authenticated against .NET passport and gets redirected to site A (if the cookie is not to old and holds a time since sing-in rule defined by the participating site). If there is no ticket cookie present, the user is asked for a password and afterwards redirected to site A with an encrypted authentication ticket and profile information attached (step 4). In step 5 site A can decrypt the authentication ticket and the profile information (if the user allows the transmission of the profile) and allows the user to enter the site. Now the user has accessed the page (step 6). Microsoft .NET Passport is not indisputable. It is often objected that Microsoft has full control and access over the centralized .NET Passport service. In contrast to this approach the Liberty Alliance Project follows a decentralized approach as we will elaborate in the following subsection.

11.5.2 Liberty Alliance Project

The Liberty Alliance Project involves more than 150 organizations working together "to create open, technical network specifications for network identity" [LibA04]. The project is based on some fundamental decisions with respect to privacy issues:

- To create a decentralized architecture that avoids storing all user information with one single entity.
- To build a federated architecture where the parties are free to link their networks.
- To support permission-based attribute sharing to enable the users' control over their data.
- To provide open and interoperable specifications without central administration that can be used by many network access drivers.
- To leverage existing systems, standards, and protocols.
- To enable companies to respond to consumer interests regarding privacy and security.

As a result of this fundamental design decision, the Liberty Alliance approach differs in several points from the .NET Passport service. Passport is a concrete service offered by Microsoft whereas the Liberty Project is a set of public specifications open to a federated implementation supported by multiple service providers.

From a high-level point of view, the Liberty Alliance specification consists of the Liberty Identity Federation Framework (ID-FF), the Liberty Identity Services Framework (ID-WSF), and the Liberty Identity Services Interface Specification (ID-SIS), all of which rest on a set of standard Web technologies like SAML, HTTP, WSDL, XML, SSL, SOAP, etc. We will now discuss some of the most important characteristics of the specifications [LiAr04].

The ID-FF part enables identity federation and management by some specific features. The opt-in account linking feature allow users with many accounts to link these accounts together (if they want to do this) and to do a single sign-in at all the sites. A prerequisite is that all these sites are Liberty enabled. The simple sign-on feature allows a user to signin once at one Liberty-enabled Web site and to be quasi-logged in at other Liberty-enabled sites without repetitive authentication (protection domain).

Let us have a look at a small example [LiAr04]. User A is logged into an on-line book shop. The book shop has to know at least the user's credentials: username and password. Knowing that the book shop application can ask user A whether A wants to federate A's book shop account identity with other identities A may have with members of the book shop's affinity group. Let us assume that A wants to share this identity and allows the book shop to make the introductions. Later, A follows a link introduced by the on-line book shop to A's on-line newspapers, which is in the same affinity group as the on-line book shop. Being aware that the newspaper site is able to recognize that A just interacts with the on-line book shop, the newspaper page asks for the username and password. Afterwards A is asked whether A wants to share A's identity between the newspaper site and the on-line book shop site. If A agrees, A's identity is shared between both sites. As a result of this A can now login to one of the pages and move to the other without having to log in again.

The ID-WSF is a foundational layer and defines a framework for creating, discovering, and consuming identity services. One key feature is permission-based attributes sharing. As mentioned in this chapter, personalization is based on user data. The Liberty technology lets users decide which attribute (information) they want to share and defines protocols that enable communication between the service provider and attribute provider. The identity service discovery feature collects the user's identity information that may be distributed across multiple providers. Further, the interaction service specification defines a protocol to obtain permissions from a user. The Simple Object Access Protocol binding (Sect. 7.3) defines SOAP headers and rules for SOAP requests and responses for SOAP-based invocation for identity services.

Finally the ID-SIS is a set of interface definitions for interoperable services built on ID-WSF. Possible services are registration, contact book, or calendar. The interoperability is granted by implementing Liberty protocols for each specific service.

11.6 Integration with Web Framework Architecture

So far we have not related the three issues of personalization, identity, and architecture management to the architectural framework proposed in Chap. 2 (Fig. 11.6). Personalization is already an explicit part of the WAA. The term security is introduced in the WAA

as well. In contrast to identity management, security is often understood as the low-level toolbox providing the necessary means to realize, for example, authentication or encryption. Identity management, on the other hand, deals with the management of the identities to be secured and authenticated. This does not fit the requirements in Web applications as motivated in this chapter. A relationship between the security mechanisms and the organizational structures modeled at a conceptual level is necessary to define more sophisticated security rules, such as "allow access to sales data only to the managing director". The third topic of this chapter – architectural management and modeling of corresponding organizational structures – can be subsumed under the topic "description" in Fig. 11.6.

Fig. 11.6. WAA (cutout) from the WAA (Chap. 2)

This classification can fit the goals of certain WAA. However, sometimes the organizational issues must play a more central and important role within WAA. In such Web applications the organizational issues can become an explicit part of the WAA. Fig. 11.7 shows the WAA extended by the three organizational issues: organizational description, identity management, and personalization.

By extending the WAA, Web application designers have to explicitly consider the organizational issues in the design phase of a Web application. A possible starting point could be the modeling of the corresponding organizational structure and its documentation in a registry. In a second step, users are assigned to this structure and it is decided which technologies are used for identity management. At least it must be determined if there are any personalization requirements for the application to be designed. To document the close relation of personalization with other functionally related WAA components, we will just consider it together with the other organizational issues. These decisions have to be made for each new Web application to be developed but always with regard to reuse and integration requirements. This means that for each Web application new organizational descriptions have to modeled, new identity management technologies and new personalization technologies have to be applied, or user profiles developed. However, the goal must be to reuse existing models, data, and technologies and to integrate new models or technologies resulting from new application requirements into the existing solution. A result of this will be a central point of control for organizational issues allowing for maximum reuse and documentation of Web applications.

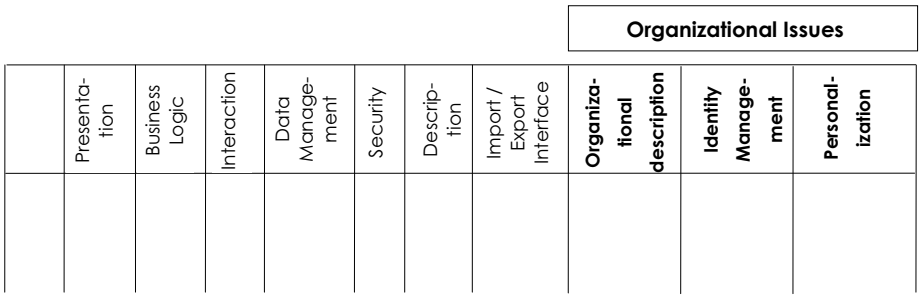


Fig. 11.7. WAA extended by organizational issues

11.7 Conclusion

Organizational issues have been motivated and an extension of the WAA proposed to consider such issues explicitly in Web applications. In the next chapter we will show how a repository can be used as enabling technology. But the organizational aspect of Web applications does not just regard the documentation as part of the registry. An organizational component is also needed to act as the central point of control for identity and user profile management. This is essential for an efficient, consistent, and synergetic treatment of organization for the Web as the application area.

12 Process Technology

Why are we considering processes in the third part of this book? The reason is very obvious. Processes are regarded as a means to integrate isolated applications from different areas. The landscape of Web applications certainly represents a complex application area.

Section 12.1 gives a short motivation of process technology. Section 12.2 then introduces the main aspects of a process model. The various usages of processes are presented in Sect. 12.3. Firstly, we regard processes as the means to find the requirements that justify the development of Web applications (Sect. 12.3.1). Here, the global character of processes plays an important role. Then, we show in Sect. 12.3.2 that processes are often needed to administer and control complex Web application scenarios. All in all, this chapter shows that processes are a very important concept for Web applications. They are closely related to organizational issues (Chap. 11) and are eventually implemented on top of a repository (Chap. 13).

12.1 Motivation and Classification

Single application programs are written to perform one or several tasks offering different user interfaces when user interaction is needed. However, application programs are normally limited to a certain scope. Processes are quite different in nature. Their intrinsic feature is that they span multiple application programs (here, Web applications) which normally span multiple organizational units. Thus, the global viewpoint of processes is accomplished.

Processes are not only comprehensive since they span multiple organizations and Web applications, but also wide ranging since they consider many aspects of a Web application and of their environment [JaBu96]. They deal with a Web application as a whole, handle the required data, define the order in which Web applications have to be executed, and determine who is responsible for executing a Web application. Section 12.2 discusses the structure of processes in detail.

Processes are described through process models. A process model can be used for different purposes. Firstly, it can be used as a means of communication. People involved in a process can use that as documentation and start reengineering efforts from it. A second use is the derivation of execution models from processes. These execution models are mostly called workflow models. A workflow comprises a process description that can be directly executed. There is a workflow management system that takes the workflow model, interprets it, and notifies the people involved to contribute to the execution of the whole workflow [JaBu96].

The distinction between process models and workflow models must be considered more closely here. In principle both models consist of the same types of modeling elements. However, the modeling elements are used differently with respect to content and form [MeBo99]. When we talk about processes – more precisely, we should talk about business or application processes but use the short form in this book – we aim at models that describe what activities are relevant and necessary in a certain application area. Process models are taken in order to provide a detailed and structured description of the ap-

plication area under consideration and can therefore be the starting point for reengineering or optimization efforts. For example, processes can be used in order to form a basis for knowledge management [JaHS01].

The precision and the degree of detail in process models are mostly not too high. The purpose here is to convey the principal structure of an application area. Thus, it is for example sufficient that a process model shows that a document "Order" is processed. It is not necessary to specify exactly where this document is stored and what format it has. Another issue is completeness. A typical application area is characterized by many exceptional cases. Although this is important for analysis to know about them, it is often not necessary to model them explicitly; they can be described very informally, e.g. by a comment. The relatively high-level generality of process models is tolerable since the users are the target recipients of process models. With their ability of interpretation they can complete imprecise and unstructured descriptions without major problems.

Workflows are derived from application processes; the former execute the latter. For this purpose, a workflow model must fulfill stricter requirements as a process model. This is due to the fact that the recipient of a workflow model is a program, the workflow management system. It automatically interprets the workflow model and so it proactively performs the workflow. Such an execution requires strict modeling elements. For instance, if an activity has to be performed that needs a certain input document, e.g. an order, it has exactly be specified where this document resides. For instance, its location is depicted by a path in the directory structure of the operating system (e.g. /projects/templates/order.doc). A workflow description prescribes a number of executable paths. If an important path is not specified, it cannot be executed. Since not all possible paths can be anticipated normally, exception treatment must be foreseen [AaJa00]; this is one of the most challenging tasks of workflow management.

In summary, (application) process models are more open with respect to content and form, while workflow models must be most precise since they are nothing but programs that execute processes. Although these two forms of processes are very different, they both fulfill important and necessary purposes. The aim of this chapter is to work out the importance of processes and workflows for our Web framework architecture. We will show how processes and/or workflows are useful to gather requirements that determine the structure and content of the WPA and WAA, respectively. We will also demonstrate how processes and, more importantly, workflows are relevant for the administration and control of the elements of the Web framework architecture.

12.2 The Perspectives of Process and Workflow Models

This section introduces a perspective-based process and workflow model [JaBu96]. It comprises the most important aspects of process and workflow descriptions. Although we restrict ourselves to introducing just five perspectives in the following, the main feature of the perspective-oriented model is its extensibility. We have learned in many projects that processes and workflows of different application areas are very different in nature. Thus, in order to be able to use one process and workflow model for more application domains, it is absolutely necessary that such a model is adjustable to different application domains.

In this section we relinquish the distinction between processes and workflows and for reasons of simplicity consider only processes. This is allowable since the discussion of

perspectives is analogous for processes and workflows. Nevertheless, the contents and the forms of these perspectives will be different as Sect. 12.2 depicts.

12.2.1 The Functional Perspective

When modeling a process first of all the different tasks involved in the process have to be identified. Let us assume that a business process for order entry management consists of the activities "analyze mail order", "process mail order", and "release mail order". In Fig. 12.1 this structure is exposed whereby the process step "process mail order" is further split up into the steps "check availability" and "enter mail order". We say that processes are decomposed into subprocesses. This concept can be applied iteratively until such a fine granularity is reached that no further decomposition is needed.

Hierarchical decomposition serves to reduce the complexity of processes. It is important that the depth of decomposition is not reduced by some system restriction but that this depth is predetermined by needs of the application.

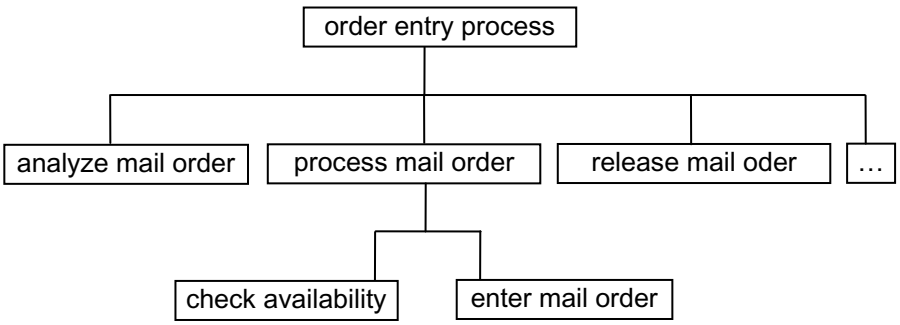


Fig. 12.1. Functional Perspective

Processes and their parts which are created by decomposition (often also called activities or process steps) form the functional perspective. The functional perspective of a process describes what the process is doing or what has to be done within the process. The functional perspective also forms a kind of backbone for the other perspectives introduced subsequently. This means that the further perspectives are all connected to the functional perspective which therefore identifies a process.

12.2.2 Organizational and Operational Perspective

What we have so far is a set of process building blocks constructed by the functional perspective. Now, two other perspectives are added which comprise the things that are coordinated by the process. This means that the process determines in what order these things are used. In a process both organizational entities and operational entities are coordinated. Figure 12.2 shows that a process model integrats both an organizational model and an operational model.

A process is associated with an organizational unit that is responsible for performing it. The simplest association is to specify a concrete person or organizational unit that has to execute the process. More complicated is to specify an organizational policy [Buss98] that determines who has to perform a process. For instance, the organizational policy

states that the person who is the manager of the project that creates the biggest sales for the company should perform the process. In Fig. 12.2 the organizational units "order entry manager" and "customer care" are associated with the process steps "check availability" and "enter mail order", respectively. By executing these two steps in a certain order the two organizational units are coordinated.

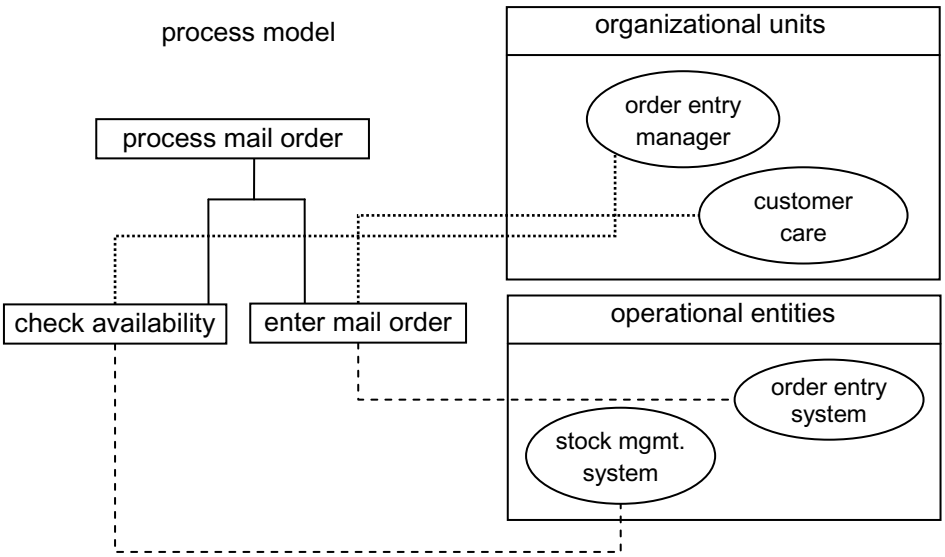


Fig. 12.2. Organizational and operational perspective of a process

The question "Who performs a task?" entails the other important question "Which applications are used to perform the task?". We call this the operational perspective. In Fig. 12.2 the stock management system is connected to the "check availability" process step, and the order entry system is associated with the "enter mail order" process steps. Again, through the execution of the two process steps the two associated applications are coordinated.

The example of Fig. 12.2 shows that processes span organizational boundaries and also integrate different applications. For our application scenario this means that processes deal with multiple Web applications that are put into a global context. The global context is represented by the process. Now, let us assume that the process is defined for a specific application area. Among other things, one can find out whether all steps of the process are well supported through Web applications or not. In the latter case, the process justifies the new development of further Web applications. We will detail this issue in Sect. 12.3.

12.2.3 Data Perspective

There is one last perspective still missing to model the static character of a process completely. This perspective is called the data perspective. It describes the input and output data required by a process step. Input data is needed in order to start the execution of a process step; the data consumed by it. Output data is produced by a process step and sub-

sequently consumed by succeeding steps. Optional and mandatory data can be distinguished.

12.2.4 Perspectives to Model Dynamic Behavior

So far we have only discussed the static aspects of a process model and how process building blocks are enriched by the organizational, the operational, and the data perspective. But an important issue of process models is to show the dynamic behavior of an application system. This means that the dependencies between process steps have yet to be modeled.

Two perspectives are used to model the dynamic behavior of a process: the data perspective and the control flow perspective. To enrich the data perspective for dynamic aspects is straightforward: the output data of a certain process step is connected with the input data of one or more succeeding steps. Such a situation means that the data producing step must be executed so that certain data is delivered which subsequently is consumed by the next process step. We call this type of dynamics the data flow perspective of a workflow.

Besides data flow, control flow expresses the dynamic behavior of a process. Control flow connects two process steps, which also determines a certain execution order. However, no data flows between these process steps. They are put into a certain sequence because some causal or temporal dependencies exist. For instance, there is a logic connection between two steps saying that when the first step is executed, the second step must be executed because of some logical reason. An example of a temporal dependency is the waiting time that has to be modeled: for instance, before a proposal for a law becomes legally binding, the proposal must be put to the public for a certain time in order to examine it.

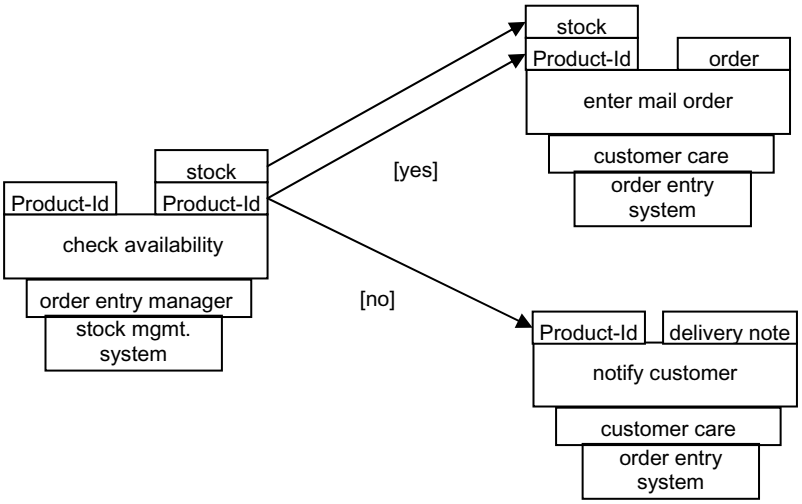


Fig. 12.3. Process example with data and control flow

We now enrich the example from Fig. 12.2 with dynamics. Figure 12.3 shows how the process now is designed. The step “check availability” controls whether enough parts

that are ordered are in stock; this is done by the order entry manager using the stock management system. If enough parts are in stock the process step “enter mail order” is executed taking as input the number of parts in stock (data variable “stock” that flows between the two process steps) and also the identification of the product (data variable “Product-Id”) the customer is interested in. Then, the activity “enter mail order” is executed by the customer care department, whereby one of the agents of that department enters the order into the order entry system. If parts are not in stock the customer care department sends a notification to the customer through the order entry system.

As Fig. 12.3 shows, the process description is quite complete. However, it might be necessary to add more perspectives which have not yet been described. For instance, a security perspective might be demanded which specifies security requirements for each step. Similarly, further perspectives can be specified. The goal is to end up with a description of the process that comprises all aspects necessary to gain a complete picture of the application domain. The next subsection shows how a process description can be used in the context of Web application engineering.

Last but not least, we revisit the discussion from Sect. 12.1. The process description of Fig. 12.3 just shows the principal idea. It must be adjusted depending on whether the description shows an application process or a workflow. The description in Fig. 12.3 might already be sufficient if an application process has to be defined. However, if the description is meant to illustrate a workflow it must be complemented with detailed information. For instance, the data must be attached with concrete data types, for the applications their API must be defined, and the organizational entities must also be specified more concretely. Nevertheless, the depicted example demonstrates what information a process or workflow model must comprise.

12.3 Using Processes in the Web Application Framework

Both processes and workflows can play an important role in the context of our Web application framework (Chap. 2). In this section we separate this discussion into two parts; firstly, we discuss the usage of processes in Sect. 12.3.1; then, in Sect. 12.3.2 the usage of workflows for the Web application framework is detailed.

12.3.1 Using Processes

Application processes are defined in order to describe a comprehensive application area (Sect. 12.1). Normally, the following steps are executed in this effort. In a first phase, the participating people of an application area are interviewed to find out the global design of the process. If it is known what has to be performed in a process (functional perspective) the organizational perspective is added. Next, the operational perspective is appended. Usually in this stage, holes in the application landscape are revealed. They show that certain parts of a process are not well covered by applications (here, Web applications). Usually, these holes have to be filled, i.e. the development of adequate Web applications is initiated.

The scenario shows that application areas are analyzed through processes in such a way that missing Web applications are discovered and their development can be triggered. This procedure justifies the development of Web applications. It prevents developments which do not fit into an already existing Web application landscape. Especially in the broad area of Web applications this is of enormous value. Due to its distributed

character application development needs a guideline that aims at the integrated development of the Web application landscape.

Through the analysis of processes missing Web applications are determined directly. According to their purpose within an application process, their functionality can be optimally derived. So, the architectures of several (missing) Web applications, i.e. the WAA for these Web applications, are resolved. Having settled the WAAs, the components of an adequate WPA can be derived as was demonstrated in Chap. 3. In summary, processes best support the purpose-oriented design of both WAA and WPA.

So far, we anticipate that each step of an application process is enacted by a Web application. Nevertheless, it can happen that one Web application is able to implement a couple of different process steps. Besides this normal case, we can identify an exceptional case. Here, multiple process steps might be implemented by one single Web application. This might be justified when process steps are very fine-grained and can well be implemented within a single Web application. In such a case, we recommend use of a special classification of a WAA (Chap. 2). The normal module business logic would then be split into two modules, static business logic and dynamic business logic. The module dynamic business logic would comprise the process-related information, whereby the static business logic would implement the static functionality needed by the process steps that are implemented in the Web application. Such a separation is valuable since it sustains knowledge of the business process in the architecture of the Web application, although the process steps are not implemented by separate Web applications.

12.3.2 Using Workflows

There are a couple of different usages of workflows in our framework architecture. The first one is very straightforward. Having identified a business process we can derive Web applications to implement the single process steps (Sect. 12.3.1). However, it might be necessary from the application point of view that the execution of the business process is proactively controlled. In such a case it is recommended to deploy a workflow management system that directly implements the business process. Concretely, the Web applications that implement steps of the process are connected by a workflow which stepwise calls the Web applications. This is the typical implementation of a business process through a workflow management system, whereby Web applications here implement the operational perspective of the process.

Another significant usage of workflows is motivated in Chap. 10 when registries are introduced. Here, workflow management can optimally be used for change management. Workflows then support administrative processes that have to realize follow-up changes in the registry [Neeb01]. As a simple example take a change in system administration (this example also involves organizational issues as discussed in Chap. 9). Assume that a certain system administrator is leaving the company. A workflow is then used to check the consequences of this change. Among other things, it looks for Web applications that are administered by this person. It then notifies those people responsible about the resulting open system administration task. These people can then assign new people to the tasks. Within a next workflow step these new people are informed about their new assignment.

There are many different sorts of administrative workflows [Neeb01]. We can name a few of them here. Many of them refer to changes of a system configuration, very often within the WPA. Then, an administrative workflow must be started to check all consequences of this modification. For instance, WAA modules which then would no longer

run must be identified and their reimplementation must be initialized. Another group of administrative workflows in the realm of the system registry is realizing what-if games. The latter are necessary in order to find out what consequences are to be expected from system modifications. In contrast to the administrative workflows mentioned above, they do not directly change the system structure. Instead, they simulate changes. For this purpose, a registry must be implemented on a powerful platform that supports sophisticated version management. This is why we choose repositories (Chap. 13) as the implementation platform for the registry.

A third usage of workflow technology is presented in the Web engineering chapter (Chap. 8). Among other things, we discuss Web content management systems there. Within the realm of Web content management systems there are at least three places where workflow management is very supportive. Firstly, we look onto the document generating process. There we describe multiple paths for the generation of a document. In principle, along these paths a document is put together out of assets, structure, layout, and logic. Due to the combination of these single tasks a wide variety of document generation paths results. Workflow management can lead through this variety and can assure that none of the steps is forgotten. For example, if a textual asset is replaced by a graphical asset, then the layout parameters of this asset must also be changed. The workflow takes care that this follow-up change happens and that it will not be forgotten.

A second usage of workflow management refers to the life cycle process of Web content management. Here, each of the phases of a lifecycle, for instance investigation, creation, and publication, can be considered as a step within a workflow. The workflow guides a user through this process and takes care that the required steps are not omitted. For example, if a new version of a Web site is published, it ensures that the former one is archived.

A third usage of workflow management concerns the publishing process. Here, the workflow management system is responsible for bringing together the parts of a Web publication consistently. These pieces are usually created by different users and show multiple dependencies between them. For example, if the structure of a document is changed, the workflow management system should inform the people responsible that the content of this site is also reconsidered. Perhaps other assets have to be associated with the Web site.

There is another area introduced in Chap. 7 that is very closely related to workflow management. Web service flow languages provide functionality that resembles workflow technology a great deal. Indeed, Web service flow languages are able to describe workflow-like processes. However, they are not able to specify the organizational aspect (Sect. 12.2.2). Merely, they coordinate the execution of functionality that can be implemented by Web applications. So they provide for a very specific and restricted implementation of workflows. Since they neglect the organizational perspective they often cannot be used to implement administrative workflows for Web application infrastructures.

In summary, processes and workflows are of great importance for Web applications. Together with a registry (Chap. 10) – as information source – and organizational management (Chap. 11) they provide a global view of this application field. They identify still open and underdeveloped parts of the Web application landscape and support its administration. The next chapter will show how these three techniques can be implemented on a common platform that allows seamless integration.

13 Repositories

In this chapter we introduce repositories as the basic technologies on which registries, organizational management, and process management can be built. The chapter is organized as follows. After a brief introduction (Sect. 13.1) two scenarios are used to motivate and address the main issues of repository technology (Sect. 13.2). Then the term metadata is considered in more detail and a distinction between structural and descriptive metadata is presented (Sect. 13.3). Finally, architectures for repository systems are introduced (Sect. 13.4) and their usage in the context of organizational and process issues is outlined (Sect. 13.5).

13.1 Introduction

The term metadata can be broadly defined as data about data. Generally speaking, the metadata describes certain aspects of the actual data: for example, its structure in terms of data formats or information as to when the data was written or by whom. Two broad kinds of metadata can be distinguished: structural and descriptive metadata. Structural metadata is a description of language constructs such as types, data format definitions, or schemata, all characterizing the structure and the semantic meaning of the information. Descriptive metadata, alternatively, refers to auxiliary data characteristics such as the last access date of a text document, or the name of the person who last modified it.

Repository systems are systems that handle metadata. Although storing, manipulating, and making data available may be done by many systems such as databases, knowledge management systems, etc., repository systems make native use of metadata. If compared [Bern97] to object-oriented databases, repository systems exhibit a number of special features:

- Repository systems serve as catalogues for data and application models (Sect. 13.3).
- Repository systems offer a set of application services on top of a database. These services include versioning, transactions, notification, etc. (Sect. 13.4).
- Repositories may serve as tools for integration.

Metadata management is a growing part of the database business [BeDa94]. Although nowadays there is a relatively small demand for metadata handling systems, it is expected that they will gain significant market acceptance in the near future. The practical expression of this tendency is reflected by the recent developments in metadata-related standards such as the OMG Meta Object Facility (MOF, [OMG02a]), OMG Common Warehouse Model (CWM, [OMG03a]), and Meta Data Coalition's Open Information Model [MDCo99], and the development of XML and the OMG XMI [OMG02b].

As mentioned above, metadata implies meta-schema; it represents the description of the structure of the data or taking another point of view – the knowledge necessary to interpret the data. Thus utilizing metadata to manipulate the corresponding pieces of data would enable different tools to operate on data without having a priori knowledge about its structure (i.e. to operate on a more general level by dynamically interpreting the specific data structure and then the data itself.) Metadata management has various important implications. Some of them will be discussed in the context of repository systems.

Repositories are systems providing a sound basis for metadata management. They are used to store and manipulate descriptions of types (or schemata) or artifacts of enterprise-wide scope. An informal definition of a repository system (or repository in short) is formulated in [Ortn99] as follows: "Repositories are systems for documenting types or schemata." Here the term "documenting" refers to the description of the structure of object-level entities (such as objects, classes, or database schemata) in terms of types (from which they are instantiated), and utilizing a development framework (relational, object-oriented, etc.) nomenclature.

Repository systems are mainly applicable to areas where a common representation of data emanating from heterogeneous sources needs to be constructed, or to fields where a complete and active description of the system would be of great use. These are basically areas where cataloguing (or maintaining a self-description catalogue) is considered central to the overall system. Alternatively, in systems which are relatively static, homogeneous, and isolated in nature (e.g. word processors, spreadsheet diagramming tools, scientific calculation environments, or even some development environments) the introduction of a repository to host the common description of development results would typically require a hardly justifiable development overhead. Examples of areas where repository systems lead to significant advances are as Fig. 13.1 suggests: data warehousing [KiRo02]; CORBA with its interface and implementation repository (Chap. 6); various computer-aided software engineering (CASE) environments; and some data modeling tools (such as ORACLE Designer, [Orac04]).

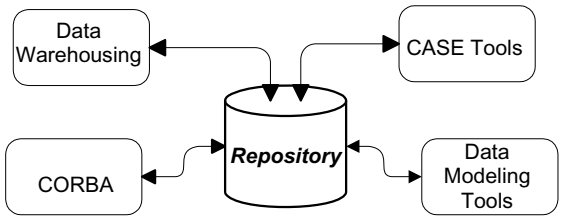


Fig. 13.1. Examples for repository usage

Typical data warehousing applications have the problem of integrating data with different format from different data sources. A major step here is the so-called ETL (Extract, Transform, Load) process. If all ETL vendors were to support a common data warehouse metamodel the ETL process would be significantly simplified and the accuracy of the data would increase. A second and much more important result would be the interoperability, which is a step towards elimination of vendor dependence. Both benefits depend on the common representation of the ETL process parameters in the form of a general meta-model.

CORBA as a middleware platform shows the need for a type management infrastructure. The CORBA Interface Repository (IR) may benefit from a metamodel because it will provide the possibility to manipulate interface definitions checked into the IR or integrate multiple IR instances.

CASE tools may certainly benefit from the use of metadata and repositories. The advantages include: design possibilities for different domains; automated code generation;

browsing capabilities; improved import/export capabilities; improved reengineering capabilities.

Data modeling tools are special kinds of CASE tools which target database systems. These require tight integration with the underlying database systems. The trouble in this field is that two distinct sets of artifacts must be managed consistently at the same time: the deployed database schema definitions and the conceptual models. If one changes (e.g. `ALTER TABLE` …) the respective models must change and vice versa. The use of metadata and repositories is also beneficial to this field.

All these application fields show an integration character. Also the design of Web applications bears this feature. Thus, as a consequence the development of Web applications can leverage repository technology.

13.2 Scenarios

Two scenarios from the wide field of Web applications justify the use of repositories. Consider the filed of large data-intensive form-based applications. These applications are tools to handle the input, output, and analysis (inclusion report generation) of data in an enterprise. In addition data may be imported from a number of secondary data sources (e.g. departmental databases) into a central database. The latter represents a typical data-warehousing scenario. Two emblematic problems crystallize after an analysis of the above scenario:

- A metamodel showing the dependency among different system modules is necessary to handle change management. This is more a registry-oriented scenario.
- A metamodel is needed to handle the import and export of data from different data sources.

13.2.1 Resolution of Dependencies and Communication

The first problem in the above example is illustrated by Fig. 13.2. A central repository is needed to record the dependencies between the database schema entities and the respective forms and their elements. This repository will contain a metamodel of the application schema and the presentation (the “forms”) elements. Once the database schema is changed and an entity (e.g. “Lines”) gets a new attribute (“lenUnits”), the definition of all forms which handle “Lines” data must automatically be changed to account for the new attribute. Such a scenario is extremely important to Web applications due to two factors: their presentations are dynamically generated and the large majority of them represent data-intensive applications. Without a repository keeping record of the different elements and the dependencies among them, the Web application would not be in a position to handle these changes.

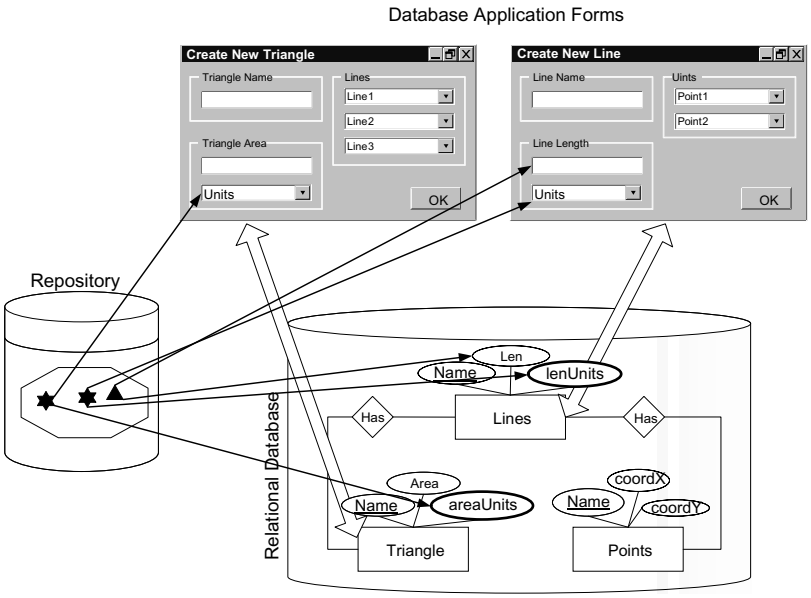


Fig. 13.2. Forms based application

13.2.2 Integration of Applications

The second problem can be reduced to the typical data transport problem (Fig. 13.3). Transporting just the data does not make much sense, because it must be interpreted at the receiving side in a special way. For this reason a converter (importing tool) for each special data source needs to be built, which is an illustration of the famous N*(N-1)/2 converter problem.

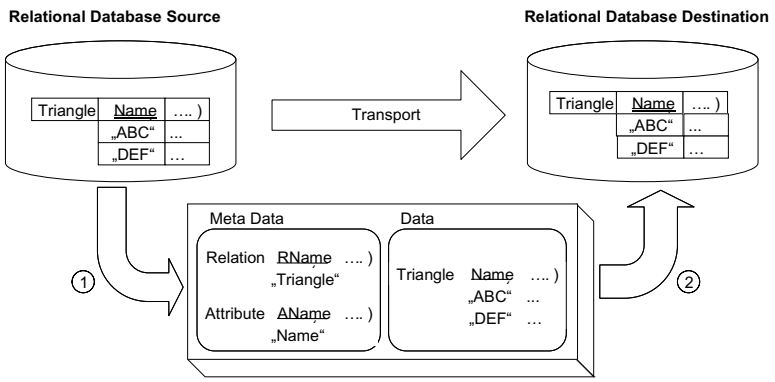


Fig. 13.3. Metadata as means to handle the data transport (import/export) problem

Transporting the data and respective metadata (both structural and descriptive) is an approach that significantly reduces the number of converters. What is required now is a single converter for a metamodel for each domain. For example, a converter and metamodel are needed for relational data; similarly a converter and a metamodel are needed for object-oriented data, and possibly an additional set for semi-structured XML data.

A repository in this case may serve as a simple translator between the data coming from the input domain and the desired output domain. As a first step the import facility, which is based on a repository, interprets the input data and metadata. The data in this case (Fig. 13.3) is the artifact `Triangle` and all instance tuples, i.e. the triangle `ABC` or `DEF`. However, the artifact `Triangle` per se is not meaningful; the importing database system does not know explicitly whether `Triangle` is an attribute or relation or data type. In order to define this explicitly one needs to transfer the corresponding metadata also. Therefore the tables `Relation` and `Attribute` are transferred, too. They explicitly define that `Triangle` is a relation and that `name` is an attribute.

13.3 Metadata

As we mentioned in the previous sections repositories are systems for handling metadata. Many of the major benefits repository systems provide are connected in many ways to the metadata. Still, we did not say how to define and use metadata. Also the properties of metadata must be considered.

We define metadata simply as normal data which describes other data. This definition has a number of important aspects raising serious questions. Firstly, the fact that metadata is data means that it can be manipulated, stored, and processed in almost the same way as regular data. Secondly, the fact that metadata "describes" other (raw) data means that it defines certain properties of the data. Depending on the kind of properties, we distinguish two categories of metadata: structural and descriptive. These are described in detail in the following sections. The third question relates to how exactly the metadata describes the data, i.e. the relationship between data and metadata. Roughly speaking, this is the type–instance relationship. An example of the type–instance relationship may be found in Fig. 13.3. `ABC` is an instance of `Triangle` and `Triangle` is an instance of type `Relation`. Last but not least, the question arises of what happens to the data when the metadata changes. In this case the data needs to be modified in a way consistent with the metadata.

13.3.1 Structural Metadata

Structural metadata describes the structure of the data and the data types or record formats in which the data is stored. Structural metadata is typically referred to as instance of a metamodel (meta-schema). In talking about metadata we should not, however, restrict ourselves just to regarding it as a fine-grained description and documentation of complex types, their structure, and relationships. More interestingly, at a metadata level one can document and handle the dependencies among different entities. For example, the data catalogue of a database system contains the structural descriptions of all relations and procedures (e.g. parts of PL/SQL packages in the case of an ORACLE database) as well as information about which procedure uses which relations (i.e. information about dependencies). Thus, if the schema of a database table changes its description would be updated, whereas the corresponding procedures depending and operating on it would have

to be identified and automatically recompiled. The meta-schema will, however, remain unchanged. Given the ever growing complexity of today's information systems a much broader and active description of the overall system with its components and the way they are interrelated with each other is here targeted.

To illustrate all the terms and they way they relate to each other let us consider the following Web service-oriented example (Fig. 13.4). We revisit the order entry Web service introduced in the example of Chap. 7. The order entry Web service is defined to have one port type called `OrdEntry`, and one operation called `pendingOEList`. The operation has two messages, `pendingOEntryListRequest` and `PendingOEentryListResponse`, as input and output messages respectively. Fig. 13.4 shows the UML notation of the WSDL service definition artifacts.

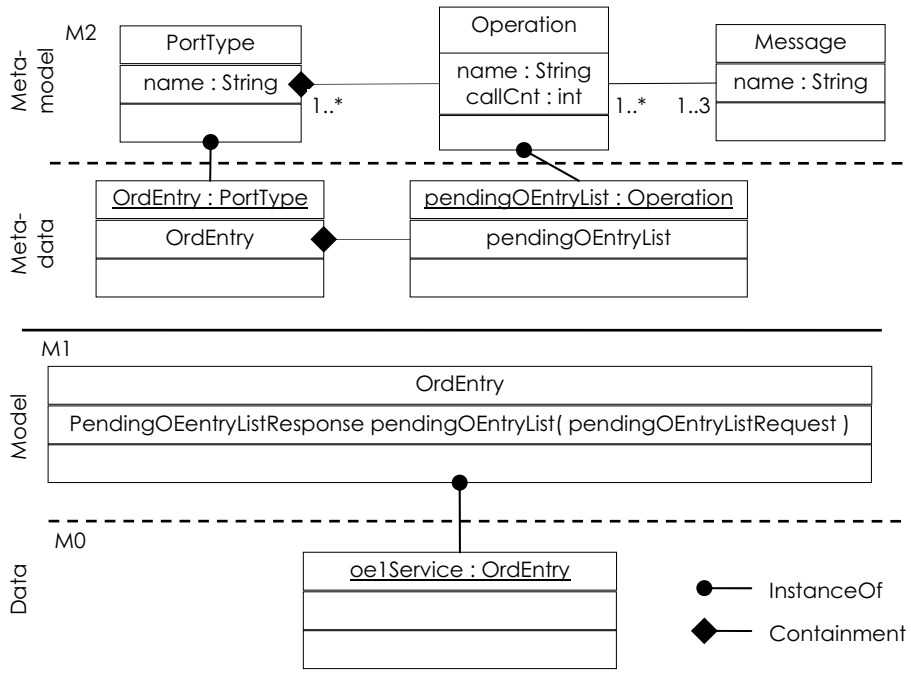


Fig. 13.4. Example of structural metadata

The actual service definition is represented by the UML class `OrdEntry`. When the service is called, it is instantiated. A concrete instance of the `OrdEntry` service is denoted by the object `oe1Service`. In this case the `oe1Service` represents the data and the `OrdEntry` represents the model.

However, the WSDL specification defines constructs (abstract or concrete part definitions) such as PortType, Operation, or Message which serve to define a concrete service such as `OrdEntry`. Therefore, these WSDL definition elements are part of the meta-model. The elements of the `OrdEntry` service definition are instances of the WSDL definitions as Fig. 13.4 shows.

As can be easily seen, the metadata (meta-objects) are instances of the meta-classes (e.g. `PortType`, `Operation`). The meta-objects define the model, i.e. elements of the

model are associated with precisely one meta-object. Types (UML classes) belonging to the model produce instance data, when instantiated.

The pair of type definitions and instances is called the language level. The metadata on a language level defines the structure of the data at the underlying level. For this reason this kind of metadata is called structural metadata.

13.3.2 Descriptive Metadata

Descriptive metadata defines non-structural properties of the data, which describe auxiliary characteristics. The simplest example of auxiliary metadata is the JavaDoc [JaDo04] comments in Java programs or the DSC (Document Structuring Conventions) comments in PostScript files. Another example of descriptive metadata from our sample scenario (Fig. 13.4) is the attribute `callCnt`, which represents a counter for the number of times the operation `pendingOEntryList` is called.

Auxiliary metadata is closely related to structural metadata, but describes user-defined or system-specific properties of the data. Descriptive metadata is therefore orthogonal to the data – in a sense that it is not as critical as structural metadata. For example, some system statistics or report generation may not function properly if the attribute `callCnt` is not present; however, the `pendingOEntryList` meta-object will still describe the presence of this operation in a port type. Imagine that `pendingOEntryList` is deleted; then to keep the data and metadata consistent the operation `pendingOEntryList` would have to be deleted, too. Auxiliary metadata is very useful in a number of cases:

- Descriptive metadata may be useful to define data properties which cannot be defined otherwise in layered architectures. For example, a Web application client may in some cases need to know some performance of the data store and storage-related parameters. These are specified as descriptive metadata and are transferred back to the client as such. Based on this descriptive metadata the client can decide what data store to use in order to execute certain classes of queries. Another example of the same issue is the system catalogue of a database system. It contains information regarding data store parameters, which should not be available due to the layered ANSI/SPARC architecture.
- Descriptive metadata can be used to define certain user-specific or system-specific properties. The `callCnt` attribute (Fig. 13.4) is an example of a system-specific property. Auditing parameters are an example of user-specified properties.

In brief, repository systems are systems that manage metadata. We distinguish two kinds of metadata: structural and descriptive. Structural metadata is used to define the structure of the data. Structural metadata concerns definitions of data such as format, records, etc. In contrast to structural metadata, descriptive metadata specifies auxiliary properties such as last access date, or the username of the user who modified a data record the last time.

13.4 Architecture of Repository Systems

The architecture of a repository system has two aspects. On the one hand, we need to distinguish the logical organization of the repository metadata (also called metadata architecture or layered metadata architecture). On the other hand, we have the repository

system architecture, which defines the functional architecture of a repository in terms of modules.

The separation between the architecture of the data and that of the system is not new. Consider for example database systems. They have the famous ANSI/SPARC architecture defining how the database data and table definitions are organized into external, conceptual, and internal schemata. Database systems are also defined in terms of their system architecture comprising database management system and data files. Further, the database management system comprises an SQL parser, query optimizer, query executer, recovery manager, transaction manager, etc.

13.4.1 Metadata Architecture

The metadata architecture of a repository system consists of four layers. As already pointed out in Sect. 13.3.1 a layer consists of types and their instances. The instances on one level are associated with types on the underlying level etc. Metadata architecture comprises the set of all layers.

OMG MOF is the standard enjoying the widest industry acceptance. OMG MOF is implemented in a couple of repository products, e.g. [Unis04], [Adap04]. MOF and metamodeling techniques will become part of the UML 2.0 specification.

The MOF metadata architecture is shown in Fig. 13.5. Four layers (language levels) can be easily distinguished: M0, M1, M2, and M3. The layer M0 contains the application instance data. M0 is not considered to be part of MOF or any other metadata standard. Examples of M0-level data can be seen in Fig. 13.4.

The M1 level defines the model from which the M0 data is instantiated. M1 is also called an information model. Examples of the M1 model are a UML class diagram modeling the classes of an application, an E/R diagram of a database schema, or a UML component diagram for a component application.

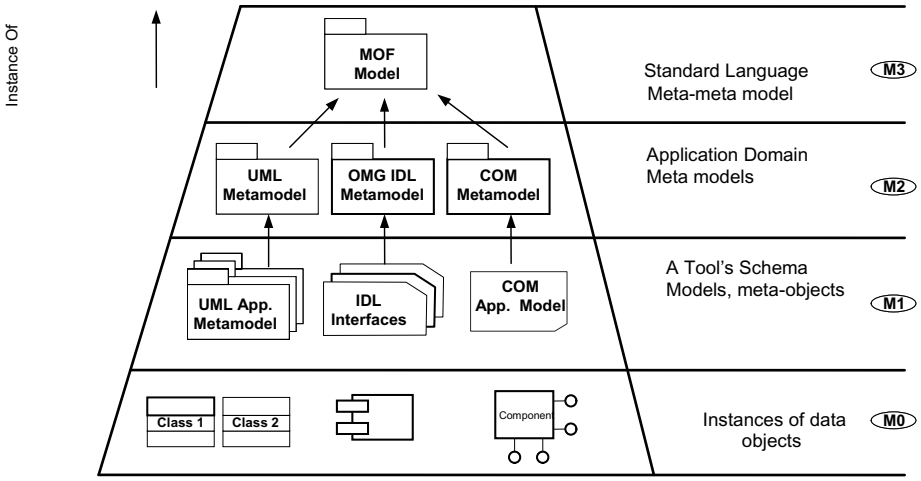


Fig. 13.5. Metadata architecture of a repository system

The M2 level defines the so-called metamodels. Metamodels define the structure of a whole domain. For example, on the M2 level one can find the UML metamodel defining

the structure of all UML application diagrams. Another example of an M2 model is shown in Fig. 13.4: it is a part of the WSDL metamodel. A repository system can manage multiple M2 models. All M2 models are instances of the MOF model. Any M2 model may have several M1 instance models.

The structure of all M2 models is described by the MOF model. The MOF model defines constructs such as package, class, operation, attribute, etc., which can be used to define the M2 model. These constructs represent the so-called abstract language (meta-meta-language). The MOF model is subject to standardization by the OMG. MOF guarantees at least syntactic interoperability if all application developers stick to MOF when developing their M2 models.

At present there are several different metadata architectures. They differ not only in the numbering conventions but also in the number of layers. Examples of such architectures are present in different metadata-related standards. These are:

- ISO/IEC IRDS – Information Resource Dictionary System [ISO90]
- EAI/CDIF – CASE Data Interchange Format [CDIF04]
- PCTE – Portable Common Tool Environment [WaJo93]
- OMG MOF – Meta Object Facility [OMG02a].

13.4.2 Repository System Architecture

In this section we briefly describe the architecture of a repository system (Fig. 13.6). The major modules of the architecture are the repository management system (RMS) and the data store. The RMS represents the repository in the same way as a database management system represents the database. It implements all metadata management functionality and offers a set of repository-specific services.

An application that uses the repository system communicates with it over the repository API. It provides an object model to handle the repository metadata and a set of APIs for the repository services.

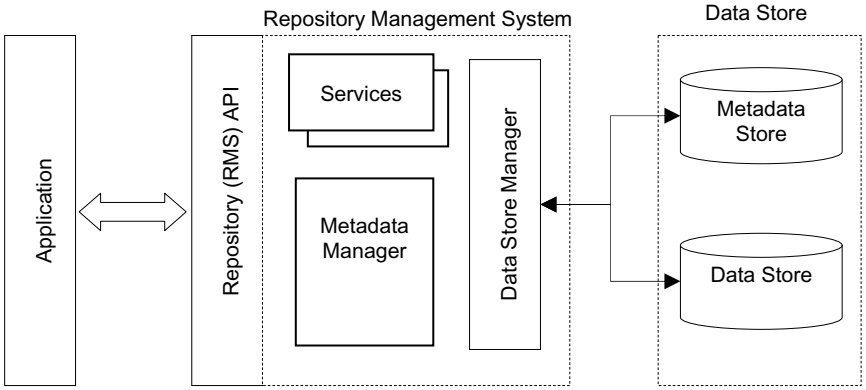


Fig. 13.6. Architecture of a repository system

The metadata manager (Fig. 13.6) is the module which manages the metadata according to the repository metadata architecture. It provides CRUD (Create, Read, Update, Delete) operations and navigation operations.

The RMS provides the repository applications with a set of services such as import/export of models and metamodels, versioning, transaction support, security, etc. Import/export of metamodels is important for the exchange of information such as models, metamodels and the respective data between repositories. Versioning allows different versions of an object to be stored in a repository. Transaction support allows for reliable operations. Features such as reliability and multiclient operation are the driving force for using transactions. A repository – like a registry – is used by many applications; it also stores sensitive data. Therefore, security is an important issue.

The data manager handles the persistence of the repository metadata. It stores the metadata in the metadata store and loads on demand. To increase the performance the data manager also implements some caching functions.

The data store represents a persistent data store for the repository data. Conceptually there is a need to distinguish between a metadata store and a data store because the metadata and the data have different properties which are reflected by different storage structures: the data tends to change much more frequently then the metadata; the metadata has smaller volume. In order to reflect this difference it is preferable to have two data stores.

To recapitulate, in this section we outlined the two repository architectures and discussed the repository system architecture. We also discussed the repository metadata architecture, which is organized in a layered manner, where the models on each underlying layer are instances of the model on the next higher level.

13.5 Repository Systems as Foundation for Registries and Organization Modeling

Repositories can be used in the context of registries to store the registry data. Every application which checks data in the registry must also check in the schema for the configuration data. These are stored as M0 or M1 data respectively. If whole ranges of application data need to be stored then the administrators may define an M2 schema.

If all application schemata are stored in a repository then it is also easy to establish relationships among the different entities as described in Chap. 10. A repository will manage these relationships automatically. A notification mechanism is part of the RMS API and can therefore be used as a basis of the registry notification. Additionally, repository systems have extensive constraints and triggers which complement the notification service. Search and discovery functions of a registry may be implemented on top of the RMS API. The type–instance relationship on which the metadata architecture is based may serve as a basic navigational mechanism. Additionally, the search and discovery API is based on the descriptive and structural repository metadata.

Organizational models and process models are checked in the repository as normal M2 models. They are created by a CASE tool and imported into the repository. Once they are there, applications can create instance artifacts and establish relationships among them. To do so an application uses the RMS API functions. In this context an application is actually the registry. It not only stores data in the registry but also provides the necessary functionality to the outside world. For example, a registry provides a search and discovery API to the applications but stores the data in the repository.

The preceding discussion justifies the repositories as an effective part of a Web framework architecture. A repository can play the central knowledge base of a comprehensive Web application, and together with the means introduced in this third part of the book functions as an administration backbone for the whole Web application landscape.

14 Putting It All Together

In the last four chapters of the third part of this book, we introduced registries, organizational management, and process and repository technology. In this chapter, we will provide an example scenario to prove that Web applications benefit from these approaches. We will show how the programming concepts introduced in Part II and the concepts presented in Part III can be used in a complementary and synergetic way to design, build, and maintain Web applications in an enterprise IT landscape.

14.1 The Scenario: the Order Entry System

Before we go into technical details, we will set the scene first (Fig. 14.1). In this chapter we consider the on-line store of a large mall chain. Detailed requirement analysis shows that this store's Web application must be based on an order entry system. Once a request is placed using the application's Web interface it is put into a queue. The order entry system takes an order from there and then processes it.

During processing the order entry system determines the closest mall according to the user's address. The next step is to inspect the mall's inventory and to determine whether all articles are in stock. If the articles are not in stock then the missing quantities of the ordered articles must be delivered from other malls and the user must be notified about a potential delay. If the right quantity of all ordered articles is in stock the order entry must be processed, which involves updating all data, issuing an invoice, billing the customer, and triggering the storage system. The last step of the process is packaging and delivery.

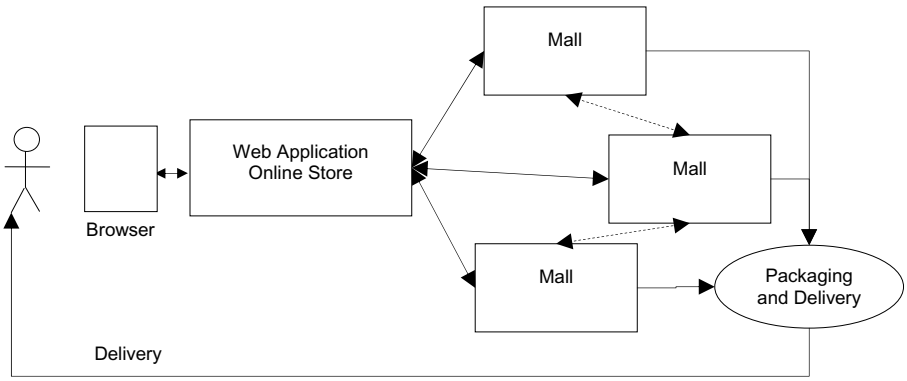


Fig. 14.1. Overall scenario

After having described the rough idea, we will now go into detail on the architecture of the Web shop. The architecture is a classical four-tier architecture (Fig. 14.2). There is a firewall granting security by shielding all internal hosts from unauthorized access from the outside. Secondly, there is the Web server which delivers the requested Web pages via HTTP. The Web pages in turn are generated on an application server. There is also a database that stores all orders placed, product information, and stock data.

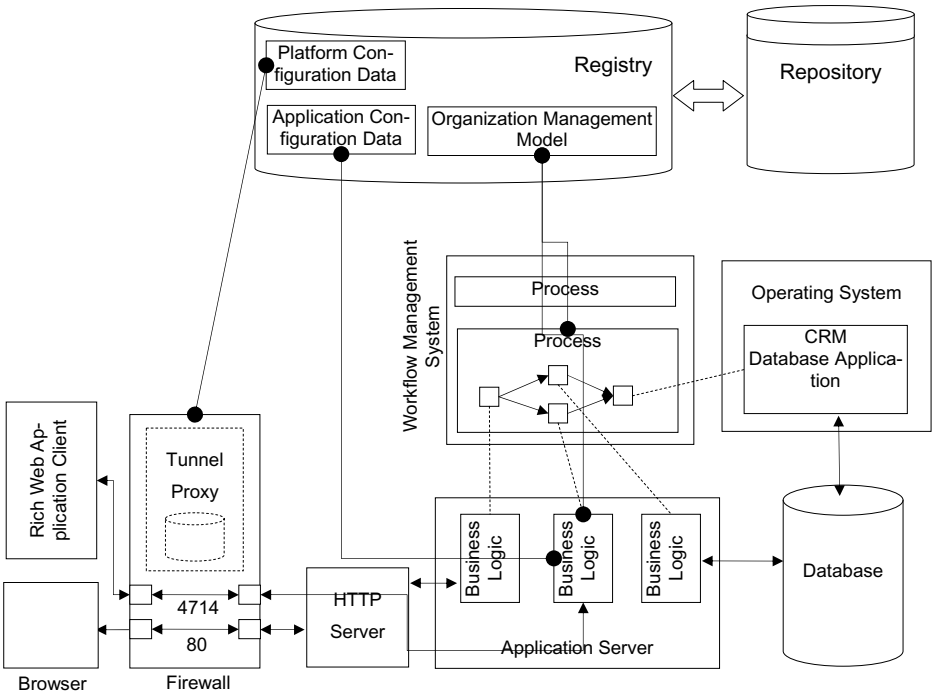


Fig. 14.2. The order entry application

14.2 The WAA

Let us revisit the approach for designing Web applications introduced in this book. We consider the example from the previous section. The initial step is requirements engineering and rapid prototyping. Let us assume that it is successfully completed and let us continue with the second step – the design of the WAA.

A bird's eye view of the on-line store application architecture shows that in principle it is similar to the application architecture presented in Sect. 3.3. The presentation is HTML based (Fig. 14.3). The HTML interface is generated by using server side scripting techniques. Therefore presentation-related logic is needed at the server side. A large part of the business logic is implemented in two separate modules. On the one hand, we have the primary (elementary) business logic modeled as classes in the business logic package. It will eventually be mapped to components. This mapping is performed when the respective Internet standards and technologies are chosen.

The second and more significant part of the business logic is the use of processes to coordinate the rest of the business logic (Chap. 12). The process component (Fig. 14.3) coordinates the execution of the single business logic components. The process controls multiple execution aspects. It has a global view of the whole application and therefore it

can determine what business logic components are executed and the sequence of their execution. In addition it controls what pieces of data are passed to which components.

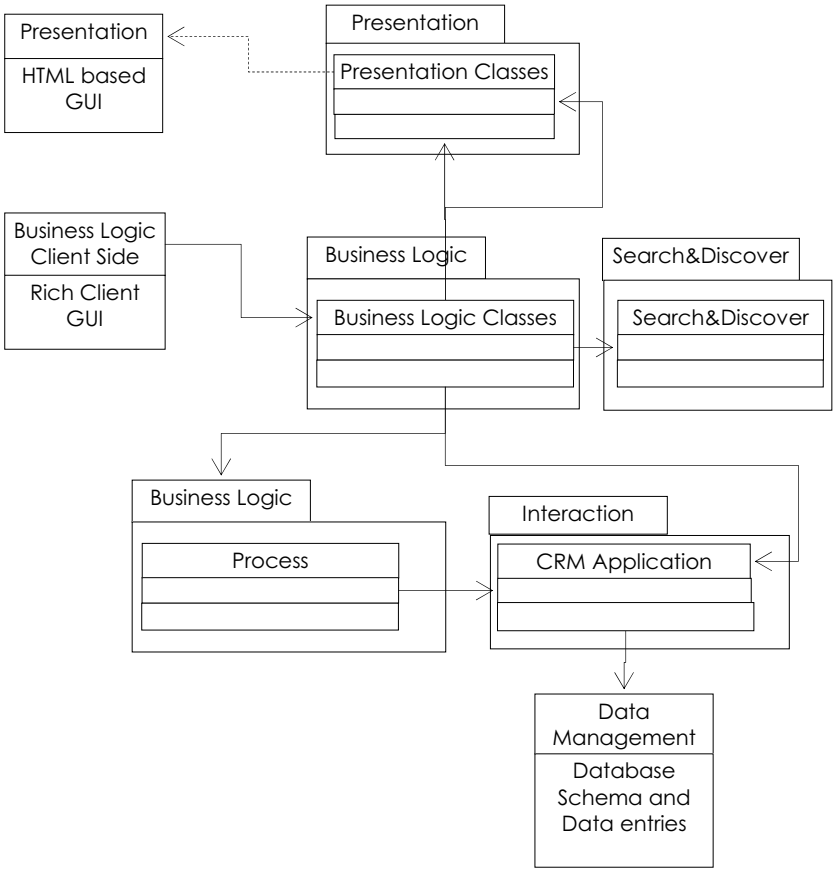


Fig. 14.3. The WAA

The process is also in a position to account for the organization of specifics. These are introduced during process modeling as an organizational aspect. It defines what persons, groups of persons, or roles are in a position to execute certain tasks. This is especially important in the above scenario. Only by means of an organizational model can it be specified that customers may reject already placed orders; or that an employee working as a packaging controller may acknowledge that the customers' order is already packaged and ready for delivery. Last but not least, only employees working as delivery persons can acknowledge the successful delivery of the order and deliver an invoice. Organizational modeling is critical in such complex business scenarios where people are involved with different roles and must be granted the privilege to execute certain tasks. Accessing the organizational structure is done implicitly by the process (Chap. 12). Resetting the business logic may use the search and discover interface to do it.

The search and discover package handles the functionality used to discover application components. This may be important when searching the nearest mall to the cus-

tomer's location and letting the package execute the customer's order. Another case is when an article is not available in this mall and has to be ordered from other malls. Using the search and discovery interface the application can list all available malls and place an inquiry about the article. These cases can only be handled efficiently when a registry is used to record all available malls. The search and discovery interface handles the communication with the registry as mentioned in Chap. 10.

14.3 The WPA

The WPA comprises four tiers (Fig. 14.4). This results from two facts. On the one hand, the business logic of the application is expected to be rather complex. A robust approach to implement it is to use components which would require an application server tier. Implementing the whole logic as part of the Web tier may not be an appropriate solution due to scalability issues.

On the other hand, the scenario may require support for rich clients, which will directly (remote invocation) communicate with the business logic. This is a typical case for distributed computing. Contemporary technologies require the use of an application server tier in this case. An additional argument in favor of this approach is the use of workflow technology, which will be introduced when analyzing the WAA. The workflow management system will also need to run as part of the application server.

The general procedure for designing the WPA is described in detail in Sect. 3.4, therefore we will not repeat these steps here. The client side platform consists of an operating system, Web browser, and an execution environment for the rich-client application. While the browser handles the Web presentation in terms of HTML pages, the rich client offers richer presentation capability in terms of GUI and additional capabilities in terms of local data store and communication.

The middle tier handles the HTTP communication and generation of the application presentation. Therefore the HTTP server and the scripting environment are located in the Web tier. The firewall provides additional security. The application server tier contains the business logic container for the workflow management system (Fig. 14.4). Last but not least, the back-end tier contains the CRM application and the database system used to store the order entry data and queue the incoming request. All components on this tier must be connected to a registry, whose role will be discussed in the next section.

In this section we will extend the order entry example. The mall IT system needs to have some supply chain management features. It might happen that the store is running low on a certain article. The order entry system has the task to determine the quantity of an article to order and to notify automatically the person in charge. The next step would be to negotiate the best prices for the new quantities. After approval by the employee in charge, the supply management system negotiates the delivery. This supply management functionality should not be implemented from scratch, as a component-based implementation of such a system already exists. Therefore, the main challenge is to integrate the existing system with the supplier's infrastructure. The solution we will develop in the next paragraphs has to meet one important requirement. The suppliers must be allowed to offer their interfaces using Web service technology. In the following, we will first consider the extension of the WAA based on the existing WAA of the order entry system. Afterwards, we will adopt the WPA to the new requirements that come up with the integration with the supplier via Web service technology.

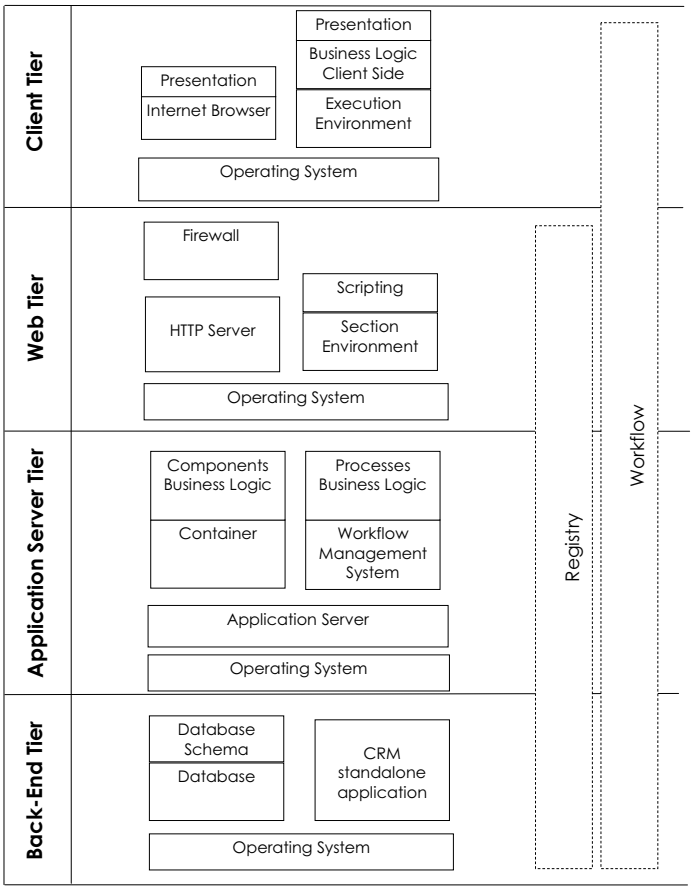


Fig. 14.4. The WPA

Let us first consider the changes to the WAA. To reflect these changes, we introduce two additional components (Fig. 14.5). The `Wrapper` class acts as an intermediary between the business logic and the supplier interface and calls the respective operations on the business logic in a definite sequence. It can also convert some of the data formats if needed. The second component comprises two classes: the `supplier` class and the `delivery` class. They serve as a representative for the suppliers' real systems (the mall supplier and the delivery company) and model these entities as abstract partners under the assumption that all suppliers and delivery companies will implement the same interface.

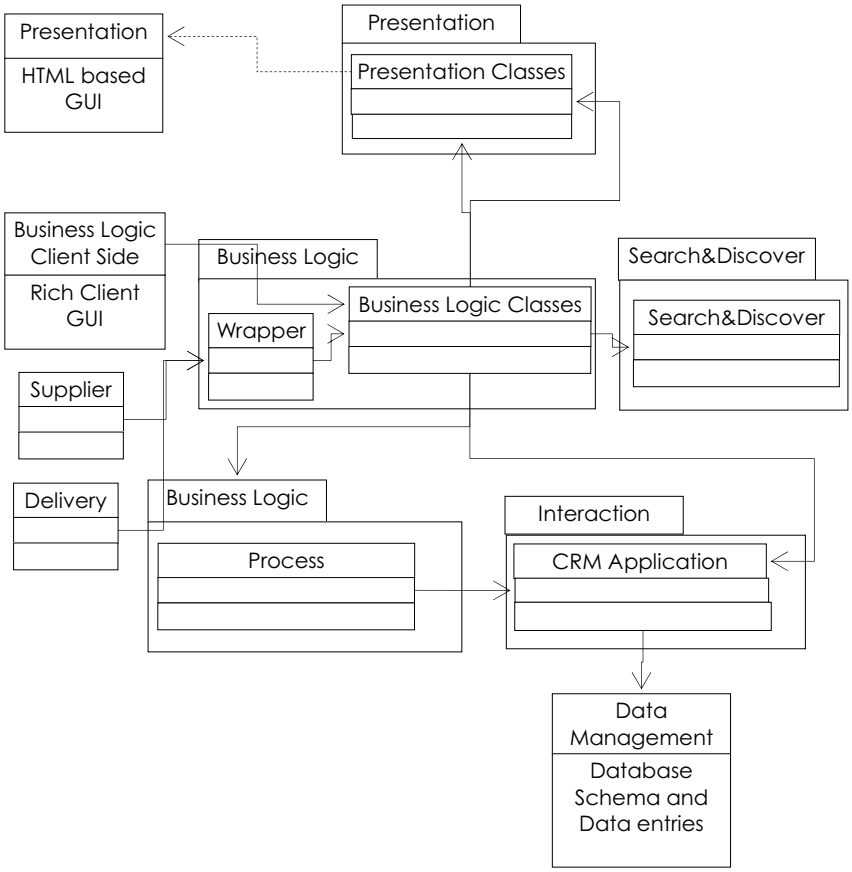


Fig. 14.5. The WPA

Although the former system offered some means of notification, e-mail notification is considered a standard option nowadays and must be implemented. And that is exactly the point where we have to consider the WPA. To enable e-mail notification, an SMTP service is needed. This additional requirement changes the WPA and introduces an SMTP server. We already mentioned that suppliers offer their interfaces using Web service technology. Therefore the designer of the mall's IT system must include support for Web services rearranging both the WPA and the WAA. The WPA must be extended with Web service infrastructure, which includes SOAP router connectors to different WPA modules such as component container or a HTTP server. The new WPA is depicted in Fig. 14.6.

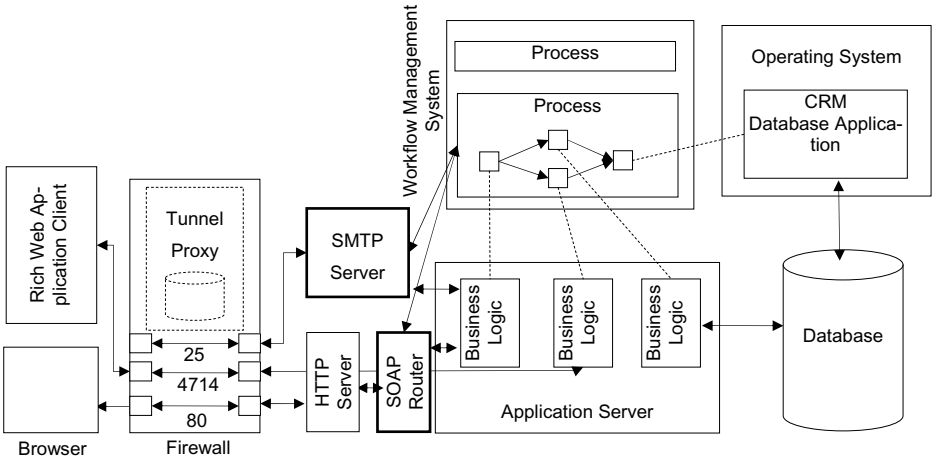


Fig. 14.6. The WPA

The result of our system redesign is depicted in a comprehensive overview in Fig. 14.7. As mentioned above, the core of the supply management system is component-oriented. In order to implement the Web service connection, the designers need to couple the components with the Web service infrastructure. Normally, such connectors are standard parts of the platform software (the component execution environment), which facilitates the Web service connection. Another issue is the design of the wrapper, which will adapt the existing supply chain management interface to the one required for the Web service connection.

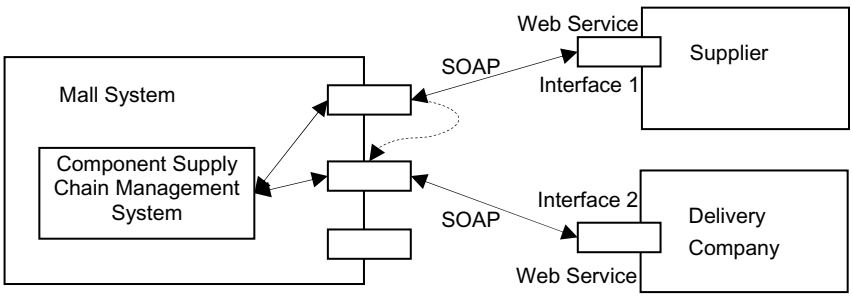


Fig. 14.7. The WPA

What we did not consider so far are the sequences of operation calls. In general, the WAA needs to be extended as well as to be adapted to the sequence of operation calls and possibly to the some of the data formats. Let us shortly discuss the two interfaces, starting with interface 1 (Fig. 14.7). The supplier requires only one operation `stockLow(ArticleID:Integer)`. When the supply management system invokes this operation, it notifies the supplier about future orders for a given article. It must plan and negotiate options on future quantities and prices (Fig. 14.8).

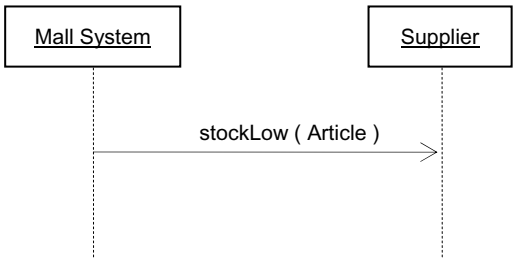


Fig. 14.8. The WPA

Interface 2 is more complex then interface 1. Once the quantity in the delivery company sinks below the critical limit, two steps have to be taken. Firstly, a query for the price of the given article needs to be placed. If the price is acceptable (approved by a person in charge), the supply management system automatically places an order for a certain quantity. The delivery company returns a possible delivery date. The supply management system approves the data. In turn, the system of the delivery company sends the delivery note and the bill for the purchase. The sequence of steps within this interaction is illustrated in Fig. 14.9.

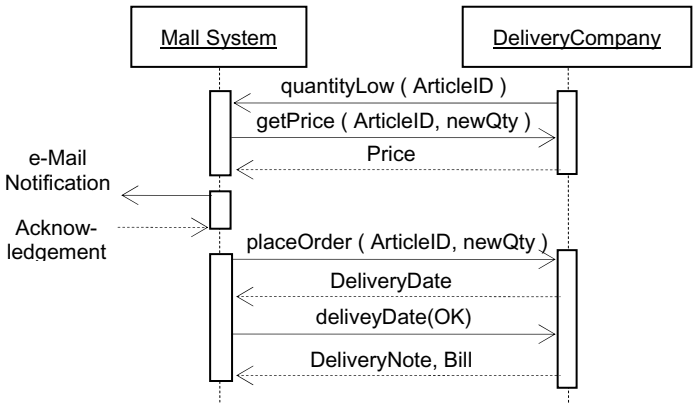


Fig. 14.9. The WPA

After having introduced the scenario and having defined how the interaction among the different parties has to be done, let us consider certain aspects of the concrete realization. The implementation strategy varies for the Web service scenario, depending on the selected development software. Although the same sequence of steps needs to be performed in principle, different development packages offer a varying degree of automation concealing the complexity of some steps behind wizards and thus increasing developer productivity. Web service implementation in Java based on open source tools, for example, require generating the WSDL interface description file and the client side stubs manually. Commercial productivity tools such as Oracle JDeveloper or BEA Web Workshop take care of this almost automatically.

The canonic sequence for developing a Web service is described in more detail in Chap. 7. Initially, the WSDL definition of the Web service interface must be created. In this example we assume that it is already available from the supplier and the delivery company. In a second step, the Web service interface must be implemented by the service provider. In this case, the service is already implemented and running. The supply chain management system plays the relative role of a "client" for the two Web services (offered by the delivery company and the supplier), therefore, it must simply build the stubs (classes `Delivery` and `Supplier`) for the two services having the WSDL files. Strictly speaking, this is the point where the Web service-specific implementation ends. On top of these stubs the programmers must implement the adaptor functionality as well, which is concentrated in the class `Wrapper` (Fig. 14.5). With this last step the implementation of the system is complete. However, some support tasks need yet to be done. These include installation of platform software and – should there be no registry – manual configuration of the system.

14.4 The Role of the Registry and Processes

In this section we focus on the synergetic role of registries and workflow in the context of the WPA. The use of these concepts may have a profound effect on the WAA. At the beginning it is worth saying that the registry technology and the workflow technology are complementary. They do not substitute other technologies; they rather catalyze certain features of the Web applications. Once again workflow and registries have a synergetic effect on Web applications – they provide the means to integrate platform modules with each other and provide dynamic discovery features for the application components. The second goal of this section is to stress once again the importance of the technologies discussed throughout the chapters of Part III of this book. At the same time it will provide some conclusions.

Let us assume that a new software component is added to the application server (Fig. 14.2). It allows managers to access the order information, cancel, or confirm orders. This feature is necessary to perform system auditing. An example of auditing is to allow huge orders to be placed by a customer before it can be verified whether the customer is solvent. It is reasonable to assume that managers doing the auditing or taking such critical order placement decisions within a company would use a rich-client application.

At this stage two decisions are very important. Firstly, how to allow automatic reconfiguration of the system as the client is introduced. Secondly, how to allow only "managers" to use the application for canceling orders.

The new auditing component will be installed and will be deployed in the application server; it will also be registered at the registry. The new component requires incoming connections to a certain TCP/IP port (e.g. 4717), therefore the platform must be automatically reconfigured. This task is done by the registry, which reconfigures the firewall to open port 4717. However, other more complex changes may be necessary; for example, special database schema to store the logging action for security reasons has to be installed.

Processes find a more technical use in the context of a registry. A process may be defined to enforce a global rule which is triggered as a result of a resolved dependency in the registry. By a global rule we mean that its scope spans multiple tiers and can configure multiple platform modules. The administrator defines a process which handles platform modules not only on the Web and application server tier, but also on the client and

the back-end tier. It is not necessary that all modules (especially the client side platform) are tightly coupled to the workflow engine. In the normal case a simple notification action such as sending a mail to all users or posting a message on a support page is sufficient.

The last important question in this scenario is how to determine which users are actually "managers" and can therefore cancel an order entry. The field of organizational modeling (Chap. 11) offers an elegant solution to this problem. The organizational model resides in the registry (Fig. 14.2) and is therefore accessible from everywhere over the search and discovery API. The firewall monitors the incoming connection requests. Once it determines that a user is currently connecting to the system it checks whether the user is granted the "manager" role in the organizational model in order to permit the connection. Now a whole process to delete the order entry is triggered in the workflow engine. A second question is what actions may be executed by "managers" in the process. As shown in Chap. 12 each step in the process has an organizational aspect which determines "who" can or must do it. Using organizational modeling the application designer may define fine-grained organization-specific groups and may use them to control the execution of certain actions.

To recapitulate, processes and registries are technologies complementing the architecture of Web applications. If not used, the Web application would not possess advantageous features such as semi-automated configuration, or user modeling. The use of registries allows platform modules to be reconfigured dynamically. A registry may deliver quite useful information in the context of change management. A registry may also preventively analyze the scope of changes potentially caused by an operation and determine what WAA components and WPA modules are affected. If critical modules are affected it may notify the administrator.

14.5 Conclusion

The programming concepts of Part II and the concepts introduced in Part III are complementary. They can and should be used in a synergetic way when building one Web application and especially when building a landscape of Web applications. Our stepwise approach helps to do this as this concluding use case has pointed out in many ways.

Many of the discussions in this book represent future-oriented and visionary ideas which cannot necessarily be implemented with contemporary technologies. For example, we showed the benefits a registry technology may provide; in practice, however, only a few products exist. They do not cover all the required functionality. While it is clear that many of these technologies will evolve in future and will eventually convert many of the futures described in this book, at present many features are not available. This should not prevent the software architects and developers from considering the ideas of this book, which show a global and integrated view. After all, many ideas such as Web content delivery or multimedia, which seemed an illusion a couple of years ago, are now part of everyday life. What is certainly true for technologies applies to software architectures too. They of course have a much longer life cycle and do not "go out of fashion" as fast, but evolve as well.

Appendix A

A.1 Introduction to UML

UML (Unified Modeling Language) [OMG03b] is a standardized language for modeling software-intensive systems. UML has evolved over the years from a proposal to the industry standard. The first version of UML appeared in 1995 as a joint effort of Grady Booch, Jim Rumbaugh, and Ivar Jacobson, preceded by the development of OMT by Rational Software in 1994 by Booch and Rumbaugh. Since then there have been many versions of UML.

This section is an executive overview of UML. The following sections are a short guide to UML use case diagrams, UML sequence diagrams, UML class diagrams, and UML package diagrams. This appendix is not meant to cover the complete set of UML modeling constructs and UML diagrams. There are numerous tutorials and books dedicated to UML. Readers looking for a full description of the UML are encouraged to refer to the UML specification [OMG03b].

A.2 UML Use Case Diagrams

Use case diagrams are part of the UML constructs facilitating requirement's engineering. A use case diagram is intended to depict the function of an organization in terms of roles and tasks. The elements of use case diagrams are actors and use cases (Fig. A.1). Actors represent persons or users interacting in some way with a system. The role of an actor is to represent user interaction such as data input and configuration. Use cases define a series of actions leading to the specification of a certain task. Indeed, use cases are associated with tasks which can later be used in system modeling and verification. A typical use case diagram contains multiple use cases, modeling the different tasks a system performs. The use case diagram elements are related to each other by three relationship types: association, dependency, and generalization.

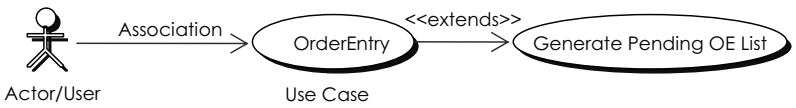


Fig. A.1. Elements of an UML use case diagram

Use case relationships may be of a certain stereotype. Stereotypes are explained in Sect. A.4. Typical stereotypes are <<includes>> or <<extends>>. They are used to denote that a use case specializes another use case; for example, by extending a dialogue or adding functionality in the form of a button. <<includes>> indicates that the included use

case will be invoked at least once, while <<extends>> indicates that the extending use case may not be invoked.

A.3 UML Sequence Diagrams

Use case diagrams provide a task-oriented view of the system, which does not include any means to model the time aspect. In other words, use case diagrams do not model the sequence of invocations between the classes resulting from the use cases. For this purpose, UML provides sequence diagrams. Alternatively, activity or collaboration diagrams (not explained here) may be used to model the high-level process flow (business process) among classes.

Use case diagrams have two dimensions. The vertical dimension represents a time axis. It represents the ordered sequence of invocations among the different classes. The horizontal dimension represents program entities (e.g. components, packages, classes, and objects) on which the invocations are performed.

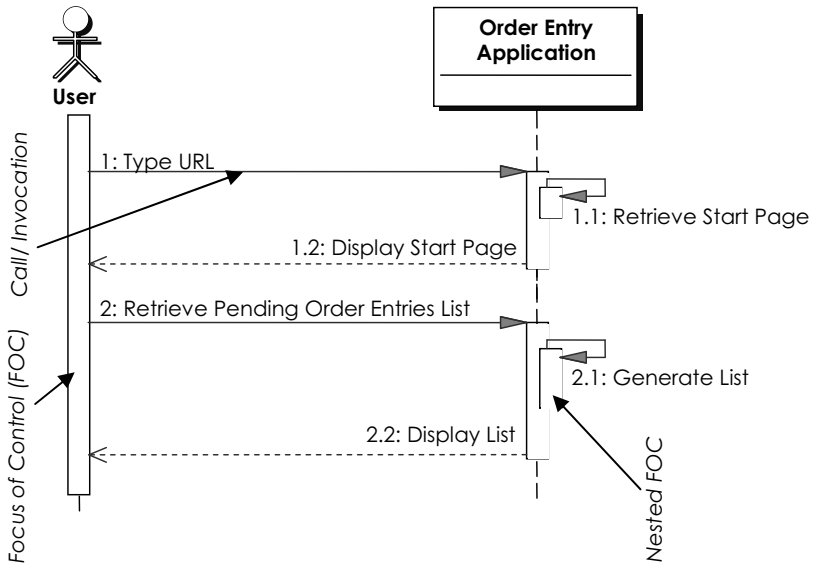


Fig. A.2. Example sequence diagram

A simple sequence diagram is shown in Fig. A.2, depicting the sequence of interactions between a user and an application. The dashed vertical lines represent the time axis for each program entity. A horizontal arrow represents an invocation. An optional dashed horizontal arrow represents return values. The narrow rectangles drawn along the time axis for each entity represent the duration of time for which the invoked action/method call is executed. In other words, for how long the program entity has the focus of control (FOC). An invocation to a program entity's own methods is a self-call, resulting in a nested FOC.

To sum up, sequence diagrams are quite useful means to represent the behavior (the dynamics) of a system. Sequence diagrams can be applied to whole use case diagrams or single use cases. The proper level of detail needs to be chosen, since sequence diagrams tend to become rather complex and unreadable if a complex scenario is depicted.

A.4 UML Class Diagrams and UML Package Diagrams

UML class diagrams are used to design models of a whole application or application modules. They represent a static view of the application. Class diagrams typically comprise a set of classes and the relationships among them. Package diagrams are closely related to class diagrams. Still, UML isolates them as a separate kind of diagram. Packages serve to partition the modeling space. A package is nothing but a container for one or more classes and their relationships. A package diagram defines the way packages relate to each other (e.g. inherit from other packages, use other package definitions (depend), nest import other packages). While a package diagram defines the rough superstructure of a model (partitioning), a class diagram defines the substructure of a package, i.e. the classes contained in each package and the way they relate to each other.

A.4.1 UML Class Diagrams

The key constituents of a class diagram (Fig. A.3) are classes and relationships. Classes correspond to real-world entity types. Relationships express the way classes relate to each other.

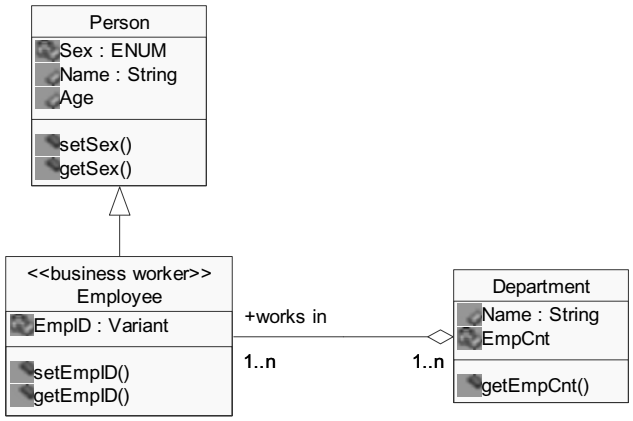


Fig. A.3. Sample UML class diagram

Each class comprises a set of attributes. Attributes represent characteristics of the class. For example, the class person has attributes Name, Age and Sex. Each class contains a set of operations. Operations represent a set of actions, which the class carries out.

Almost any UML modeling element (including classes) may have a stereotype. Stereotypes are enclosed in double angle brackets (consider the Employee class). Stereotypes are used to express a special type of modeling element. This is the right place to

destinguish between modelling elements and subclasses. Modelling elements have to do with syntax of the model, while subclasses define the semantics of the modeled artifact. In the case of Fig. A.3, the <<business worker>> stereotype defines that the Employee class is a representative of a special kind of modeling element closely related to Class. Employee being a subclass of Person implies that employees are a special subset of all persons having for example an EmpID attribute.

Classes are connected by relationships. UML defines a number of different relationship types. The most frequently used UML relationships are listed in Fig. A.4.

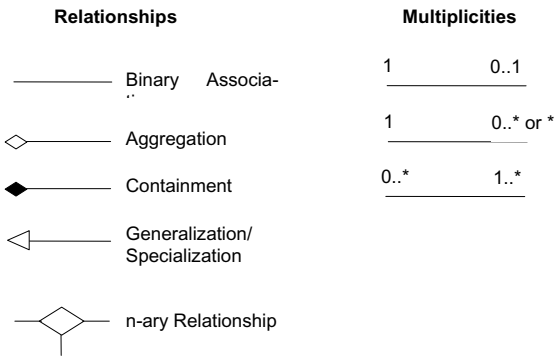


Fig. A.4: UML relationships

Every relationship has a type, a degree, and multiplicities. The degree of a relationship is determined by the number of classes it connects. UML knows binary and *n*-ary relationships. Binary associations (Fig. A.4) are the most general type of relationships in UML. To express that a class is an integral part of another class (e.g. Employee works for Department) UML provides the aggregation or containment relationship. While aggregation allows the parts to be contained in different wholes (Fig. A.3), the containment relationship allows parts to be contained only in one whole (e.g. a body has a head). A generalization/specialization relationship is used to express inheritance. Relationships (Fig. A.4) may also have multiplicities or cardinalities at both ends. Multiplicities determine the number of instances of a class, which are related to instances of the other class. If the lower bound is set to zero the relationship is said to be optional. A lower bound of one means a mandatory relationship. The upper bound can be either 1 or *n* (*), specifying that many objects, instances of that class, may be related to instances of the other class.

A.4.2 UML Package Diagrams

UML package diagrams are used to define the superstructure of UML models. UML packages can be loosely defined as simple containers for other UML modeling elements, e.g. classes, relationships, etc. In other words, a UML package can be thought of as a container for class diagrams. The goal is to simplify UML class diagrams by partitioning them into packages. Packages can form hierarchical structures by including nested packages.

A package diagram defines the relationships among packages in a UML model. A sample package diagram is shown in Fig. A.5. Dependency is the most generic type of relationship between packages. Package dependency is modeled as a dashed arrow. Generally, whenever the package at the arrow's side changes the other (the dependent) package must change as well. Packages can also be specialized. A specialization relationship exists between DataStore and OODB (object-oriented database) and RelationalDB packages. Package specialization allows classes defined in the "super-"package to be extended (specialized) in the "sub-"package.

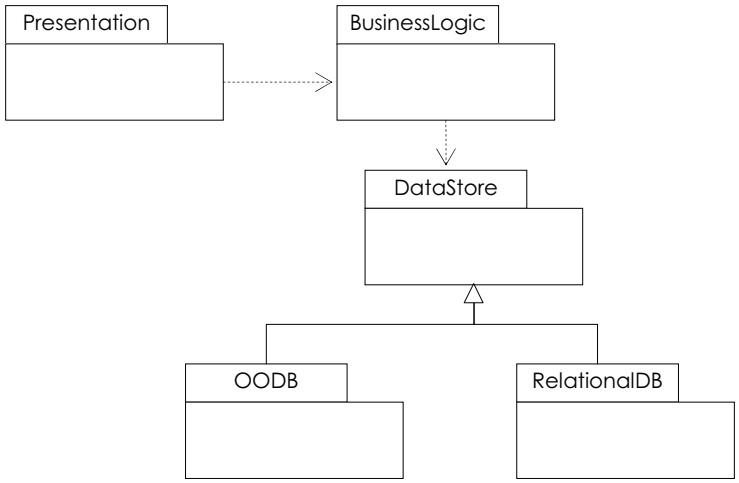


Fig. A.5. Sample UML package diagram

References

AaJa00	W. v. d. Aalst, S. Jablonski (eds.): Flexible Workflow Technology Driving the Networked Economy, Special Issue of International Journal on Computer Systems Science & Engineering (CSSE), Vol. 15, No. 5, 2000
AaLM82	F. W. Aallen, M. E. S. Loomis, M. V. Mannino: The Integrated Dictionary/Directory System, Computing Surveys, Vol. 14, No. 2, 1982
AcDA04	Microsoft Active Directory Architecture, White Paper, http://www.microsoft.com/technet/prodtechnol/windows2000serv/technologies/activedirectory/deploy/projplan/adarch.mspx, 2004
ACKM03	G. Alonso, F. Casati, H. Kuno, V. Machiraju: Web Services, Springer, 2003
Acti04	Microsoft ActiveX Controls, http://www.microsoft.com/com/tech/ActiveX.asp, 2004
Adap04	Adaptive Repository, http://www.adaptive.com/products/repository.html, 2004
AlCM03	D. Alur, J. Crupi, D. Malks: Core J2EE Patterns, Best Practices and Design Strategies, Prentice Hall, 2003
Amaz04	Amazon.com Web Services, http://www.amazon.com/webservices, 2004
Appl04	Sun Applet Resources, http://java.sun.com/applets/, 2004
ASP04	Active Server Pages, http://msdn.microsoft.com/asp, 2004
BaGP00	L. Baresi, F. Garzotto, P. Paolini: From Web Sites to Web Applications: New Issues for Conceptual Modeling, Proceedings ER'2000 Workshop on Conceptual Modeling and the Web, 2000
BaZL03	T. Baier, C. Zirpins, W. Lamersdorf: Digital Identity: How To Be Someone On The Net, Proceedings of e-Society 2003 IADIS International Conference, 2003
BeDa94	P. A. Bernstein, U. Dayal: An Overview of Repository Technology. Conference on Very Large Databsas (VLDB 1994), 1994
Bern96	P. A. Bernstein: Middleware: A Model for Distributed System Services, Communications of the ACM Vol. 39, No. 2, 1996
Bern97	P. Bernstein: Repositories and Object-Oriented Databases. Proceedings of BTW '97, Springer, 1997
BhRa00	S. Bhattacharjee, R. Ramesh: Enterprise Computing Environments and Cost Assessment, Communications of the ACM, Vol. 43, No. 10, 2000
BizT04	Microsoft BizTalk Server, http://www.microsoft.com/biztalk/default.asp, 2004
BMR+96	F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996
BPEL03	Business Process Execution Language (BPEL) for Web Services Version 1.1, http://www-106.ibm.com/developerworks/library/ws-bpel/, 2003
BPML04	Business Process Modeling Language (BPML), http://www.bpmi.org/bpml.esp, 2004
Burb92	S. Burbeck: Applications Programming in SmallTalk-80: How to use Model-View-Controller (MVC), http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, 1992
Buss98	C. Bussler: Organisation Management in Worfklow Management Systems (in German), Deutscher Universitätsverlag, 1998
CaWH00	M. Campione, K. Walrath, A. Huml: The Java Tutorial: A Short Course on the Basics, Addison-Wesley, 2000
CCM02	Corba Component Model Specification, Version 3.0, http://www.omg.org/technology/documents/formal/components.htm, 2002
CDIF04	CDIF Standards, http://www.eigroup.org/cdif/how-to-obtain-standards.html, 2004
CFac04	Catalog of OMG CORBAfacilities Specifications, http://www.omg.org/technology/documents/corbafacilities_spec_catalog.htm, 2004
CFB+03	S. Ceri, P. Fraternali, A. Bangio, M. Brambilla, S. Comai, M. Matera: Designing data-intensive Web applications, Morgan Kaufmann, 2003

CGI04 The CGI Specification - Version 1.1, http://hoohoo.ncsa.uiuc.edu/cgi/interface.html, 2004

CIDL04 Catalog of OMG IDL/Language Mappings Specifications, http://www.omg.org/technology/documents/idl2x_spec_catalog.htm, 2004

Coco04 The Apache Cocoon Project, http://xml.apache.org/cocoon/, 2004

COM+04 Com+ Components Services, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/complus_anchor.asp, 2004

Cona02 J. Conallen: Building Web Applications with UML, Addison-Wesley, 2002

Cona99 J. Conallen: Modeling Web Application Architectures with UML, Rational Software White Paper, 1999

CORB04 CORBA, http://www.corba.org/, 2004

CSer04 Catalog of OMG CORBAservices Specifications, http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm, 2004

CuER02 F. Curbera, D. Ehnebuske, D. Rogers: Using WSDL in a UDDI registry, version 1.07, UDDI best practice, http://www.uddi.org/pubs/wsdlbestpractices-V1.07.Open-20020521.pdf, 2002

Daft03 R. L. Daft: Organization Theory and Design, South-Western College Publ., 2003

DCND90 Data Communications Network Directory, ISO 9594, Recommendations X.500-X.521, 1990

DCOM04 DCOM Technical Overview, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp, 2004

DMTF04 DMTF CIM Schema, http://www.dmtf.org/standards/cim/cim_schema_v28, 2004

Dubl04 Dublin Core Metadata Initiative, http://dublincore.org/, 2004

DUNS04 D&B D.U.N.S Number, http://www.dnb.com/US/duns_update/index.html, 2004

Ecke02 B. Eckel: Thinking in Java, Prentice Hall, 2002

EdEd99 G. Eddon, H. Eddon: Inside COM+ Base Services, Microsoft Press, 1999

ElNa02 R. Elmasri, S. B. Navathe: Fundamentals of Database Systems, Addison-Wesley, 2002

Emme00 W. Emmerich: Engineering Distributed Objects, John Wiley & Sons, 2000

FCGI04 FastCGI, http://www.fastcgi.com, 2004

FKNT02 I. Foster, C. Kesselman, J. Nick, S. Tuecke: Grid Services for Distributed System Integration, IEEE Computer Magazine, Vol. 35, No. 6, 2002

FlFe01 D. Flanagan, P. Ferguson: JavaScript – The Definitive Guide, O'Reilly & Associates, 2001

Fran03 D. S. Frankel: Model Driven Architecture: Applying MDA to Enterprise Computing, John Wiley & Sons, 2003

Geig95 K. Geiger: Inside ODBC, Microsoft Press, 1995

GHJV97 E. Gamma, R. Helm, R. Johnson, J. Vassilides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1997

Goog04 Google.com Web Service API, http://www.google.com/apis, 2004

GrRe93 J. Gray, A. Reuter: Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993

HaHP00 D. Hatley, P. Hruschka, I. Pirbhai: Process for System Architecture and Requirements Engineering, Dorset House Publishing, 2000

HoNS00 C. Hofmeister, R. Nord, D. Soni: Applied Software Architecture, Addison-Wesley Longman, 2000

IBM04 IBM Distributed Relational Database Architecture Reference (SC26.4651)

IBMU04 IBM UDDI Node, uddi.ibm.com, 2004

IIS04 Microsoft Internet Information Services, http://www.microsoft.com/windowsserver2003/iis/default.mspx, 2004

Iona04 IONA iPortal Application Server - Technical Overview, http://www.netfish.com/support/whitepapers/ipas_techoverviewWP.pdf, 2004

IRFC94 IETF RFC 1630, Universal Resource Identifiers in WWW, http://www.w3.org/Addressing/rfc1630.txt, 1994

ISO87 ISO/IEC 8824:1987 and ISO/IEC 8825:1987

ISO90 ISO/IEC 10027:1990, Information technology – Information Resource Dictionary System (IRDS) framework, 1990

J2EE04 Java 2 Platform Enterprise Edition, http://java.sun.com/j2ee, 2004

JaBu96 S. Jablonski, C. Bussler: Workflow Management: Modeling Concepts, Architecture and Implementation, International Thomson Publishing, 1996

JaDo04 Sun Core JavaDoc Tool, http://java.sun.com/j2se/javadoc/, 2004

JaHS01 S. Jablonski, S. Horn, M. Schlundt: Process Oriented Knowledge Management. Eleventh International Workshop on Research Issues in Data Engineering (RIDE): Document Management for Data Intensive Business and Scientific Applications, Heidelberg, 2001

Jaka04 The Jakarta Project – Taglibs, http://jakarta.apache.org/taglibs/, 2004

Java04 The Java Programming Language, http://java.sun.com/, 2004

JAXM03 Sun Microsystems JAXM 1.1.2, http://java.sun.com/xml/jaxm/, 2003

JAXR03 Sun Microsystems JAX-RPC 1.1, http://java.sun.com/xml/jaxrpc/, 2003

JBH+98 H. Johner, L. Brown, F.-S Hinner, W. Reis, J. Westman: Understanding LDAP. IBM International Technical Support Organization, http://www.redbooks.ibm.com, 1998

JBOS04 JBOSS Application Server, http://www.jboss.org/, 2004

JCAr04 J2EE Connector Architecture, http://java.sun.com/j2ee/connector/, 2004

JDBC04 JDBC Data Access API, http://java.sun.com/products/jdbc/, 2004

JONA04 JONAS Application Server, http://jonas.objectweb.org/, 2004

JSP04 Java Server Pages, http://java.sun.com/products/jsp/, 2004

KaBu03 D. Karastoyanova, A. Buchmann: Components, Middleware and Web Services, Proceedings of IADIS International Conference WWW/Internet, 2003

Katc03 T. Katchaounov: An overview of Web services and related technologies, Technical report, University of Uppsala, 2003

KeCh03 J. O. Kephart, D. M.Chess: The Vision of Autonomic Computing. IEEE Computer Magazine, January, 2003

KeRi98 B. Kernighan, D. Ritchie: The C Programming Language, Prentice Hall, 1998

KiRo02 R. Kimball, M. Ross: The Data Warehouse Toolkit, John Wiley & Sons, 2002

Kirt98 M. Kirtland: Designing Component-Based Applications, Microsoft Press, 1998

KlWB03 A. Kleppe, J. Warmer, W. Bast: MDA Explained: The Model Driven Architecture - Practice and Promise, Addison-Wesley, 2003

KPRR03 G. Kappel, B. Pröll, S. Reich, W. Retschitzegger: Web Engineering (in German), dpunkt Verlag, 2003

LiAr04 Liberty ID-FF Architecture Overview, http://www.projectliberty.org/specs, 2004

LibA04 Liberty Alliance Project, http://www.projectliberty.org/, 2004

Loma97 P. Lomax: Learning VBScript, O'Reilly & Associates, 1997

Loud03 K. C. Loudon: Programming Languages – Principles and Practice, PWS Publishing, 2003

Mane01 A. T. Manes: Enabling Open, Interoperable, and Smart Web Services. The Need for Shared Context, Sun Microsystems, Inc., 2001

MDCo99 Meta Data Coalition, Open Information Model Version 1.1, http://www.mdcinfo.com/OIM/MDCOIM11.pdf, 1999.

MeBo99 K. Meyer-Wegener, M. Böhm: Conceptual Workflow Schemas, Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems, Edinburgh, Scotland, September 2-4, 1999

Micr04 Microsoft UDDI Node, uddi.microsoft.com, 2004

MISA04 Microsoft Internet Server API Documentation, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/HTML/_core_internet_server_api_.28.isapi.29_.extensions.asp, 2004

Mogu02 J. C. Mogul: Clarifying the fundamentals of HTTP, Proceedings of WWW2002, 200

Mons01 R. Monson-Haefel: Enterprise Java Beans, O'Reilly & Associates, 2001

NAIC03 NAICS Association, http://www.naics.com, 2003

Neeb01 J. Neeb: Administration of Workflow Management Solutions (in German), Shaker Verlag, 2001

NeGr91 S. Newmann, J. Gray: SQL Access and IBM DRDA: A Comparison in a Multi-Vendor Setting. Digital Equipment Corporation, http://research.microsoft.com/~gray/papers/SqlAccessVsDrda.doc, 1991

Net04	Getting Started in .NET, http://www.microsoft.com/net/, 2004
Newc02	E. Newcomer: Understanding Web Services, Addison-Wesley, 2002
NMMZ00	J. Noack, H. Memaneche, H. Memaneche, A. Zendler: Architectural Patterns for Web Applications, 2000
NSAP04	Netscape NSAPI Basics, http://developer.netscape.com/docs/manuals/enterprise/nsapi/svrop.htm, 2004
OCCM04	Open CCM – The Open CORBA Component Model Plattform, http://openccm.objectweb.org/, 2004
OMG02a	Object Management Group: Meta Object Facility Specification Version 1.4, http://www.omg.org/technology/documents/formal/mof.htm, 2002
OMG02b	Object Management Group: XMI - XML Metadata Interchange Specification Version 2.0, http://www.omg.org/technology/documents/formal/xmi.htm, 2002
OMG03a	OMG. Common Warehouse Metamodel (CWM) Specification Version 1.1, OMG Document formal/03.03.02, 2003
OMG03b	OMG Unified Modeling Language (UML), Version 1.5 OMG Document formal/03.03.01, http://www.omg.org/cgi-bin/doc?formal/03.03.01, 2003
OMG04	The Object Management Group, http://www.omg.org/, 2004
OOrb04	OpenOrb Ver. 1.4.0, http://openorb.sourceforge.net/, 2004
OraA04	The Oracle Application Server, http://www.oracle.com/appserver/, 2004
Orac04	Oracle Designer, http://otn.oracle.com/documentation/designer.html, 2004
Orbi04	ORBIX Ver. 6.1, http://www.iona.com/products/orbix.htm, 2004
OrHa98	R. Orfali, D. Harkey: Client/Server Programming with Java and CORBA, John Wiley & Sons, 1998
OrHE96	R. Orfali, D. Harkey, J. Edwards: The Essential Distributed Objects Survival Guide, John Wiley & Sons, 1996
Ortn99	E. Ortner: Repository Systems Part 1: Multi Layering and Development Infrastructure and Repository Systems Part 2: Conception and Maintenance of a Development Repository (in German), Informatik-Spektrum, Vol. 22, Issue 4 and Issue 5, 1999
PasR04	Microsoft .NET Passport Review Guide, http://www.microsoft.com/net/services/passport/review_guide.asp, 2004
Pass04	Microsoft .NET Passport, http://www.passport.com, 2004
Patt00	R. Patton: Software Testing, SAMS, November, 2000
Pers04	Personalization Consortium, http://www.personalization.org/, 2004
PetS04	Java Pet Store, http://java.sun.com/developer/releases/petstore/, 2004
PHP04	PHP, http://www.php.net/, 2004
PiPe91	T. Pittman, J. Peters: The Art of Compiler Design – Theory and Practice, Prentice Hall, 1991
Posi03	Standard of Information Technology – Portable Operating System Interface (POSIX), http://posixcertified.ieee.org/, 2003
RDFC04	Resource Description Framework (RDF): Concepts and Abstract Syntax ,W3C Recommendation, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/, 2004
RMI04	Java Remote Method Invocation, http://java.sun.com/products/jdk/rmi/, 2004
RoRi02	G. Rothfuss, C. Ried: Content Management with XML (in German), Springer Xpert.press, 2002
RPC95	Remote Procedure Call Protocol Specification Version 2, http://www.ietf.org/rfc/rfc1831.txt, 1995
SDK04	Sun Microsystems. J2EE 1.4 SDK, Enterprise Edition 1.4, http://java.sun.com/j2ee/1.4/download-sdk.html, 2004
SemW04	The Semantic Web Community Portal, http://www.semanticweb.org, 2004
Serv04	The Java Servlet Technology, http://java.sun.com/products/servlet/, 2004
SGML04	The SGML History, http://www.sgmlsource.com/history/, 2004
ShSN01	R. Sharma, B. Stearns, T. Ng: J2EE Connector Architecture and Enterprise Application Integration, Addison-Wesley, 2001

SiSJ02	I. Singh, B. Stearns, M. Johnson: Designing Enterprise Applications with the J2EE Platform, Addison-Wesley, 2002
SiWS02	Security in a Web Services World: A Proposed Architecture and Roadmap, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwssecur/html/securitywhitepaper.asp, 2002
SOAP00	Simple Object Access Protocol (SOAP) Specification Version 1.1, http://www.w3.org/TR/SOAP/, 2000
Somm00	I. Sommerville: Software Engineering, Addison-Wesley, 2000
SPKH01	ICE Implementation Cookbook, http://www.icestandard.org, 2001
SQLJ04	SQLJ, http://www.sqlj.org/, 2004
SSI04	Introduction to Server Side Includes, http://httpd.apache.org/docs/howto/ssi.html, 2004
Stev92	W. R. Stevens: Advanced Programming in the Unix Environment, Addison-Wesley, 1992
Stoe00	H. Störle: Models of Software Architecture – Design and Analysis with UML and Petri-Nets, Books on Demand, 2000
Szyp97	C. Szyperski : Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1997
TMQ+03	D. Trowbridge, D. Mancini, D. Quick, G. Hohpe, J. Newkirk, D. Lavigne: Enterprise Solution Patterns Using Microsoft .NET, Microsoft Press, 2003
UDDI03	UDDI Version 3.0.1, UDDI Technical Committee Specification, http://uddi.org/pubs/uddi-v3.0.1-20031014.htm, 2003
Unis04	Unisys Universal Repository Manager, http://www.unisys.com/marketplace/urep, 2004
UNSP04	United Nations Standard Products and Services Code (UNSPSC), http://www.unspsc.com, 2004
Visi04	VisiBroker Ver. 6.0, http://www.borland.com/visibroker, 2004
WaJo93	L. Wakeman, J. Jowett: PCTE - The Standard for Open Repositories, Prentice Hall, 1993
WebO04	Web-Ontology (WebOnt) Working Group, http://www.w3.org/2001/sw/WebOnt, 2004
WebS04	Web Sphere – Enterprise Application Server, http://www.ibm.com/websphere, 2004
WeSA03	Web Services Architecture, W3C Working Draft, http://www.w3.org/TR/ws-arch, 2004
Wild99	Erik Wilde: Wilde's WWW: Technical Foundations of the World Wide Web, Springer, 1999
WiMI04	Microsoft Windows Management Instrumentation: Background and Overview, White Paper, http://www.microsoft.com/windows2000/docs/WMIOverview.doc, 2004
WSCI02	Web Service Choreography Interface (WSCI) 1.0, http://www.w3.org/TR/wsci/, 2002
WSCO03	Web Services Coordination (WS-Coordination), http://www-106.ibm.com/developerworks/library/ws-coor/, 2003
WSDL03	Web Service Description Language Specification Version 1.1, http://www.w3.org/TR/wsdl, 2003
WSFL01	Web Service Flow Language (WSFL) Version 1.0, http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, 2001
WSPo03	Web Service Polciy Framework (WSPolicy), http://www-106.ibm.com/developerworks/library/ws-polfram/, 2003
WSSe02	Web Service Security (WS-Security) Specification 1.0, http://www-106.ibm.com/developerworks/webservices/library/ws-secure/, 2002
WSTL02	Web Services Trust Language (WS-Trust) Specification Draft, http://www-106.ibm.com/developerworks/library/ws-trust/, 2002
WSTr02	Web Service Transaction (WS-Transaction), http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/, 2002
XAML04	Transaction Marcup Language (XAML), http://www.xaml.org, 2004
XHTM04	The Extensible Hypertext Markup Language, http://www.w3.org/TR/xhtml1/, 2004
XKMS01	XML Key Management Specification (XKMS), http://www.w3.org/TR/xkms/, 2001
XLANG01	XLANG, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001
XMLE02	XML Encryption Syntax and Processing, W3C Candidate Recommendation, http://www.w3.org/TR/2002/CR-xmlenc-core-20020304/, 2002
XMLI01	XML Information Set, W3C Recommendation, http://www.w3.org/TR/xml-infoset, October 2001

XMLS04	XML Signature WG, http://www.w3.org/Signature/, 2004
XSLT04	XML Transformations (XSLT) Version 1.0, W3C Recommendation, http://www.w3.org/TR/xslt, 2004
XSP04	User Documentation XSP, http://xml.apache.org/cocoon/userdocs/xsp/, 2004
YeHK95	W. Yeong, T. Howes, S. Kille: RFC 1777 - Lightweight Directory Access Protocol, 1995
ZsTZ01	O. Zschau, D. Traub, R. Zahradka: Web Content Management – To plan and to maintain Web Sites Professionally (in German), Galileo Press, 2001

Index

I

J

L

M

N

O

P

R

S

T

U

V

W

X