Engineering Device-Independent Web Services

Ph.D. Thesis

Engin Kirda

Engineering Device-Independent Web Services

An XML/XSL-based approach to creating flexible and extensible multi-device services

Ph.D. Thesis

at

Technical University of Vienna

submitted by

Dipl.-Ing. Engin Kirda

Distributed Systems Group, Information Systems Institute,
Technical University of Vienna
Argentinierstr. 8/184-1
A-1040 Vienna, Austria

19th August 2002

© Copyright 2002 by Engin Kirda

Advisor: o. Univ.-Prof. Dr. Mehdi Jazayeri
Second Advisor: a.o. Univ.-Prof. Dr. Gabriele Kotsis

Abstract

The popularity of computing devices such as Personal Digital Assistants (PDAs) and mobile phones have been increasingly and these devices have been getting more powerful every day. Although the latest PDAs are even able to display frames, it is still important to adapt the content for these devices in order to provide a satisfactory surfing experience for users. Web services in the near future will not only have to support mobile access, but will also have to deal with other forms of Web access such as voice interfaces. Hence, Web services will often need to be *device-independent* and will have to support different XML Web formats.

Although much work has been done on providing mobile access to Web content, the focus has mainly been the adaptation of HTML content to make it viewable on mobile devices that might have memory and screen-size limitations. Only a few attempts have been made to date to integrate device-independence into the design, implementation and maintenance phases of Web services.

This dissertation provides solutions to the problem of designing and implementing interactive, maintainable, device-independent Web services. It introduces a novel XML/XSL-based design and implementation technique and a development tool suite to support the Web developer. The constructed services can be accessed by a wide range of Web devices such as mobile phones, PDAs with micro HTML browsers, speech-based Web interfaces and traditional full-fledged HTML browsers.

My general thesis is that Web services can effectively be made device-independent if device-independence support is integrated into the Web service design, implementation and maintenance phases. I present an extended model of the traditional Web service life cycle that takes device-independence support into account and describe the Device-Independent Web Engineering (DIWE) framework for engineering device-independent Web services. I introduce the novel concepts of page splitting, process partitioning and XSL stylesheet pre-processing.

Kurzfassung

Elektronische Geräte wie Personal Digital Assistants (PDAs) und Mobiltelefone sind in den letzten Jahren sehr populär und leistungsfähig geworden. Die neuesten PDAs können sogar Frames in Webseiten darstellen. Trotzdem ist es noch immer wichtig, den Webinhalt für diese Geräte so anzupassen, dass die Benutzer zufrieden sind und eine positive Erfahrung mit der Website haben. Bald werden viele Websites nicht nur mobilen Zugang, sondern andere Formen des Webzugangs wie zum Beispiel Sprachschnittstellen unterstützen. Die Websites der Zukunft müssen *geräteunabhängig (device-independent)* sein.

Der Fokus der Forschung bis jetzt ist die Anpassung und Abbildung des HTML Inhalts von Websites gewesen damit sie auf mobilen Geräten mit wenig Hauptspeicher und kleinen Displays dargestellt werden können. Nur wenige Forschungsgruppen haben versucht, Geräteunabhängigkeit in den Design-, Implementierungs-, und Wartungsphasen der Website zu integrieren.

Diese Dissertation presentiert Lösungen zum Problem des Entwerfens und der Implementierung von interaktiven, geräteunabhängigen Websites. Sie beschreibt eine neue XML/XSL-basierte Methodologie und ein Webentwicklungswerkzeug.

Contents

1 Introduction **1**

1.1 Overview . . . 1
1.2 Contribution of this Dissertation . . . 2
1.3 Structure of this Dissertation . . . 3

2 Web Engineering basics **5**

2.1 Terminology . . . 5
2.2 Web engineering: An emerging field . . . 6
2.3 Web service characteristics and requirements . . . 8
2.3.1 Information transfer characteristics . . . 8
2.3.2 Stakeholders . . . 8
2.3.3 Basic Web service requirements . . . 8
2.4 Web Service Life Cycle . . . 9
2.4.1 Requirements Analysis . . . 10
2.4.2 Design . . . 11
2.4.3 Implementation . . . 11
2.4.4 Maintenance . . . 11
2.5 Flexibility . . . 12
2.5.1 XML . . . 13
2.5.2 XSL . . . 14
2.6 The device-independent Web engineering problem . . . 15
2.6.1 Historical overview . . . 15
2.6.2 Problem: Constructing maintainable, interactive device- independent Web services . . . 18
2.7 Summary . . . 21

3 Related Work **22**
3.1 Brief overview of research on device-independent Web access 22
3.2 Traditional Web engineering approaches 23
3.2.1 The Dexter hypertext reference model 23
3.2.2 The Relationship Management Methodology (RMM) 24
3.2.3 Object-Oriented Hypermedia Design Methodology (OOHDM) . . . 24
3.2.4 W3DT and eW3DT 25
3.2.5 Webcomposition and W3Objects 26
3.2.6 Strudel 26
3.2.7 Araneus 27
3.3 Mobile Web access techniques 27
3.3.1 Quality aware transcoding 28
3.3.2 Digestor 28
3.3.3 Annotation-based Web content transcoding 29
3.3.4 The Business Card Search Service (BCSS) 30
3.3.5 Web access with PDAs: PowerBrowser 30
3.3.6 Web content and form summarization 30
3.4 A taxonomy for device-independent Web engineering 31
3.5 Device-independent Web engineering approaches 34
3.5.1 OO-H Method 34
3.5.2 WebML 36
3.5.3 JML 37
3.5.4 SISL 37
3.5.5 UIML 37
3.5.6 iStudio 38
3.5.7 Cocoon 39
3.5.8 Microsoft ASP.NET and the Mobile Developer Toolkit 40
3.5.9 Total e-mobile 41
3.6 Summary 42

4 DIWE: A conceptual framework for device-independent Web engineering **43**
4.1 Rethinking the Web Service Life Cycle 43
4.2 Basis of solution: Separation of Layout, Content and Logic (LCL) 45
4.3 Main requirements for a device-independent Web engineering framework . 46
4.4 Overview of the DIWE framework 47
4.4.1 Web service design, implementation, deployment and maintenance . 47
4.4.2 Processors 49
4.5 Flexible Web service construction in three steps 50
4.6 Device-independent Web service construction in three steps 52

4.7 The MyXML language . . . 54
4.7.1 Overview . . . 54
4.7.2 MyXML Namespace . . . 55
4.7.3 A simple MyXML example: Searching for musicals . . . 57
4.7.4 Another MyXML example: Shopping Cart . . . 59
4.7.5 Post XSL stylesheet application . . . 63
4.8 XSL stylesheet pre-processing for stylesheet reuse . . . 64
4.9 Page splitting . . . 66
4.9.1 Page splitting descriptors and parameters . . . 68
4.9.2 A simple page splitting example . . . 69
4.10 Process partitioning . . . 71
4.10.1 Process partitioning parameters . . . 72
4.10.2 A simple process partitioning example . . . 72
4.11 Device-independent application logic interfacing . . . 76
4.11.0.1 Calling the logic in three steps . . . 76
4.11.0.2 A simple example . . . 78
4.12 Summary . . . 78

5 The MyXML tool suite: A prototype implementation 79
5.1 The MyXML tool suite . . . 79
5.2 The MyXML compiler . . . 82
5.2.1 Usage . . . 82
5.2.2 Implementation . . . 83
5.3 Configurable device-independence components . . . 85
5.3.1 The Dispatcher component . . . 86
5.3.1.1 Configuration grammar . . . 86
5.3.1.2 A configuration example . . . 87
5.3.1.3 Implementation . . . 88
5.3.2 The Collector component . . . 89
5.3.2.1 Configuration grammer . . . 89
5.3.2.2 A configuration example . . . 90
5.3.2.3 Implementation . . . 91
5.3.3 The Output component . . . 91
5.3.3.1 Configuration grammer . . . 91
5.3.3.2 A configuration example . . . 92
5.3.3.3 Implementation . . . 93
5.4 MyXMLDesigner . . . 93
5.4.1 Overview of the IDE . . . 94
5.4.2 Support for design . . . 95
5.4.3 Support for implementation . . . 96
5.4.4 Support for configuration and deployment . . . 96
5.4.5 Support for Web page creation and maintenance . . . 97
5.4.6 Architecture and implementation . . . 98
5.5 Summary . . . 99

6 Case Study: VIF e-Commerce Web service **100**
6.1 The Vienna International Festival (VIF) Web site 100
6.1.1 Service overview 101
6.1.2 Main VIF components 101
6.2 VIF e-commerce Web service 102
6.2.1 The programme 102
6.2.2 Detailed event information 102
6.2.3 Ticket availability, date and price information 103
6.2.4 The shopping cart 103
6.2.5 Completing the order (checking out) 103
6.3 Implementation with the MyXML tool suite 104
6.3.1 Design 104
6.3.1.1 Device identification 104
6.3.1.2 Data organization planning 104
6.3.1.3 Content definition 105
6.3.1.4 XSL stylesheet definition 106
6.3.2 Implementation 106
6.3.2.1 Construction of the pages 106
6.3.2.2 Integration of PDA device family 107
6.3.3 Deployment 108
6.3.4 Maintenance 108
6.4 Usage scenarios 109
6.4.1 Ordering a ticket using a traditional browser 109
6.4.2 Ordering a ticket using an iPAQ PDA 109
6.4.3 Ordering a ticket using a WAP phone 110
6.5 Summary 110

7 Evaluation and Future Work **122**
7.1 Empirical proof of concepts 122
7.1.1 Setting up an experiment 122
7.1.2 Example: Measuring readability 123
7.2 Analysis and discussion 123
7.2.1 Stylesheet complexity and numbers 123
7.2.1.1 Discussion 124
7.2.1.2 Conclusion 124
7.2.2 Complexity 125
7.2.2.1 Discussion 125
7.2.2.2 Conclusion 125
7.2.3 Layout adaptation 125

7.2.3.1 Discussion 126
7.2.3.2 Conclusion 126
7.2.4 Graphical and navigational design 126
7.2.4.1 Discussion 127
7.2.4.2 Conclusion 127
7.2.5 Layout/Content/Logic (LCL) separation 127
7.2.5.1 Discussion 127
7.2.5.2 Conclusion 128
7.2.6 Comparison of the DIWE framework to other approaches 128
7.3 Laying out future work 131
7.3.1 Higher level abstractions 131
7.3.2 UML for visual modeling 131
7.3.3 Re-engineering for device-independence 132
7.4 Summary 132

8 Conclusion **133**

A Sample case study code listings **135**

Bibliography **147**

List of Figures

2.1 Life Cycle of a Web Service [Sch98b, TL97] 10
2.2 The difficulty of supporting small displays: The DSG homepage as seen on an iPAQ PDA 16
2.3 Screenshots of the 1995 and 2001 VIF home pages 17
2.4 Part of the Perl script implementing the HTML grading service 19
2.5 Part of the Perl script implementing the WAP grading service 19
2.6 Part of the VIF 2000 servlet code implementing a shopping cart 20

3.1 Adaptation of HTML for mobile computing devices (Hori et. al [HKO+00]) 29
3.2 OO-H Design Process (Gomez et al. [GCP01]) 35
3.3 WebML graphic notation for data units, and a possible rendition in HTML (Ceri et al. [CFB00]) 36
3.4 A sample iStudio fragment that defines an XHTML form (Skarra et al. [SHKE01]) 38
3.5 Part of a logic sheet in Cocoon 40

4.1 Life Cycle of a device-independent Web Service 44
4.2 Web service design, implementation, deployment and maintenance 48
4.3 Differences in description granularity 49
4.4 Interactions between the user's device, the Web server and the generated static content 50
4.5 Interactions between the user's device, the Web server, the application logic and the generated functionality that produces the dynamic content 51
4.6 Sequence diagram showing the interactions between the device-independence components for static content 52
4.7 Sequence diagram showing the interactions between the device-independence components for dynamic content 53
4.8 Example MyXML file to search in a database 58
4.9 XSL stylesheet for formatting the output 58
4.10 Part of the generated Java Source Code 59
4.11 MyXML content definition for a shopping cart 60
4.12 XSL layout definition for the shopping cart 61
4.13 Part of the generated shopping cart Java code encapsulating the HTML code 62

4.14 Invoking the generated code 63
4.15 XSL Stylesheet reuse with pre-processing 65
4.16 XSL Stylesheet for PDA access after pre-processing 66
4.17 XSL Stylesheet for full HTML access after pre-processing 66
4.18 Page splitting using groups and subgroups 67
4.19 MyXML document for the events page 69
4.20 XSL layout definition for HTML event page 69
4.21 XSL layout definition for WML event page 70
4.22 An online WML-based order with process partitioning compared to a traditional HTML-based order 71
4.23 XSL layout definition for HTML Web form 73
4.24 Screenshot of simple HTML Web form 73
4.25 XSL layout definition for the partitioned HTML Web form 74
4.26 Screenshot of the partitioned HTML Web form – First group 75
4.27 Screenshot of the partitioned HTML Web form – Second group 75
4.28 Invoking the *Checkout* layout/content class from the application logic . . . 77
4.29 The MyXML-generated *Checkout* layout/content class 77

5.1 Relations between the tools in the MyXML tool suite 79
5.2 The MyXML tool suite in Web service construction and operation based on the DIWE framework 81
5.3 Flowchart showing the main steps taken by the MyXML compiler 83
5.4 UML class diagram describing the architecture of the MyXML compiler . . 84
5.5 The Dispatcher component configuration DTD 87
5.6 A Dispatcher configuration for a service 88
5.7 UML class diagram showing the architecture of the Dispatcher component . 89
5.8 The Collector component configuration DTD 90
5.9 A typical XML Collector component configuration 90
5.10 UML class diagram describing the architecture of the Collector Component 91
5.11 The Output component configuration DTD 92
5.12 A typical XML Output component configuration 92
5.13 UML class diagram of the Output component 93
5.14 The MyXMLDesigner visual Integrated Development Environment (IDE) . 94
5.15 Configuring general device properties 97
5.16 Simplified UML class diagram describing the architecture of MyXMLDesigner 98

6.1 Main VIF Components in 2000 101
6.2 Screenshot of the project pane for the VIF project 107
6.3 Adding the PDA layout to the Web service 107
6.4 Default HTML programme page 111

6.5 Default HTML detailed event information 112
6.6 Default HTML ticket reservation page 113
6.7 Default HTML shopping cart 114
6.8 Completing the order (checking out) in the default HTML layout 115
6.9 Default HTML order confirmation 116
6.10 Programme, detailed event information and ticket reservation for the PDA device family (screenshots from an iPAQ running Windows CE) 117
6.11 Shopping cart and order form for the PDA device family (screenshots from an iPAQ running Windows CE) 118
6.12 Programme, detailed event information and ticket reservation for the WAP device family (as seen on a WAP emulator) 119
6.13 Part of ticket reservation and shopping cart for the WAP device family (as seen on a WAP emulator) 120
6.14 Order form for the WAP device family (as seen on a WAP emulator) 121

7.1 The full HTML interface of the VIF programme as seen on an iPAQ PDA . 126

List of Tables

3.1 Comparison of device-independent Web engineering approaches 32
3.2 Comparison of device-independent Web engineering approaches 33

4.1 Page splitting-related CGI parameters that the page splitting processor interprets 68
4.2 Descriptors that the page splitting processor substitutes at run-time 68
4.3 Table showing process partitioning-related CGI parameters the Collector component understands 72

5.1 The Web service life cycle phases each tool in the MyXML tool suite supports 80
5.2 The functionality provided by the tools in the MyXML tool suite 80
5.3 Table showing the device-independence components and the functionality they provide 85

6.1 Identification of MyXML dynamic content functionality on each page . . . 105
6.2 Device configurations for the VIF case study 108

7.1 Comparison of the DIWE framework with other approaches 129
7.2 Comparison of the DIWE framework with other approaches 130

Chapter 1

Introduction

1.1 Overview

Millions of pages and terabytes of information exist on the World Wide Web (WWW) today. The Web is a dynamic, constantly changing medium and it is the largest growing area of the Internet. With the advent of the WWW, the demand for Web sites (i.e., services) suddenly grew and many organizations realized the huge potential of the Web. The Web quickly became a powerful and important means to stay in contact with customers, provide online services, express opinions and make profit with e-commerce applications.

The primary language used on the Web is still the Hypertext Markup Language (HTML) supported by the Hypertext Transfer Protocol (HTTP). HTML was originally created because scientists at CERN were looking for ways to share information and documents over the Internet [BCL+94]. It was never expected to gain popularity this fast and it was not designed for the requirements we see in Web sites today: Web sites need to be manageable, changeable, and need to provide dynamic functionality for interaction with users. The typical Web development environment usually needs a combination of different technologies, tools and architectures.

Until the late 90s, the focus of Web service engineering research was the development of tools, technologies and methodologies for the design, implementation and maintenance of HTML-based Web sites. The common assumption was that a Web site would always be accessed by a browser found on a personal computer or a laptop. Recent developments in mobile computing software and hardware not only have changed this view, but have also increased the importance of *device-independent* access to Web content: The ability to access Web sites using a wide variety of *Web devices*. A Web device is any hardware or software that can be used to access Web content [LS99] such as telephones equipped with speech recognition software, digital televisions and Personal Digital Assistants (PDAs).

One of the next challenges faced by the research community and the World Wide Web Consortium (W3C) is the definition of standards, tools, methodologies and technologies for the “browser-less Web” and device-independent Web sites.

A major drawback of HTML has turned out to be its lack of support for device-specific content specification. An HTML Web page, with its tables, fonts, forms, etc., usually only adequately supports the display of a personal computer and may cause usability problems for

Web devices with smaller display and memory sizes (e.g., mobile phones). Further HTML drawbacks are the inflexibility to easily incorporate layout (i.e., presentation, user interface) design changes and the inability to reuse content embedded in HTML.

In order to eliminate HTML's shortcomings and to define extensible standards that address current Web requirements, the World Wide Web Consortium (W3C) defined the eXtensible Markup Language (XML) [W3C98a] and the eXtensible Stylesheet Language (XSL) [W3C00]. XML is a syntactic meta-language for defining content and other languages and XSL was proposed and designed because XML by itself does not contain any layout semantics. XSL can be used to add presentation information to content defined in XML. XML and XSL have gained popularity fast both in industry and in academia. These standards have paved the way in creating the device-independent Web by providing a basic flexible infrastructure to independently define content and layout information. This separation of layout and content allows the same content to be displayed on different devices by providing the appropriate presentation information.

XML and XSL alone are not sufficient to design and build device-independent Web sites that are easy to manage and that can be adapted to meet changing requirements. Users frequently expect interaction, personalization and up-to-date information. Often, major updates involving multiple documents and external information sources such as databases are necessary.

To support the increasing variety of devices used by people to access Web content, Web service providers and developers are increasingly concerned with the questions:

- How can a service be *designed* and *implemented* so that it is able to support different Web devices?
- How can we *make a service device-independent without increasing* the maintenance effort significantly?

This dissertation provides solutions to the questions and problems mentioned above. It introduces a novel XML/XSL-based design and implementation technique and a development tool suite to support the Web developer in engineering device-independent, interactive Web services. These services can be accessed by a wide range of Web devices such as mobile phones, PDAs with micro HTML browsers, speech-based Web interfaces and traditional full-fledged HTML browsers.

1.2 Contribution of this Dissertation

The integration of device-independence support into the Web service design, implementation and maintenance phases has not received much attention. Most solutions that have been proposed only tackle a part of the problem (e.g., Web access through *mobile* computing devices), but ignore the bigger problem of how to deal with device-independent Web access in general. These approaches do not always work when many different devices with varying display and memory sizes have to be supported.

The new generation of PDAs (e.g., the Compaq iPAQ) and mobile phones (e.g., the Nokia Communicator) are getting more powerful every day so limitations such as memory and CPU

power will probably become less important in the near future. Although the latest PDAs are even able to display frames, it is still important to adapt the content for these devices in order to provide a satisfactory surfing experience for users.

This dissertation introduces the notion of a device-independent Web service and defines it as a service that can be *extended* to support different Web devices of widely varying technical capabilities. It treats the *mobile Web access problem* as a special case of device-independence support.

My general thesis is that Web services can effectively be made device-independent if device-independence support is integrated into the Web site design, implementation and maintenance phases. Adaptation is not only the key to mobile information access [Sat96b], but to multi-device access in general.

To this end, the dissertation makes the following contributions to knowledge:

- A taxonomy for the comparison of device-independent Web site engineering approaches.
- A novel XML/XSL-based conceptual framework for building device-independent Web sites by using a reuse strategy. A constructed site can be easily extended by adding device-specific user interfaces to it and existing functionality does not have to be modified.
- The concept of *page-splitting and stepping* by layout marking so that the information on a Web page can be split into chunks to support devices with restricted memory or display sizes.
- The concept of *process-partitioning and stepping* by layout marking so that Web form-based interactions in a Web site can be divided into independent steps for interactions with devices that have restricted memory or display sizes.
- The concept of *device-specific XSL stylesheet pre-processing* for reusing existing XSL stylesheets to ease the overall maintenance effort.

All the concepts have been implemented and demonstrated in a prototype implementation that is available on the Web for download [1]. The prototype implementation, the *MyXML tool suite*, includes a visual integrated Development Environment (IDE) for engineering device-independent Web sites and supports device configuration, device maintenance and device-independent content authoring.

1.3 Structure of this Dissertation

This dissertation is structured as follows:

The next chapter gives a brief introduction to the Web engineering discipline and introduces basic terms and concepts such as XML, XSL and the World Wide Web service life cycle. It describes the device-independent Web site engineering problem.

[1] http://www.infosys.tuwien.ac.at/myxml

Chapter 3 presents the related work and discusses the different existing strategies and approaches to creating and supporting device-independent Web sites. It introduces a taxonomy for the comparison of device-independent Web site design and implementation approaches.

Chapter 4 presents an extended model of the traditional Web service life cycle that takes device-independence support into account. It presents the Device-Independent Web Engineering (DIWE) conceptual framework for engineering device-independent Web sites and discusses the novel concepts of page splitting, process partitioning and XSL stylesheet preprocessing.

Chapter 5 presents and discusses the MyXML tool suite for engineering device- independent Web sites. The tool suite is a prototype implementation of the conceptual framework presented in Chapter 4. The suite consists of the MyXML processor, three configurable run-time device-independence components and the MyXMLDesigner visual Integrated Development Environment (IDE).

Chapter 6 discusses the usage of the MyXML tool suite in the device-independent implementation of the Vienna International Festival e-commerce Web service. It shows how the tool suite was used to provide Web site access to full-fledged HTML browsers, PDAs and WAP-enabled mobile phones without the need to modify the existing functionality.

Chapter 7 evaluates the presented concepts and the MyXML tool suite. It discusses potential problems and lays out future work.

Chapter 8 summarizes and concludes this dissertation.

Chapter 2

Web Engineering basics

This chapter provides an introduction to the Web engineering discipline. It introduces basic technologies such as XML and XSL and discusses concepts such as the Web service life cycle. It describes the device-independent Web site engineering problem.

2.1 Terminology

I first define some basic terms that will be used with consistent meaning in the context of this dissertation.

The term *Web Service* has been used since the mid 90s to describe the information offered to users on a Web site (e.g., see [CFB00, ICL97, KJKS01, Sch97]), it is recently often being used to denote *browser-less* (i.e., machine) access to content on a Web site (e.g., see [Alp, dev, Sun]). Hence, to eliminate possible confusion and ambiguity, I make the following definitions:

- **Content**: The information that is offered to the user (e.g., the price for a book).
- **Static content**: Content that does not change at run-time. It is mainly stored in files on servers or in databases and is presented to the user without any processing (e.g., a home page defined in HTML).
- **Dynamic content**: Content that is generated at run-time based on the interaction with the user (e.g., an e-commerce application that presents a welcome text and lists the current items in a user's shopping cart).
- **Layout (i.e., user interface)**: The formatting information with which the content is formatted for presentation (e.g., fonts, graphics, buttons, tables, etc.).
- **Application logic**: The functionality that is necessary for providing the interaction and services to the users (e.g., maintaining the dialog between the user and the service that culminates in the purchase of a ticket.).
- **Web page**: Static or dynamic content on a Web site that is intended for browser-access and that is accessible through a unique URL.

- **Web service (or Web application)**: Functionality supported by one or more Web pages that provide some sort of interaction or information to the user for achieving a certain task (e.g., booking a ticket, retrieving price information, searching). The access to a Web service can be browser-less, or via browser.
- **Static Web service:** A Web service that returns static content.
- **Dynamic Web service:** A Web service that returns dynamic content.
- **Web site**: Collection of *Web pages* and *Web services* in a *single* administrative domain (e.g., the Web site of a company).
- **Web tool**: A software application that eases the construction of Web applications in some way.
- **Web technology**: An industry standard or a collection of Web tools for constructing Web applications.
- **Web engineer**: A Web developer who follows a systematic approach to construct Web services.

2.2 Web engineering: An emerging field

With the advent of the WWW, the demand for home pages suddenly grew; many organizational Web sites were initially created without a systematic approach by individuals who were interested in this new technology and who quickly gained basic knowledge of HTML. Although the ability of *anybody* to put *any* information on the Web has clearly contributed to the popularization and success of the Web, it also resulted in several problems that are found in many of today's Web sites.

First, because of the lack of understanding for the Web and hypermedia concepts, a single employee, often referred to as *webmaster*, was often designated to diverse tasks related to the Web site such as designing the information, the graphical look of the pages and the management and updating of information. The workload in many cases was too much for a single person to handle. Large and complex Web sites usually require a team of content providers and graphic, layout and interface designers. Indeed, management is a collaborative task [Str95]. Hence, many webmasters designed the Web pages according to *their* taste and picked the information that *they* found important. This sometimes conflicted with the business objectives of the management level and the image they wished to convey.

Second, webmasters did not have previous hypermedia experience in many cases and the lack of design guidelines showing what is good and bad on the Web resulted in excessive use of Web technologies such as frames and JavaScript. Furthermore, dynamic functionality (e.g., a Web-based database program for checking in and checking out books in a library) is often developed in an ad-hoc manner and most of the time the programs are script-based and not well documented or designed. This increases the management complexity of Web sites and makes maintenance difficult. Maintenance becomes especially sophisticated once the webmaster, not rarely the single person who has a complete understanding of the system,

leaves the organization. Some authors have referred to the current situation on the Web as the *Web crisis* (e.g., [GM01]) and have likened it to the *software crisis* (e.g., [She95]) in the 1960s when much of the produced software was not reliable and failed to reach basic levels of quality and user satisfaction.

Due to the nature of the Web, users expect a Web site to offer interactive and up-to-date content. Managing and maintaining a Web service, hence, usually becomes a challenging task once the number of services and the amount of offered information exceed a certain limit. *Web engineering* (e.g., [GM01, KJKS01]) is a discipline that deals with the systematic design, implementation, and deployment of large-scale, complex, Web-based information systems. It attempts to define processes and provide development tools that cover all phases in the life cycle of a Web service. The Web engineering discipline is young and there is consensus on the need for more evaluation, but many challenges remain, including issues such as scalability, multi-device access, increased performance, robustness, extensibility, maintainability and flexibility.

Much of the initial research on Web site design and development was based on the results of more than thirty years of hypertext research (e.g., see [Eng95, Nel95]) and a majority of Web engineering researchers came from a hypertext background. A Web site, after all, consists of a collection of hyperlinks. Although the WWW is not actually hypertext according to the Dexter Hypertext Model [HS94], hypertext researchers were quick to realize that many concepts involved in the design of hypertext are also applicable to the design and implementation of Web sites. As a result, several approaches emerged that integrated hypertext navigational structure considerations into the design process (e.g., [DIMG95, ISB95, SR95]).

As the demand for Web sites steadily increased and the amount of information grew, many sites started using relational databases to store and manage a large proportion of the offered information (e.g., news sites such as Reuters, CNN, portals such as Yahoo and e-commerce sites such as E-Bay). Hence, the database community also started working on Web site design and maintenance issues, but their focus mainly being the engineering of data-intensive, relational database-backed sites. Several approaches were proposed that adapted database concepts for Web site management and design. (e.g., [Goe98, CFP99, FFKL98]).

Since the mid-90s, the Web engineering field has been gaining popularity fast and researchers involved in this area possess all sorts of backgrounds such hypertext, data engineering, databases, library sciences, education, and lately even reverse engineering. The software engineering community, however, has been slow to pick up on the trend and to make a significant contribution with its knowledge. As a result, many of the well-known approaches for Web service design and implementation mainly concentrate on *static* or *database-based* content and fall short in supporting *dynamic* Web-based interactions such as those needed in e-commerce applications.

Hence, many Web applications and services are developed in an ad-hoc manner today and the main reason is the lack of practical methodologies, approaches and guidelines. Documentation, for example, is as important in Web engineering as it is in software engineering and unfortunately often equally ignored. One reason for this is sometimes the general misconception that the services being built are "simple" anyway, and are to be used only for "this year". A Web service, however, is often put together using a number of different technologies and dependencies. Due to the nature of the Web, the architectures of Web applications are distributed and not always easy to comprehend.

As the Web engineering discipline is becoming more mature, it is becoming evident that methodologies, tools, and technologies are needed that can effectively deal with the differing requirements for building Web-based information systems.

2.3 Web service characteristics and requirements

2.3.1 Information transfer characteristics

The World Wide Web (WWW) consists of the classical client/server model where clients (i.e., browsers) contact Web servers and request information. The information is returned to the client in a reply message. Users have to locate and retrieve the information actively and have to remember (or *bookmark*) the locations of services they are interested in.

One of the main reasons for the success of the Web is the possibility of integrating legacy applications, data sources and external services under a uniform, platform-independent interface (e.g., publicly available gateways provide access to libraries and flight booking systems that are often legacy applications with a Web interface).

The HTTP protocol used in the Web is stateless and insecure. A transaction management function often needs to be added to Web applications because of the lack of state and security. Most Web servers support the Secure Socket Layer (SSL) protocol for protecting Web communication against eavesdropping.

2.3.2 Stakeholders

Just like in software engineering (e.g., see [GJM91]), there are different stakeholders in Web site engineering projects: The *content managers* are responsible for providing and maintaining the content to be offered on the Web site. The *graphic designers* deal with the appearance of the Web pages in the site. The *Web engineers* have to develop the application logic and have to integrate it with the content and the layout information. Usually, one or more project managers are responsible for the timeliness of the project and the overall coordination. Finally, the *visitors* (i.e., users) of the site are the target audience that consume the offered information and use the services.

2.3.3 Basic Web service requirements

Each stakeholder in a Web site engineering project will have a different set of requirements for the Web site.

The content managers will mainly be interested in *easy-to-use update mechanisms*. They will need content management applications that allow them to edit, delete and enter information into the Web site and *versioning mechanisms* to enable them to keep track of changes in content and work concurrently.

The graphical designers will be interested in providing an *attractive, appealing graphical look* that will attract visitors and that will increase the acceptance of the Web site.

The visitors of the Web site will be mainly interested in *up-to-date content* and *usability*. If there is no *consistent navigational model* and the site is difficult to use, the typical visitor will leave and not come back again. This is because the attention spans of users in hypermedia environments are very low, and users are impatient [RM98].

Visitors will also want to use *different Web devices they have to access the content* in the site. A user, for example, will appreciate a Web service that provides a satisfactory surfing experience with her PDA. On the other hand, she will be frustrated if the information is difficult to find or access with the PDA.

The project managers will mainly be interested in decreasing the implementation and maintenance costs and will look for ways to decrease the time-to-market.

All these factors and requirements create a great challenge for Web engineers. The changes need to be integrated into the site swiftly, without the need for the *Under Construction* sign that is now infamous, and highly unpopular among Web surfers.

The Web engineers will look for the ability to *integrate off-the-shelf software components* to ease construction and for ways to *utilize information in legacy data repositories* to eliminate the need to redefine content. Further, they will aim to provide *location independence* in case the service needs to be migrated. Their overall goal will be to design and construct the service in such a way so that future requirements can be integrated with ease: They will attempt to construct *extensible, changeable* services.

The key to successfully dealing with all these requirements is to systematically cover all the phases in the *Web Service Life Cycle*.

2.4 Web Service Life Cycle

Some authors [NN95] have likened Web engineering to the software engineering process [GJM91]. There are fundamental differences, however. Web engineering includes some additional tasks: Data analysis, information architecting, navigation management and data organization. Using the software engineering process in Web engineering may be both difficult and inadequate [Sch98b].

Every Web service has a life cycle [TL97] that consists of a sequence of four major steps: *Requirements Analysis*, *Design*, *Implementation* and *Maintenance*. Most of the existing Web authoring systems concentrate on the implementation phase and only few provide support for the design stage. All stages, however, are important for Web services and have to be supported. Figure 2.1 depicts the Web service life cycle.

From the very beginning of the Web, tools first concentrated on content authoring using HTML. Later, more sophisticated tools were introduced that provided WYSIWYG support for authoring content. The next generation of tools started providing help in navigating, interface design and site management.

The vast majority of the available Web tools today are able to create pages and graphical layouts using simple templates, but lack support for handling major updates involving multiple documents, dynamic data, and the integration of external information sources such as databases.

A typical Internet development environment is still quite fragmented. A combination of

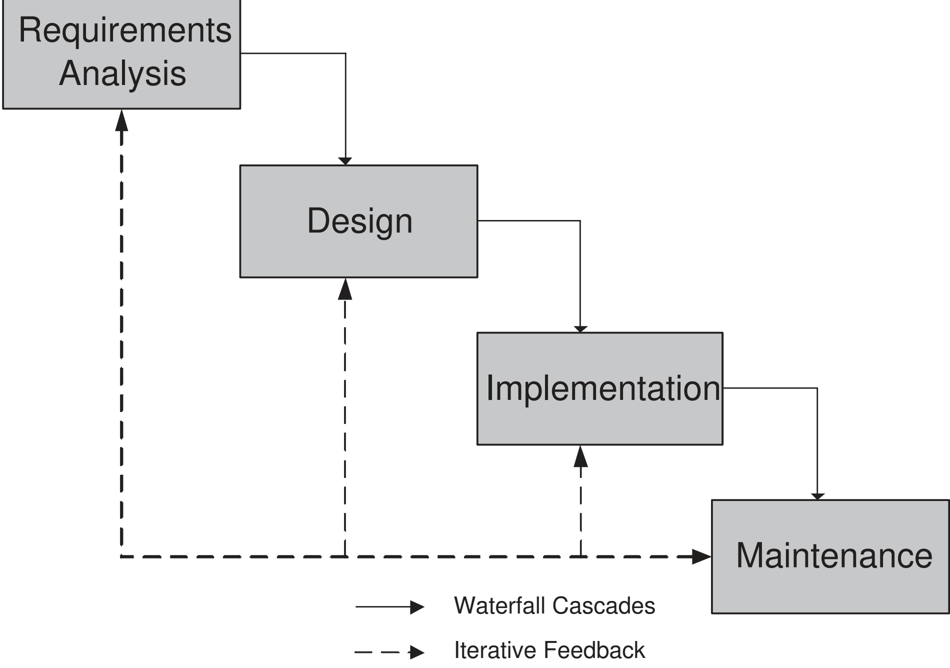


Figure 2.1: Life Cycle of a Web Service [Sch98b, TL97]

many tools is necessary to implement a Web service. Several alternative approaches have been introduced (e.g., [GWG97a, Mau96]) that attempt to support all phases of the Web engineering process.

2.4.1 Requirements Analysis

The first step in Web engineering is to analyze what type of information needs to be provided and in what way. Standard software requirements analysis is often necessary when interactive services need to be provided and Web applications need to be written.

All stakeholders are involved in this phase and each state what they expect the site to do. The Web engineers usually do not have explicit requirements. Extensibility, for example, is a valuable asset and a requirement for the Web engineer, but not really a requirement for the other stakeholders.

The Web engineer's job is to fulfill the requirements of the other stakeholders. If the Web services that are designed and constructed can easily be modified and extended, it will make the Web engineers' lives easier once requirements start to change in the future. Clearly, Web engineers have the main responsibility in building maintainable, extensible Web sites because other stakeholders are mainly interested in *having* their requirements covered, but not in *how* they are implemented.

2.4.2 Design

The information collection is organized in the design phase and an architecture of the service is defined.

Different stakeholders are involved in the design phase. The graphical designers provide layout mock-ups of the Web pages and use them to get feedback from prospective visitors and other stakeholders. The usability of the mock-ups and the graphical appearance are evaluated and improved in an incremental process.

Content managers identify the information that will be offered to visitors and plan and coordinate how it will be inserted into the site.

The Web engineers design the architecture of the Web service application logic and plan the integration of the content and the layout. Further, they design the content update mechanisms that will be used to insert content into the site.

The project managers coordinate the activities between the stakeholders, organize regular meetings and keep track of the progress.

2.4.3 Implementation

In the implementation phase, the information and functionality planned and organized in the design phase is coded in an appropriate format.

Most Web sites use HTML files to deliver static content. These HTML documents can be written using editors or generated from relational databases using widely available Web tools.

The functionality and support for interactions is usually implemented using popular Web technologies. Most of these technologies generate dynamic content by either writing HTML to a stream that is sent back to the calling client or by mixing layout information with application logic in files that are interpreted at run-time by an *application server*. An example of the first form of interaction are the Perl script[Pag], Java servlet[Jaw98], and C#[Arc01] technologies and an example of the second form are the PHP[RSS$^+$99] and Coldfusion[col] technologies.

Scripting languages can be *server-side*, or *client-side*. Perl, for example, is a server-side scripting language that is interpreted on the Web server. Javascript, on the other hand, is embedded into HTML and is interpreted locally on the user's browser.

Usually, a combination of different technologies are used to implement the interactive functionality. A server-side Perl script may be used in an e-commerce application, for example, to check a relational database for shopping cart information. To save bandwidth, the user's input may be validated on the browser using a client-side Javascript before it is sent.

2.4.4 Maintenance

Service maintenance is one of the most important and costly issues in Web Engineering. Similar to software management, the handling of a Web service becomes non-trivial once its size increases [Sch97].

Most Web sites today change their appearance at least once a year to stay attractive. Minor changes in the look-and-feel of a site several times a year are very common, and major modifications are not rare.

Modifications are motivated by better understanding of user needs based on previously gained feedback, new requirements, optimization strategies and new market directions.

Service maintenance involves **information updates** and **content management**, **navigation management**, **version management** and **service migration**.

Ad hoc navigational links are embedded almost anywhere in Web services. Unfortunately, links may be broken due to the nature of the Web. Navigational management is necessary for checking the validity of the links and resources for consistency and integrity.

Service migration is the movement of a part of, or the entire Web service, to another host. Service migration is often necessary as hardware is updated, performance requirements change, and new versions of software components become available.

Version management is an important issue in service maintenance because it allows Web engineers to issue releases of scripts and source code and keep track of functionality changes. Furthermore, a versioning system allows content managers to work concurrently. Versions increase the manageability and maintainability of the service – especially when dynamic content is involved.

Versioning also allows the analysis of the evolution of the site. By checking the logs, site-specific information can be retrieved such as the pages that had to be updated regularly and those that did not change much. This information can be utilized to maintain and adapt the services according to the users' needs.

Standard versioning systems, such as CVS, [cvs] may be deployed for version management.

2.5 Flexibility

A flexible Web service is a service that is easy to extend and maintain. The modifications in the graphical layout and the look-and-feel of the service is one important flexibility issue for Web services. The most important aspect of flexibility, though, is the ability to integrate new functional requirements without having to do major modifications to the system.

The first generation of HTML document standards lacked support for layout flexibility. Attempts were later made to eliminate these shortcomings by extending the HTML standard with technologies such as the Cascading Style Sheets [W3C]. CSS defines common formatting properties such as font size, font family, font weight, paragraph indentation and paragraph alignment. One can specify, for example, that all *H2* HTML elements should be formatted in 24pt Times New Roman font. Multiple stylesheets can be applied to a single element and the styles then cascade according to a particular set of rules.

In order to eliminate HTML's shortcomings and to define extensible standards that meet requirements such as layout flexibility, the World Wide Web Consortium (W3C) defined the eXtensible Markup Language (XML) standard along with the XML Style Sheet Language (XSL).

Both technologies are important for Web engineering because they are standards and have gained popularity fast. Many software vendors are integrating XML and XSL support into their products and a wide range of XML/XSL-based tools are available today such as editors and configuration tools.

2.5.1 XML

XML is a set of rules for defining semantic *tags* that break a document into parts and identify the different parts of a document. It is a meta-markup language that defines a syntax used to define other domain-specific, structured markup languages [Har99].

XML is not just another markup language such as HTML. HTML defines a fixed set of tags (e.g., H1 for Heading 1, H2 for Heading 2, etc.) that describe a fixed number of elements. The main difference of XML is that it is a markup language in which one can define tags as one wishes. These tags must be organized according to certain general principles, but their meaning is flexible.

Suppose we would like to describe students by noting their name, age and computer science knowledge. We can create tags for each of these. The XML definition of this information may look something like this:

```
<?xml version="1.0"?>
<student>
      <name> Engin Kirda </name>
      <age> 28 </age>
      <knowledge> Expert :-)) </knowledge>
</student>
```

This listing uses meaningful tags such as *age* and *name* that we defined.

The tags we defined can be documented in a *Document Type Definition* (DTD). The DTD [W3C98b] can be thought of as a vocabulary and a syntax for certain kinds of documents. XML definitions do not necessarily need to have a corresponding DTD. A DTD merely allows the *validity* (i.e., the conformance to the syntax defined in the DTD) of XML information to be checked. All XML documents, however, have to follow a specific set of rules such as having a header at the beginning and having a closing tag for every opening tag (e.g., if there is an <engin> tag, then there *must* be an </engin> closing tag). XML documents that conform to these specific set of rules are said to be *well-formed*. Well-formedness is the minimum requirement for XML information.

While one might find it useful to write documents that use a single markup vocabulary, it is sometimes even more useful to mix tags from different XML definitions. The problem, however, is that when mixing tags from different XML definitions, one might find the same tag used for two different things. In an e-commerce related XML definition, for example, the tag *name* could refer to the name of an article rather than the name of a student as in the previous example. *Namespaces* disambiguate these instances by associating a Universal Resource Identifier (URI) with each tag set and attaching a prefix to each element to indicate which tag set it belongs to. Thus, one could have a *students:name* tag and an *articles:name* tag.

Unlike HTML, XML does not describe the layout (i.e., formatting/presentation) of the elements on a page. It describes a document's structure and meaning and *only* contains tags that say what is in the document and not *how* the document should be presented.

A layout can be added to an XML document with a stylesheet. For this purpose, XSL is used.

2.5.2 XSL

XSL is an advanced stylesheet language specifically designed for use with XML documents. In fact, XSL documents themselves are XML documents.

XSL is divided into two parts: transformations (XSLT) and formatting objects (XSL:FO). XSL:FO is a language for describing 2D layout of text in both digital and printed media. XSLT, on the other hand, is a language for transforming one XML document into another textual format.

XSL documents contain a number of rules (called *templates*) that apply to particular patterns of XML elements. An XSL processor reads an XML document and compares it to the rules in the stylesheet. Whenever a rule is recognized, transformation rules are invoked and corresponding output text is generated. Unlike CSS, the output text is arbitrary and is not limited to the input text plus formatting information. XSL is far more flexible and powerful than CSS and it is better suited to XML documents. XML documents can also be easily converted to HTML documents with CSS stylesheets.

The following simple XSL stylesheet prints the HTML fragment "
This is some name
" for every student *name* tag defined in the previous XML definition. Every time it recognizes a *student* tag, it recursively processes student names.

```
<?xml version="1.0" ?>
<xsl:stylesheet
      xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
<xsl:template match="student">
      <xsl:apply-templates/>
</xsl:template>

<xsl:template match="name">
        <br>
This is some name
<br>
</xsl:template>
```

Template rules defined by the **xsl:template** element are the most important part of an XSL stylesheet. Each template rule is an xsl:template element. These associate particular input with particular output. Each xsl:template element has a **match** attribute that specifies which nodes of the input document the template is instantiated for.

To get beyond the *root element* (i.e., the first tag, *student* in our example), the XSL processor needs to be told to process the children of the root. In general, all child elements are recursively processed using the **xsl:apply-templates** directive.

XSL provides advanced functionality such as *conditional loops*, *if/then/case* directives and a powerful mechanisms (i.e., *XPath*) for selecting elements.

One frequent use of the XML and XSL technologies is to create flexible static content for *multi-purpose* publishing[1] [LS99]. Using XSL, the content in XML is transformed into an appropriate format for different target devices.

Although the XML and XSL standards have created a basic flexible infrastructure to independently define content and layout information, they are not sufficient alone to design and build device-independent Web sites that are flexible and maintainable.

2.6 The device-independent Web engineering problem

This section discusses the device-independent Web site engineering problem that this dissertation tackles. It presents simplified, as well as real-world examples to define and illustrate the problem.

2.6.1 Historical overview

The problem of device-independence is not new in computer science. Since the very early days of computing, computer displays and hardware have always had widely varying technical characteristics. Hence, differences such as display sizes and graphical capabilities had to be supported by operating systems and programs. Modern operating systems provide device abstractions to programs and support different devices using drivers.

The situation was similar for the Web in the mid 1990s: Many users existed that did not have access to graphical browsers and were using browsers such as Lynx on dumb terminals with text-only characteristics. As a result, it was considered good Web design practice to offer the content in pure textual form (without graphics) as well as in a more appealing graphical look. The reader familiar with the Web since its early days will remember pages that had a "text only" link in the navigation bar. Another solution that was often used to deal with limited text-based devices was to keep the design of the HTML-pages as simple as possible so that all browsers and displays could satisfactorily cope with the rendering of the content.

In fact, the original HTML definition did not contain elements such as <font> and the font *size* attribute. Generic font type and size tags such as <h1> and <h2> were used that were device-independent: The *browser* interpreted the size and fonts of headings according to user settings or the device characteristics.

As concepts such as *corporate image* or *identity* [Qui94] started emerging and gaining in importance, however, the demand for more functionality grew and companies such as Netscape and later Microsoft started expanding the HTML element set to meet the demand. As a result, HTML incompatibilities occurred because of the different HTML namespace implementations. Unfortunately, this is still the case sometimes in Web development. It is not uncommon, for example, for a table to look quite different on a browser such as Netscape when compared to the Internet Explorer. These differences are the main reason why Web

[1] Sometimes also called syndication

companies and customers usually agree on a browser in projects that will be guaranteed to work with the provided functionality.

With the increase in available functionality, the trend of supporting alternative, simpler text interfaces largely disappeared. Instead, users visiting a site were often “encouraged” to download a newer version of a browser (e.g., the infamous “This site is best viewed with Netscape version...” type messages) and the common assumption that a user would at least have 600-pixel width screen estate started to establish itself among Web designers. Many Web sites today require at least a mid-size display (i.e., minimum 800x600 pixel size) for a satisfactory surfing experience and HTML extensions such as frames cause problems on smaller displays. Figure 2.2 illustrates the difficulty of supporting small displays. Note that the user cannot see a large proportion of the information on the site. Much scrolling is required, thus, increasing the cognitive overhead and decreasing usability [RM98].

Figure 2.2: The difficulty of supporting small displays: The DSG homepage as seen on an iPAQ PDA

Even though most browsers conform to the W3C HTML standards that were later agreed upon, browser-specific (and therefore device-specific) functionality already exists and elements that are not device-independent such as <font> have therefore been standardized.

Figure 2.3 depicts the differences in design between the 1995 version of a commercial Web site (the Vienna International Festival home page) and the 2001 version. Note how much simpler the 1995 version is compared to the newer version. Tables are used extensively by the graphical designers in the 2001 design.

Recent developments mobile computing software and hardware (e.g., Wireless Access Protocol (WAP) access provided by mobile phone providers) and speech technology (e.g., the definition of the VoiceXML XML language for defining speech-based Web applications) have highlighted the need for device-independent Web access, once again [BFJT01]. The

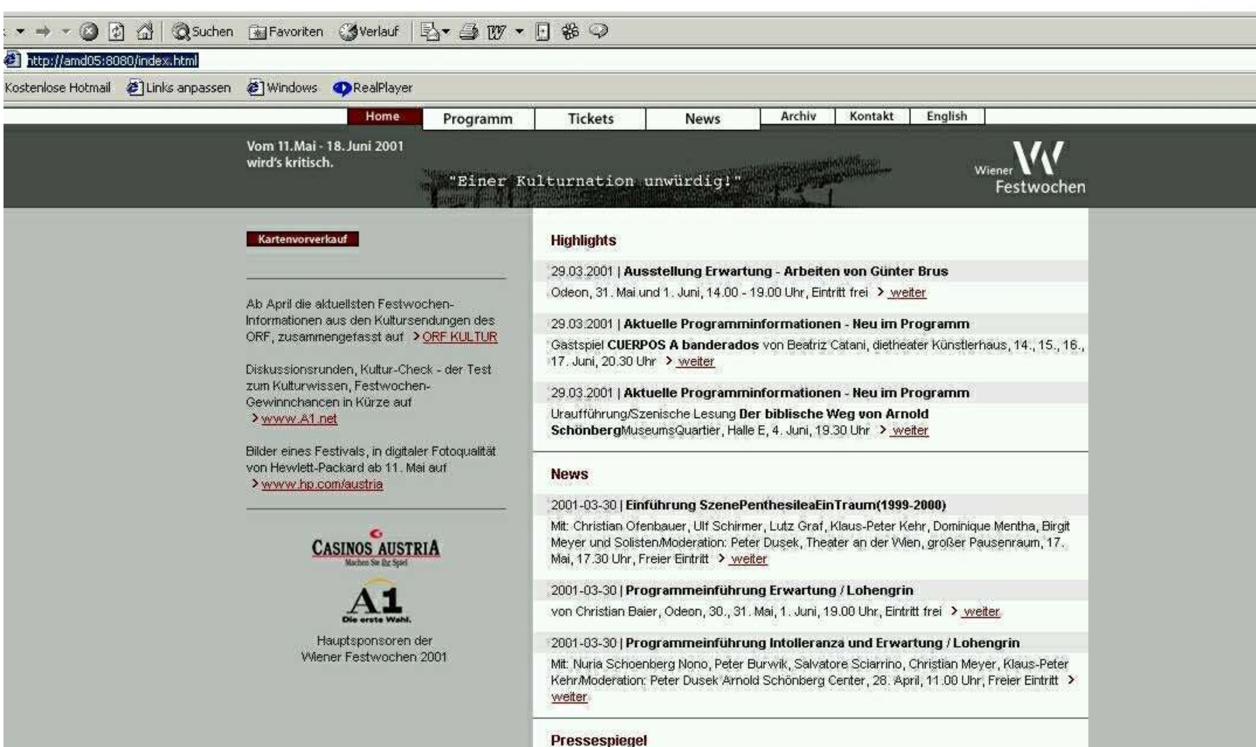


Figure 2.3: Screenshots of the 1995 and 2001 VIF home pages

challenge, however, is greater this time. The Web site has to be accessible by users using Web devices that have a wide range of display sizes and memory limitations, and that may require a special XML-based *Web format* (e.g., the Wireless Markup Language (WML) is an XML language that has been specially designed to describe small pages that can be accessed by WAP-enabled mobile phones).

The next section describes and illustrates the device-independence problem from the Web engineering point of view.

2.6.2 Problem: Constructing maintainable, interactive device- independent Web services

One Web-based mobile computing service that has become quite popular in the last couple of years is providing custom-tailored information for PDAs that users can download from a Web site for offline-browsing.

Varnum in [Var00], for example, discusses how PDA-services were deployed at Ford and presents an experience report. The problem was that managers and company leaders that were higher up in the company hierarchy did not have any time to get information from the company intranet. These people were always busy and only had time between meetings. They preferred to get their emails in paper form and had time for correspondence in cars, during flights, etc. Interestingly, though, it was observed that PDAs had found a high acceptance by these people.

The IT department decided to utilize the wide usage of PDAs (i.e., Palms in this case) and developed a system with which Web sites in the intranet can be downloaded to these devices. The interactions are offline and any forms submitted are queued in the PDA. All requests are sent once the PDA is synchronized and connected to the PC.

Only some services are offered for PDAs and the server-side scripts offering these services had to be modified or duplicated. One observation was that it pays to make low use of images on the Palm output. Users are primarily interested in acquiring information and the images do not serve an important navigational purpose on low-resolution displays. Even after eliminating graphics, though, there is still precious little screen space available. Furthermore, although the PDA is able to render tables, simpler pages render faster. Thus, performance considerations played a significant role in designing the pages.

The server-side scripts were in Perl and it was not too difficult to modify them in this case.

The problems, however, were: 1) There was little logic reuse – hence making code maintenance more difficult as the site grows, and 2) The modification of the Perl scripts is an ad-hoc solution and although it solves the problem, the solution is temporary and does not guarantee that the services will be able to support other Web devices and formats in the future as the requirements evolve. Supporting a speech interface using VoiceXML, or creating a PDF version of the information in the intranet for the managers, for example, would need a considerable implementation and maintenance effort.

In the Distributed Systems Group at the Technical University of Vienna, we experienced a similar problem. We had a Web-based grading service that enabled the students to look up the grades they had earned in courses. This service was a script-based solution using Perl.

```
package grading;

dbmopen (%files,"files",undef);

#-- gradingCodeLayout --------------------------------------------
$classes{"gradingCodeLayout"} = '

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>

<TITLE>  Engin Kirda  </TITLE>

<BODY BGCOLOR=#FFFFFF  TEXT=#000000
  LINK=#0000FF  ALINK=#000000  VLINK=#800080 >

<table border="0" width="100%" cellspacing="0" cellpadding="0">
  <tr>
     <td bgcolor="#000000" VALIGN="CENTER">
     <img src= images/title.gif  ALT="Document Title">
     </TD>

  </tr>
</table>
<table><tr><td>
Grades $forwhom:</td></tr>
<tr><td>
$content</td></tr></table>';
```

Figure 2.4: Part of the Perl script implementing the HTML grading service

```
package grading;

dbmopen (%files,"files",undef);

#-- gradingCodeLayout --------------------------------------------
$classes{"gradingCodeLayout"} = '<?xml version=\"1.0\"?>
<!DOCTYPE wml PUBLIC \"-//WAPFORUM//DTD WML 1.1//EN\"
             \"http://www.wapforum.org/DTD/wml_1.1.xml\">
<wml>
<template>
      <do type=\"prev\"> <prev/> </do>
</template>
<card id=\"result\" title=\"Query Results\">
<p>
Grades $forwhom:</p>
$content

</card>
</wml>
';
sub gradingCodeLayout # call as gradingCodeLayout
      ($content,$forwhom,$httproot,$now)
   {
    local ($content,$forwhom,$httproot,$now) = @_;
    $_ = $classes{"gradingCodeLayout"};
    eval qq/"$_"/;
    }
```

Figure 2.5: Part of the Perl script implementing the WAP grading service

After seeing that WAP services were being offered by Web sites such as banks and cinemas, we decided to offer the grading service we had through an additional WAP interface.

Figures 2.4 and 2.5 depict parts of the Perl scripts that provide the functionality. The layout information (e.g., tags such as <html>, <wml>, <table>) are directly hard-coded into the source code. Note, also, that there is a considerable amount of overlap and duplication between the scripts. This approach is typical and the easiest solution to building Web services that can support more than one device. It clearly does not scale and may cause maintenance nightmares. If, for example, there is a need to generate a PDF report for the student grades, the source code has to be *copied* and modified in an ad-hoc manner to incorporate the new requirement. This approach would solve the problem for a while, but whenever there is a need to change the underlying application logic, the duplicated logic would have to be modified as well.

The described problem is wide-spread when popular, traditional technologies such as Java servlets, PHP and ASP are used. Figure 2.6 shows a fragment of the Java servlet code from a commercial Web site that provides shopping cart functionality. The servlet displays the contents of the user's shopping cart in HTML. The entire HTML information is intermixed with the content and the application logic and is hard-coded into the code.

One source-code level solution would be to integrate device-specific content and layout information into the application logic for every new device. This integration, however, is not necessarily easy because it involves the analysis and modification of the code. This can be an error-prone and expensive task. It may become especially difficult when different display sizes have to be supported and memory limitations exist.

```
 // Go through the Shopping List and print everything...
for (int i=0; i<=eventList.size()-1;i++) {
   ShoppingCartEvent event = (ShoppingCartEvent)
      eventList.elementAt(i);
   database.getEventInfo(event.getEventId());
   out.println
   ("<tr><td colspan=\"3\" align=\"center\"><font size=\"2\"><b>"+
           database.getEventTitle()+"</b></font></td></tr>");

   if (!database.getSecondEventTitle().equals("Nothing")) {
      out.println("<tr><td colspan=\"3\" align=\"center\">"+
               "<font size=\"2\">Shown together with:<BR><b>"+
            database.getSecondEventTitle()+
                           "</b></font></td></tr>");
   }
```

Figure 2.6: Part of the VIF 2000 servlet code implementing a shopping cart

The presented examples show that the main device-independence Web site engineering problem is the increase in maintenance complexity as the number of devices that need to be supported grows. Because Web sites are usually not designed to support Web devices of varying technical characteristics, it is sometimes difficult and costly to integrate support for a new device.

A maintainable, higher-level solution is needed to support the design and implementation of interactive, device-independent Web sites. The solution has to cover the following main

requirements:

1. It should provide support for the different phases in the Web service life cycle.
2. It should support both static and dynamic content.
3. Application logic *reuse* should be possible so that the logic *does not* have to be duplicated. The *same* logic *needs to work* without modifications with *any* Web device no matter what its display and memory size is.
4. It should be possible to provide the content in the site in *any standard XML Web format* (e.g., VoiceXML, WML).
5. It *should not* increase the maintenance effort significantly.

I define the notion of a *device-independent Web site* in this dissertation as a site that is *flexible* and can be *extended* to support different Web devices of widely varying technical capabilities and propose a solution that fulfills the requirements listed above. I present a novel XML/XSL-based Web service design and implementation technique that allows the systematic construction of device-independent, flexible Web sites. New Web device support can be added to the Web sites with ease and existing functionality does not have to be modified.

2.7 Summary

This chapter provided a brief introduction to the Web engineering discipline. It introduced basic technologies such as XML and XSL and discussed the Web service life cycle and service flexibility. It described the device-independent Web site engineering problem and defined the goals for a solution that allows the engineering of maintainable, device-independent Web sites.

Chapter 3

Related Work

Much research has been done since the early 1990s on Web service design techniques, methodologies and development tools. Most of the existing work focuses on the construction of *HTML-based* Web services. Since the beginning of the year 2000, device-independent Web engineering has been receiving growing interest.

This chapter presents related work. First, it describes and discusses traditional Web engineering approaches and mobile Web access techniques that *do not explicitly* attack the device-independent Web engineering problem, but that are relevant and important as background work. Second, it introduces a taxonomy for classifying and comparing the solutions that *explicitly* tackle the device-independent Web engineering problem and third, it describes and evaluates these approaches.

The next section gives a brief overview of research on device-independent Web access.

3.1 Brief overview of research on device-independent Web access

The majority of the authors describe the *mobile* information and *Web access* problem (e.g. [Sat96b, KAK^{+}00]). Many conferences and workshops are being held that address problems related to information access from mobile computing devices with restricted capabilities such as mobile phones and PDAs.

The term *mobile e-commerce* [Sen00] has also recently gained popularity. There is a general expectation that much commerce over the Internet and the Web will be performed via mobile devices in the next decade (e.g. [Gla01]).

At the same time, there is another growing market: Web access via speech recognition and synthesis technologies. This application area is especially important for companies involved in the speech technology market. Speech recognition systems are already being deployed in many organizations such as airports and banks. They allow customers to call by phone and retrieve information such as flight information and the current account balance. Providing speech access to the Web, thus, is interesting for these companies.

Research in speech-based Web access has led to the specification and development of VoiceXML [Luc00]. VoiceXML is an XML-based language that allows interactive speech

applications to be written that provide access to Web content. It has been quickly adopted by companies and the number of VoiceXML development environments and tools are increasing every day.

Ralph in [RS01], for example, looks at WAP's "failure". Much hype was involved in marketing WAP and users' expectations were not met. Interactions with WAP devices are usually so difficult that, according to Ralph, speech interfaces based on VoiceXML will increase in importance.

Ralph also states that British Telecom (BT) has been experimenting with *Portia*, a corporate voice portal that provides a voice interface to the systems that people use every day in the course of their work. It was discovered that people that are using a portable laptop or a PDA also use Portia. The experimental usage results were promising and users use Portia because it seems to be quick. Portia has been running in a trial setting with 200 users (see [RS01]).

Clearly, the problem is not only *mobile* or *speech* access to the Web, but device-independent access in general. As a result, a device-independent Web working group was established within the World Wide Web Consortium last year and this group aims to address general device-independence issues that are related to Web access from a wide variety of fixed and mobile devices such as watches, televisions, telephones, PDAs and mobile phones.

3.2 Traditional Web engineering approaches

Traditional HTML-based Web engineering approaches and tools have been classified in the past (e.g., [Fra99, Sch98b]) as belonging to four groups: Page-based editors, site management tools, Web service models and object-based approaches.

Among the phases in the life cycle of a Web service, the design phase is usually the one that is either ignored or that receives less attention (e.g., [GM01, KJKS01]). Since the mid 90's, the special importance of the design phase in the Web service life cycle has been identified by many authors (e.g., [BMY95, BN96, NN95, Qui94, Str95]). Several models and methodologies have been proposed for the construction of Web services and hypermedia systems.

3.2.1 The Dexter hypertext reference model

The Dexter Hypertext Reference Model [HS94] is the most influential hypermedia reference model in literature. It was defined because many hypermedia systems existed and it was difficult to classify and compare them. Because of the existing differences, it was important to capture the significant abstractions both formally and informally.

The Dexter Hypertext Reference Model consists of three layers: The *Within-component*, *Storage* and *Run-time* layers. The Within-component layer covers the content and structures within hypertext *nodes*. The Storage layer describes the network of nodes and links that is the essence of hypertext. The Run-time layer describes mechanisms supporting the user's interaction with the hypertext.

The model focuses on the storage layer and the mechanisms of *anchoring* and presentation specification that form the interfaces between the three layers. The fundamental entity in the storage layer is the component. A component is either an atom, a composite entity or a link made from other components.

At the time the Dexter Hypertext Reference Model was defined, no hypertext systems existed that had to support more than one type of layout. The model, however, is quite flexible and there are no restrictions that only a single layout has to be built on top of the Storage and Within-component layers.

Unfortunately, the Web is not hypertext according to the Dexter Model because a storage layer that contains a database of nodes (i.e., content) and links does not exist. "Broken" links can exist on the Web whereas this is not possible in a hypertext system.

Another shortcoming of the Dexter model is that it does not take application logic into consideration.

3.2.2 The Relationship Management Methodology (RMM)

The Relationship Management Methodology (RMM) [DIMG95, ISB95] for building hypertext applications is well-known in the Web engineering community. It is one of the first attempts to define guidelines for the systematic construction of Web applications (i.e., hypertext).

RMM is based on a data modeling language, Relationship Management Data Model (RMDM), that is developed by the authors and based on the Entity Relationship (ER) Model (e.g., [TYF86]) used in database modeling.

The methodology is based on the traditional software engineering process and focuses on the design, implementation and construction phases for hypermedia applications. It has seven steps for hypermedia service management: 1.) *ER Design*, 2.) *Slice Design*, 3.) *Navigational Design*, 4.) *Conversion protocol design*, 5.) *User-Interface screen design*, 6.) *Run-time behavior design* and 7.) *Construction*. Steps four to seven are tasks beyond the modeling of hypermedia information and must either be done manually, or by using tools that provide automated support for these steps. (e.g., RMC [DIMG95]).

The RMM Methodology is well-suited for applications that have a regular structure, especially where there is a frequent need to update the information to keep the system current. Traditional sites that rely heavily on a RDBMS can benefit from the usage of this methodology.

The main restriction of the methodology is that it has no support for the design and integration of application logic.

3.2.3 Object-Oriented Hypermedia Design Methodology (OOHDM)

The Object-Oriented Hypermedia Design Methodology [RSL99, SR95, SRB96, Sd98] (OOHDM) consists of four steps. The methodology uses Object-Oriented (OO) concepts and techniques for systematically building hypertext applications.

The OOHDM steps are 1.) *Domain Analysis*, 2.) *Navigational Design*, 3.) *Abstract Interface Design*, 4.) *Implementation*.

In the domain analysis step, a conceptual model of the application domain is built using well-known OO modeling principles. The model is augmented with some primitives such as users and tasks.

In the navigational design stage, the navigational structure of the hypermedia application is described in terms of navigational contexts that are induced from navigation classes such as nodes, links, indices and guided tours. Links are derived from conceptual relationships defined in the first step.

In the abstract interface design phase, the abstract interface model is built by defining perceptible objects (e.g., a picture, a city map, and for so forth) in terms of interface classes. Interface objects map to navigational objects, providing a perceptible appearance.

Finally, in the implementation phase, interface objects are mapped to implementation objects.

Just like RMM, OOHDM assumes that Web services are merely hypertext. The main focus of the methodology is the design of the navigation, but no real support is provided for the implementation and maintenance stages.

In theory, it could be possible to use OOHDM to model the user interfaces and navigation for different devices that a Web service supports.

3.2.4 W3DT and eW3DT

The WWW Design Technique [BN96] (W3DT) has been proposed by Bichler and Nussler. The authors present observations that have a high practical value. In [BN96], the authors identify the problems on the Web well and also note the insufficiency of traditional hypertext modeling methodologies such as OOHDM and RMM.

The difference of their technique, they state, is that it has been designed for large *Web sites* (in contrast to hypertext). Analogous to our previous discussion, the authors note that although the Web is based on hypertext, it is not really hypertext according to the Dexter model. Their paper identifies the importance of communication between the different parties involved in a Web project (users, designers, application developers, etc.). Furthermore, it draws attention to distributed services and states that Web services might be distributed across organizations and corporations. It notes that design mechanisms and methodologies are missing in this area.

According to Bichler and Nussler, models are needed for communication between management, end-users and programmers. These models help to avoid structural inconsistencies and the reusing of global structures of applications becomes possible.

W3DT is a simple-to-use graphical methodology. They have also implemented a simple tool that provides support during the graphical modeling. The tool generates HTML templates and CGI code from the model.

Scharl and Bauer [BS00b, Sch98a] have extended W3DT and called it the Extended W3DT (eW3DT). The ideas they present attack the problem of meta-modeling Web-based Information Systems (WISs) for communication between users, managers, designers and

implementors. It presents a graphical representation of Web interactions[1].

The graphical models the authors present are HTML-based (e.g., they are graphical notations for HTML pages), the content is embedded in HTML pages and there are no considerations for device-independent access.

3.2.5 Webcomposition and W3Objects

Webcomposition [GWG97b, GGS+99] concentrates on the manageability and maintainability of hypertext services and extends OOHDM. A Web application is decomposed hierarchically into so called components. At the higher level, a component may model a page or even a site. Further down, a component relates to parts of HTML pages such as tables and navigation bars. The Webcomposition model allows the sharing of components and prototype documents.

A component in Webcomposition can be associated with any complete resource such as an HTML page or a Perl script generating an HTML page.

Some of the ideas presented in Webcomposition are quite similar to W3Objects [DMCS95, ICL96, ICL97]. In W3Objects, components are simply called objects. Different views can be built on services and 'components'. So called *W3OScripts* are able to access the functional interface of a service. This mechanism can be used to include other views as components.

Both W3Objects and Webcomposition provide support for covering all the phases in the Web service life cycle.

3.2.6 Strudel

In [FFKL98], the authors present a Web site management tool, Strudel, that adapts database concepts for Web site management. The key idea of the tool is the separation of the structure, content and visual presentation of Web sites.

The designer first creates a uniform model of all the information in the site. Then, the builder builds the site using a query language – StruQL.

Strudel is based on a semi-structured data model of labeled, directed graphs. This model was introduced to manage semistructured data, which is characterized as having few type constraints, irregular structure, and rapidly evolving or missing schema.

One disadvantage of the tool is that existing data needs to be integrated using wrappers and scripts written by hand. Furthermore, the authors state that Strudel does not have any dynamic content generation support.

The layout, in Strudel, is integrated using HTML templates. The authors state that the usage of HTML templates in their system have many advantages. The usage of HTML templates, however, is not new and many industry tools such as PHP [RSS+99] and Coldfusion [col] provide similar functionality. One disadvantage of using HTML templates is that they do not support complex navigational structures.

[1]Note that the same problem was picked up by Conallen later and he extended UML to model Web interactions – see [Con99]

3.2.7 Araneus

Araneus [AMM+98a, AMM98b] aims to define an environment for managing unstructured and structured Web content in an integrated system called Web-Based Management System (WBMS). A relational database is used to store data and meta-data about the structural information.

The Araneus system has a conceptual model and a design process. First, the database is modeled using the traditional EER [TYF86]. Then, the hypertext conceptual modeling formalizes navigation by converting the EER schema into an Navigation Conceptual Model (NCM) schema. The implementation is done using page-schemas in the *Penelope* language that specifies how the physical pages are constructed from the content in the database and the logical page schemes.

One disadvantage of Araneus is that it requires a proprietary HTML-dependent template language for specifying the layout. Furthermore, it does not have any support for application logic integration.

3.3 Mobile Web access techniques

Initially, much of mobile computing research concentrated on operating system, file, resource and data management support for mobile users mainly carrying laptops (e.g., [LB96, MES95, Sat96b, Sat96a, Sat89]). As the importance of the Web increased, more people have started working on mobile Web access problems and some have even predicted that one of the next big challenges of the Internet is mobile access to Web content (e.g., [AF99, Fra97]).

Several *transcoding* techniques have been proposed that attempt to convert and adapt content available in HTML to be viewable on mobile devices. The quality of images, for example, may be decreased at run-time for devices that have limited memory sizes. Another example is displaying images of varying quality to the user based on the available bandwidth. Some approaches try to automatically convert content available in an unsuitable form (e.g., HTML with frames) to a suitable form (e.g., WML, HTML without tables, etc.). The aim of these approaches is to provide "intelligent" algorithms that can convert the content with minimal information loss and provide a satisfactory surfing experience for users.

Some researchers are focusing on *summarization* techniques that attempt to automatically summarize Web content by extracting important information and making it viewable on devices with small displays or memory limitations. Rules have to be often set up with which summarization and extraction can be 'guided'.

Existing summarization approaches belong to two classes [HM00]: Knowledge-poor and Knowledge-rich approaches.

Knowledge-poor approaches rely on not having to add new rules for each new application domain or language. Knowledge-rich approaches assume that if you grasp the meaning of the text, you can reduce it more effectively, thus yielding a better summary.

Summaries may be extracts or abstracts. Knowledge-poor approaches, at least for the short term, are likely to dominate applications, particularly when augmented with extraction learning mechanisms.

Summarization research is still young and there is consensus on the need for more evaluation [HM00]. Many challenges remain, including the need to scale techniques for generating abstracts.

The transcoding and summarization techniques that have been proposed to date solely concentrate on providing Web access to PDAs and mobile phones.

3.3.1 Quality aware transcoding

Chandra et al. have proposed transcoding techniques to provide differentiated service to Web devices and to dynamically allocate available bandwidth among different device classes, while delivering good quality of information content for all clients [CEV99, CE99, CEV00].

The idea presented is to deliver the information on a Web server according to network connectivity and client device characteristics. The technique proposed concentrates on adapting JPEG images based on bandwidth information. If the connection is weak (i.e., slow), for example, the quality of the JPEG images on the server are reduced to increase the speed of access.

The authors state that in theory, they can use their technique to transcode other multimedia binary objects as well. They say that while they restrict their efforts to the "single metric" (i.e., JPEG images), the techniques are equally valid for any transcoding with well-understood tradeoff characteristics. The solution proposed, however, cannot be used for transcoding text content.

Chandra et al. give some interesting statistics about the percentage of images in Web sites. 77% of the bytes accessed through the Web, they state, belong to multimedia objects. Of these, 67% are transfered for images.

The authors also state that image transformations are important for mobile devices. They provide solutions for a part of the device-independence Web engineering problem: A way to deal with images on mobile devices.

3.3.2 Digestor

Bickmore and Schilit's Digestor [BS97] is a software system that automatically re-authors arbitrary documents from the Web to display appropriately on small screen devices such as PDAs and mobile phones. Bickmore and Schilit's paper on Digestor is one of the first papers in literature that explicitly mentions device-independence.

Digestor is implemented as an HTTP proxy that dynamically re-authors requested Web pages using a heuristic planning algorithm and a set of structural page transformations to achieve "the best looking document" for a display size. HTML pages are analyzed and split into a number of smaller pages that are more easily displayed on PDAs.

WAP and many other Web formats such as XSL:FOP for PDF generation did not exist at the time the system was designed so the tool only concentrates on HTML to simple HTML (e.g., no cascading tables) conversions.

Digestor deals with images by providing a set of techniques that transform all images in the pages by pre-defined scaling factors (25%, 50% and 75%) and making reduced images

hypertext links back to the originals.

The authors state that Digestor does a good job of automatically re-authoring Web pages for display on devices with small screens. They do note, however, that the pages are not always aesthetically pleasing.

3.3.3 Annotation-based Web content transcoding

Hori et al. present an annotation-based Web content transcoding technique in [HKO+00]. They introduce a framework of external annotation, in which existing Web documents are associated with content adaptation hints as separate annotation files. The authors also present a WYSIWYG annotation tool and a transcoding module that they have implemented.

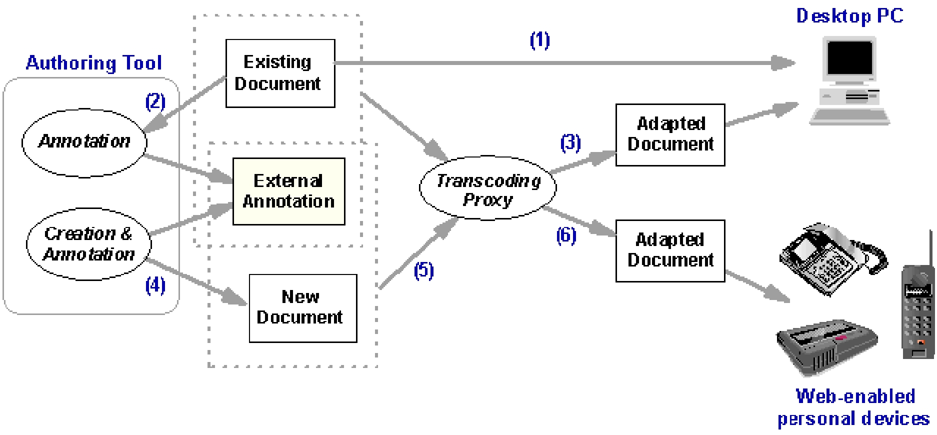


Figure 3.1: Adaptation of HTML for mobile computing devices (Hori et. al [HKO+00])

Figure 3.1 illustrates Hori et al.'s annotation framework for transcoding HTML documents for mobile devices. As the syntax of the annotation files, RDF is used. In addition, W3C XPath and XPointer technologies are used for associating annotated portions of a document with annotating descriptions.

The idea the authors present is quite simple and effective. By using their visual tool, portions of a Web page can be marked (i.e., annotated). For example, a navigation bar can be marked to be displayed on a separate page on a PDA and the page header can be left out for PDA access. This "extraction" information is stored in external files.

Whenever a PDA device accesses the pages, a proxy server converts the pages based on the annotation information. Hence, large amounts of information on an HTML page that do not fit on a mobile device can be split and spread over a number of smaller pages.

The approach does have a major disadvantage, though. Every time the annotated HTML pages change, the annotation definitions need to be updated.

Furthermore, the presented approach only supports PDAs and similar devices with HTML browsers.

3.3.4 The Business Card Search Service (BCSS)

In [KAK+00], Kaasinen et al. describe their experiences in adapting and summarizing existing HTML pages for WAP access.

The authors have implemented a case study service, Business Card Search Service (BCSS), that users can use to search contact information by making queries to a business card database. They have used this application to test how users interact with WML pages that have been converted from HTML.

The HTML/WML conversion proxy server they have developed converts HTML-based Web content automatically and on-line to WML. This approach gives the mobile users transparent access to their familiar Web pages from their mobile phones and other mobile devices.

The study the authors present indicates that if HTML-based Web services follow certain guidelines, they can be converted automatically to WML and adapted to the client device. They state that Web services need to be *mobile-aware* in order to produce acceptable results for users.

The authors report that conversion is not always easy and does not always deliver usable results.

3.3.5 Web access with PDAs: PowerBrowser

In [BGP00], Buyukkokten et al. address the problems of interacting with the Web through wirelessly connected PDAs. As a way to address bandwidth and battery life limitations, they provide local site search facilities for all sites.

They incrementally *index* Web sites in real time as the PDA user visits them. These indexes have narrow scope at first, and improve as the user dwells on the site, or as more users visit the site over time. The authors address the keyword input problem by providing site specific keyword completion, and indications of keyword selectivity within sites.

The PowerBrowser system the authors have built provide two alternative techniques for interacting with the Web through PDAs. These techniques are of two categories: The first supports browsing. The second helps users search more effectively.

The user browses the Web through an HTTP Proxy server. The proxy server fetches Web pages on the PDA's behalf, dynamically generates summary views of Web pages, and manages the site search facility. The connection between the PDA and the Power Browser Proxy Server is established through a wireless modem in the implementation.

The PowerBrowser mainly focuses on easing searching on PDA devices and dealing with input limitations.

3.3.6 Web content and form summarization

In [BGP01] and [KBGP01], the authors present algorithms they have adapted and used for summarizing Web pages and forms so that they can be displayed on handheld devices. They take HTML pages using a proxy, partition (i.e., split) the pages and the user is able to *'mine'* into the partitions.

In [BGP01], Buyukkokten et al. discuss five alternative methods for displaying *Semantic Textual Units (STUs)* to find out how effective each of them are in helping users solve information tasks on PDAs quickly. STUs are page fragments such as paragraphs, lists, or ALT tags that describe images.

The first method, *Incremental* displaying, is the same as the method used in the PowerBrowser discussed in the previous section.

The *All* display method shows the text of an entire STU in a single state. No progressive disclosure is enabled.

The third method, *Keywords*, displays in its first state the "important" keywords that occur in the STU by using a special algorithm.

The *Summary* method consists of only two states. In the first state the STU's 'most significant' sentence is displayed. The second state shows the entire STU. The authors present an algorithm for determining significant sentences.

The *Keyword/Summary* method combines the previous two methods. The first state shows the keywords. The second state shows the STU's most significant sentence. Finally, the third state shows the entire STU.

The authors have conducted experiments with users to find out which technique is most efficient. Keyword summary (i.e., displaying some keywords instead of the whole text) seems to be the most efficient technique.

[KBGP01] is similar to [BGP01], but this time the authors describe algorithms for effectively displaying Web Forms on PDAs.

The approaches mainly concentrate on HTML to simple HTML summarization.

3.4 A taxonomy for device-independent Web engineering

No one to date has attempted to analyze and classify existing approaches that tackle the device-independent Web engineering problem. One reason is probably because different, disjunct research communities (e.g., database, mobile computing and Web engineering people) are working on the problem in parallel.

This section introduces a taxonomy of device-independent Web engineering approaches. Tables 3.1 and 3.2 present the comparison of solutions that tackle the device-independence problem.

The taxonomy is structured as follows: A general section lists the main objective of the approach and the technical features it provides such as static and dynamic content and external database integration support.

The life cycle support section lists the support provided by the approaches for the design, implementation and maintenance phases in the Web service life cycle.

The usability section focuses on the usability aspects of the approach such as its ease of learning and the required developer skills.

The standards section indicates if the standard content and layout definition technologies are used in the approach (e.g., XML or relational databases for content and XSL for layout). Some approaches use proprietary formats for defining the content and layout.

	Approach Name	OOH	WebML	JML	SISL	UIML	iStudio	Cocoon	MS MDT	Total e-Mobile
General Technical Features	Main Objective	To support all Web devices	To support all Web devices	To support all Web devices	To support speech interfaces	To support all User Interfaces	To support all Web devices	To support flexible services	To support mobile devices	To support mobile devices
	Conceptually Platform Independent	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
	External Database Integration	No	Yes	Yes	No	No	No	Yes	Yes	Yes
	Static Content Support	Yes	Yes	Yes	No	Yes	No	Yes	No	Yes
	Dynamic Content Support	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Life Cycle Support	Design Support	Yes	Yes	No	No	No	Yes	No	No	No
	Implementation Support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Maintenance Support	Yes	Yes	Yes	No	No	Yes	Yes	No	No
Usability	Ease of Learning	Low	Medium	High	Medium	Medium	Medium	Medium	High	Medium
	Required Developer Skills	Medium	Medium	High	Low	Medium	Medium	High	Low	Medium
	Service Complexity	Medium	Medium	Medium	Low	Low	Medium	Medium	Low (hidden)	Unknown
	Visual Interface	Yes	Yes	No	No	No	Yes	No	Yes	No

Table 3.1: Comparison of device-independent Web engineering approaches

	Approach Name	OOH	WebML	JML	SISL	UIML	iStudio	Cocoon	MS MDT	Total e-Mobile
Standards	Standard Content Definition (e.g., XML)	No	Yes	Yes	No	Yes	Yes	Yes	No	Yes
	Standard Layout Definition (e.g., XSL)	No	No	No	No	No	No	Yes	No	Yes
Flexibility and Maintainability	Overall Service Maintainability	Medium	Medium	Low	Low	Low	Medium	High	Low	Medium
	Overall Service Flexibility	Medium	Medium	Medium	Low	Low	Medium	High	Low	Medium
	LC Separation	No	Yes	Yes	No	No	No	Yes	No	Yes
	LL Separation	Yes	No	No	Yes	Yes	Yes	Yes	No	Yes
	LCL Separation	No	No	No	No	No	No	Yes	No	No
	Logic Reuse	Yes	No	No	Yes	Yes	Yes	Yes	No	Unknown
Device-Independence Support	XML Web Formats	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
	Device Detection	No	No	No	No	No	No	Yes	Yes	Yes

Table 3.2: Comparison of device-independent Web engineering approaches

The flexibility and maintainability section assesses the flexibility and maintainability of the solution and identifies if there is application logic reuse, Layout/Content (LC), Layout/Logic (LL), or Layout/Content/Logic separation in the solution.

The device-independence support section indicates if the solution is able to support different XML Web formats and if there is device detection support.

3.5 Device-independent Web engineering approaches

Some technologies and tools have been proposed to support the *implementation* of device-independent Web services, but only few have drawn attention to the lack of support for the *design* and *maintenance* phases (e.g., [FKST00]).

Several Web engineering proposals have appeared lately that explicitly tackle the device-independent Web engineering problem.

3.5.1 OO-H Method

The Object-Oriented Hypermedia (OO-H) method is proposed by Gomez et al. in [GCP01]. The authors state that their approach allows Web developers to conceptually model and generate device-independent Web services.

OO-H attempts to provide a standard-based framework to capture all the relevant properties involved in the modeling and implementation of Web application interfaces. The methodology contains two views: the navigation view extends a class diagram with hypermedia navigation features and the presentation view uses the different elements regarding the interface appearance and behavior to model a number of interconnected template structures expressed in XML.

The navigational views are defined in so called Navigational Access Diagrams (NADs) and presentation views are defined in so called Abstract Presentation Diagrams (APDs). Both NADs and APDs capture the interface-related design information with the aid of a set of patterns, defined in an interface pattern catalog integrated in the OO-H method proposal.

A model compiler in the framework generates the Internet application front-end for the desired client platform and/or language (e.g., HTML, XML, WML). The authors state that they have developed a CASE tool that automates the development of Web applications modeled with the OO-H method.

Each NAD instance reflects the information, services and required navigation paths for the associated user's navigation requirements fulfillment. Figure 3.2 illustrates the OO-H design process.

The authors have adapted a template approach for the specification of the visual appearance and page structure (i.e., APDs) on the Web. The framework contains five types of templates: *tStruct*, *tStyle*, *tForm*, *tFunction* and *tWindow*.

tStruct instances define the information that has to appear on the abstract page. *tStyle* instances define features such as physical placement of elements, typography or color palette. *tForm* instances define the data items required from the user to interact with the system.

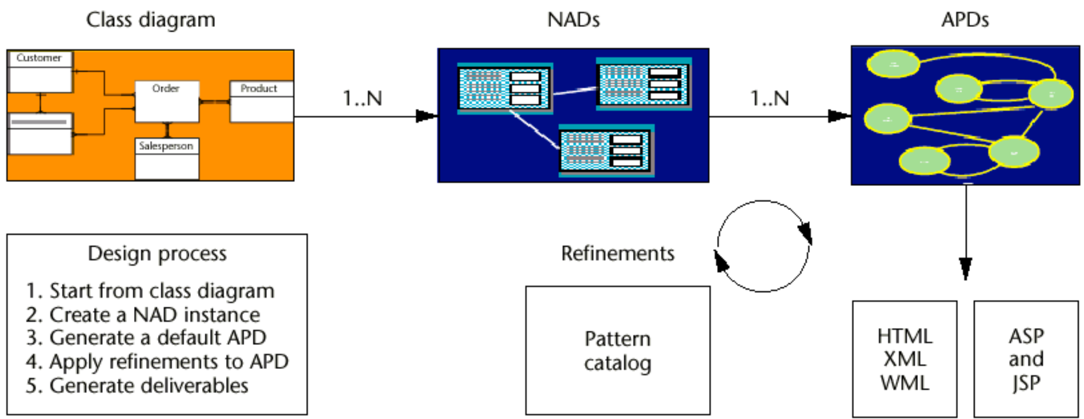


Figure 3.2: OO-H Design Process (Gomez et al. [GCP01])

tFunction instances capture client functionality and *tWindow* instances define a set of simultaneous views available to the user.

The framework allows the Web developer to choose patterns and to instantiate and use them in the application being constructed. These patterns can be instantiated by using commands such as:

```
Dlist->addAPDPage(h); h.name'head'
```

The command above, for example, inserts a header template into a page.

In their paper, the authors present a case study HTML Web site that can be used to manage discussion lists. The user first sees a list of discussion topics and by clicking on the link, she sees the list of messages in that discussion group and is able post replies and messages to it. The authors state that they have developed the sample application using JavaServer pages and Java Bean components as the server-side and HTML as the client-side technology.

The authors say that by invoking the model compiler, they are able to generate user interfaces for different devices and present the screenshot of a page as seen on a WAP device to illustrate the device-independence of the approach. The paper does not give any details about the model compiler.

A single page in the example site they provide contains a list of discussion topics and the message overview pages list all the messages in a discussion group. To support WAP devices, the model compiler they describe takes the page specifications and generates a WML version of the functionality.

One problem is that a single page that contains too much information such as a high number of messages in a discussion group may cause errors on devices with memory limitations. The approach of mapping a single HTML page to another device does not always give satisfactory results.

The OO-H method is a promising new approach that specifically tackles the device-independence problem.

3.5.2 WebML

The Web Modeling Language (WebML) [CFP99, CFB00] is a high level modeling and specification language for Web sites. The language was developed in an EU project and it is completely XML-based. WebML is an evolution of AutoWeb [FP00] developed by the same research group.

WebML enables designers to express the core features of a site at a high level without committing to architectural details. A CASE tool is provided that can be used to create XML specifications that are then used to automatically generate server-side scripts.

The system has a *structure* and *hypertext* model. The hypertext model consists of *Composition*, *Navigation*, *Presentation* and *Personalization* models.

The fundamental elements of WebML structure model are entities that are containers of data elements (i.e., *data units*), and relationships, which enable the semantic connection of entities. Figure 3.3, for example, depicts the graphical notation for data units and a possible rendition in HTML. The data unit displays the contents of the *Artist* entity.

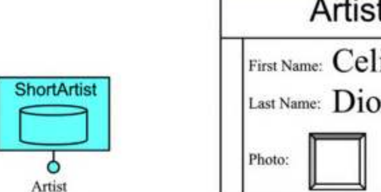


Figure 3.3: WebML graphic notation for data units, and a possible rendition in HTML (Ceri et al. [CFB00])

WebML uses the notion of pages that can be used to compose the content in data units. The layout information is defined using a special, tool-supported XML syntax.

In their paper (i.e., [CFB00]), the authors state that they are able to generate layout code of their choice such as WML for WAP devices and HTML for traditional browsers: They say that WebML can be used to support "multi-access" Web sites.

The examples they provide in the paper show a traditional, HTML-based service being constructed. As future work, they state that they are working on WML extensions to the language.

Because large database query results cannot always be displayed on some devices in practice because of memory limitations, in their project Web site [web01], the authors describe the problem and indicate that they have done some extensions to the tool that allows the database query results to be automatically split for WAP devices.

WebML does not provide any support for dynamic content and only deals with static content that is stored in relational databases.

3.5.3 JML

In [BS00a], Barta and Schranz describe the Jessica Markup Language (JML). JML attacks the multi-purpose publishing problem so that Web content can be generated for various target platforms such as XML, WML and HTML.

JML provides object-oriented support to abstractly describe information for the Web. The authors state that the approach includes typical OO benefits such as encapsulation, reusability, and inheritance. The most basic components of JML are pages and layouts. JML is an XML-based language. It is quite similar to the Jessica system (e.g., see [BS98]), but has some extensions that allows it to support formats other than HTML.

JML solely aims to separate the layout and the content for static content and does not deal with dynamic content.

3.5.4 SISL

Several Interfaces Single Logic (SISL) [BCD+00, GJJL00] is a system that has been designed and developed by Lucent technologies.

The idea in SISL is to use reactive constraint graphs to model the service logic. The authors have developed an XML language for writing special SISL programs. They claim that programs written in SISL are device-independent. They have constructed a service that reuses the application logic and supports voice interfaces in VoiceXML and an HTML interface.

A special *service monitor* takes care of the interaction between the user interfaces and service logic. User interfaces can be developed in the language of choice and the service monitor runs it.

SISL separates the logic from the layout but does not attempt to separate the content from the application logic. A static text such as “Welcome to this site”, for example, is embedded into the source code.

The authors state that they provide mechanisms to customize the user interface. The main HTML interface forms depicted in [BCD+00] are generated automatically.

The developers of SISL say that they plan to use SISL for PDAs and mobile devices.

The papers on SISL give a detailed analysis of the problems related to speech/ voice interfaces. The tool does not attempt to cover the design and maintenance phases in the Web service life cycle.

3.5.5 UIML

The User Interface Markup Language (UIML) [APBW99, AP99, Abr00, Lin01] seeks to create one canonical syntax that can be used to specify user interfaces. Using this syntax, the user interface definition becomes platform and language independent. By using specialized model compilers, a common user interface description can be converted to WML, HTML, VoiceXML, Java Swing, etc.

In UIML, a user interface is a set of user interface elements with which the user interacts. Each interface element has data (e.g., text, sound) used to communicate information to the user. Runtime interaction is done using events. Events can be local (i.e., between user interface elements) or global (i.e., between interface elements and objects that represent an application's internal program logic).

UIML is truly device-independent and has model compilers for WML, VoiceXML, HTML and Java and it has been shown to work in example sites.

UIML treats the Web as just another user interface for the application logic. Web applications, however, are more than user interfaces because they also have hypertext characteristics such as embedded links and a significant content maintenance overhead. UIML is a language that can be effectively used to describe user interfaces and components.

One disadvantage of the approach is that although it separates the layout from the logic, the content is often embedded into the UIML specifications. Although UIML has a *template* mechanism to group common user interface elements, supporting a common Web look-and-feel and multi-lingual Web sites is still not easy because the content is intermixed with the user interface definition.

The authors state that they attempt to separate the content from the user interface but the content they refer to is content describing user interface elements (e.g., text on buttons).

Similar to SISL, UIML does not attempt to cover the design and maintenance stages in the Web service life cycle.

3.5.6 iStudio

iStudio [SHKE01] is an application development environment based on the Java, XML and XSL technologies. The developers of iStudio state that the tool can be used to build device-independent Web services. The application logic can be reused to support different devices such as VoiceXML browsers, PC browsers and WAP devices.

```
<is:fragment name="body">
<form method="post" action="validateUser">
  <is:attr name="action">
    <is:link objAlias="TransferTable" clearParams="true">
      <is:param name="action"><is:content/></is:param>
    </is:link>
  </is:attr>
  <table>
    <tr><td> Cellular # (10 Digits): </td>
      <td><input type="text" name="userID" size="10" value="">
         <is:attr name="value"><is:temp name="userID"/></is:attr>
      </input></td></tr>
  </table>
  <p><input type="submit" name="submit" value="SUBMIT"/></p>
</form>
</is:fragment>
```

Figure 3.4: A sample iStudio fragment that defines an XHTML form (Skarra et al. [SHKE01])

The main objective of iStudio is to support the service creation and the reduce development time through the definition of reusable and extensible application components. The developer uses XML in the approach to specify a service (i.e., its business logic, data presentation, authentication and permissions, configuration). A suite of iStudio tools transform the specifications into a collection of Web-capable objects that implement the service.

A run-time engine in the system interprets service code and responds to client requests.

iStudio is conceptually similar to the W3Objects and Webcomposition approaches discussed in Section 3.2.5. Instead of intermixing the layout and logic as Webcomposition and W3Objects do, though, iStudio uses a concept the authors denote *fragments* to separate the layout and to map elements in the layout to the application logic. The fragments are comparable to components and objects in the Webcomposition and W3Objects systems.

To support different devices, fragments have to be written that produce the appropriate code (e.g., WML for WAP, HTML for PCs). One problem with these fragments is that fragments containing content may need to be duplicated for different devices. Figure 3.4 depicts a typical fragment that defines an XHTML form. Much of the embedded content in the fragment, for example, would have to be duplicated for WML and other devices. This would have a negative effect on maintainability.

3.5.7 Cocoon

Cocoon is a Java servlet-based application server that is based on freely available XML parses (e.g., Xerces [Apa01b]) and XSL processors (e.g., Xalan [Apa01a]). Cocoon can be used for the real-time translation of XML files on a web server to HTML and any XML-based Web format such as WML.

Cocoon is designed to allow Developers, Business Analysts, Designers, and Administrators to work with each other in parallel without breaking the other person's contribution.

The Cocoon community believes that the problem with using technologies such as ASPs [RAS00] or ColdFusion [col] templates is that " all of the the look, feel, and logic are intermixed." Maintenance, hence, is often much more difficult, costs more and takes longer. If the site layout design is introduced late in the design phase, for example, the cost of integrating the graphical look may become significantly higher. Cocoon aims to separate concerns and to enable the involved parties to work in parallel as much as possible.

The Cocoon project proposes two technologies for providing flexible and layout independent dynamic content in web pages; XSP (eXtensible Server Pages) and DCP (Dynamic Content Processor). XSP is completely based on XML/XSL technology and uses XSL tag libraries and associated code generation style sheets (logic sheets) to generate compilable source code. DCP uses a simpler approach than XSP but is an interpreted language and thus, has a performance drawback. DCP is only intended to support dynamic content.

Cocoon supplies a number of different components for the Web developer. The types of components are *Generators*, *Transformers*, *Serializers*, *Readers*, and *Actions*.

A Generator will create SAX[2] events for a SAX stream. A *FileGenerator*, for example,

[2]SAX is the Simple API for XML, originally a Java-only API. SAX was the first widely adopted API for XML in Java, and is a de facto standard. The current version is SAX 2.0.1, and there are versions for several programming language environments other than Java.

reads an XML file from an input source, and converts it into a SAX stream.

Transformers read a SAX stream, manipulate the XML stream, and send the results to the next component in the chain. The provided *LDAPGenerator*, for example, is a class that can be plugged into a pipeline to transform the SAX events that passes through it into queries and responses to and from an LDAP interface.

Actions are the main form of logic processing in Cocoon. There are a number of approaches that can be taken when developing Actions. One possibility is to create a specific action for each piece of application logic. This approach is heavy handed and requires much development time to create actions.

The preferred method for creating actions in Cocoon is to provide a generic action that can handle a wide range of specific actions. The Database Actions and Validator Actions are examples of this approach. They will read a configuration file specified by a parameter, and they will modify the specific results based on the configuration file.

Serializers read a SAX stream and convert it into the servlet's output stream. Readers read an input stream and copy the results to the servlet's output stream.

Cocoon also provides functionality for querying, updating and embedding content stored in relational SQL databases.

```
<p>
      Name: <text name="name" size="30" required="true"/><br/>
      <xsp:logic>
      if (<xsl-formval:is-toosmall name="name"/>)
            <xsp:text>"Name" must be at least 5 characters</xsp:text>
      } else if (<xsp-formval:is-toolarge name="name"/> {
            <xsp:text>"Name" was too long</xsp:text>
      }
      </xsp:logic>
</p>
```

Figure 3.5: Part of a logic sheet in Cocoon

Figure 3.5 shows part of a logic sheet in the Cocoon system. Although Cocoon aims to separate the layout, logic and content, these are still intermixed to a certain degree.

An interesting feature of Cocoon is its ability to automatically detect devices based on the HTTP request header information. The system can be configured to detect devices and to invoke the corresponding stylesheets.

Cocoon is only an implementation technology and does not provide any direct support for the design and the maintenance phases of Web services.

3.5.8 Microsoft ASP.NET and the Mobile Developer Toolkit

Microsoft's new ASP.NET framework [dev] has extensive support for the creation of Web pages and Web services. The Visual Studio graphical development environment enables Web developers to rapidly create Web pages, Web sites and Web services. For example, Microsoft's C# has been designed to make it easy to export C# methods as Web services.

The way ASP.NET deals with Web services is quite low-level: The framework lacks a higher-level, language-independent model for dealing with device-independent Web services.

Recently, Microsoft has started shipping the Mobile Developer Toolkit that is an extension to the Visual Studio Development Environment. This toolkit provides a visual environment for creating and deploying Web services for mobile devices.

The developer creates an application by placing components such as *buttons* and *text fields* into forms. Content is also inserted into these forms in terms of *label* components. Based on the characteristics of a device (e.g., PDA, WAP phone, etc.), the platform automatically adapts and renders the forms to be viewable on the device.

The main advantage of this development platform is that applications can be rapidly developed without a high technical knowledge. The disadvantage is that the created applications are not flexible and rather difficult to maintain because of the use of forms (e.g., changing a logo on each page could mean that the developer has to manually delete each logo component on every form).

Similar to Cocoon, the Mobile Developer Toolkit provides an implementation platform and technology, but does not aim to support the design or maintenance stages of Web services.

3.5.9 Total e-mobile

There are several commercial systems that claim to enable the construction of device-independent Web services. Bluestone's Total-e-Mobile business solution [blu02], for example, is one such system and is a good representative.

Unfortunately, it is not always possible to find out technical details about these commercial products and to test how good they work.

Bluestone says that its solution is device-independent and that "regardless of the device being used, Total-e-mobile can serve up correctly formatted, fully functional content from a single URL. It does this by automatically sensing the client device and using HP Bluestone's Dynamic-Stylesheet-Engine (DSE) to format content appropriately for any known device whether it is a browser, a cell phone or a vending machine" [tot01].

The product uses XSL to define layout information for new services and can support various mobile devices. It uses conversion techniques for existing HTML pages. Devices are defined and identified by cookies that the clients store.

Technologies such as Bluestone, though, usually do not provide any support for the design phase of Web applications. In their white papers, for example, Bluestone state that XSL stylesheets can be simply used to define layouts for different devices. They do not explain, however, how the developer can *design* the stylesheets and the service to minimize redundancy and maximize reuse.

3.6 Summary

This chapter presented related work. It described and discussed traditional Web engineering approaches and mobile Web access techniques that *do not explicitly* attack the device-independent Web engineering problem, but that are relevant and important as background work. It then introduced a taxonomy for classifying and comparing the solutions that *explicitly* tackle the device-independent Web engineering problem and described and evaluated these approaches.

Chapter 4

DIWE: A conceptual framework for device-independent Web engineering

This chapter introduces a novel conceptual framework for device-independent Web engineering. The Device-Independent Web Engineering (DIWE) framework is composed of an XML-based Web language that is used to separate the layout, content and application logic to construct flexible Web services and four default run-time processors that provide device-independence support during service execution.

The framework introduces and uses two novel techniques, *page splitting* and *process partitioning* by layout marking, that allow the Web developer to tune the selected information and the sizes of generated pages according to the characteristics of a device that is being targeted. These techniques attack the problem of displaying Web pages on devices with small displays and memory sizes. The adaptation of content for a Web device is performed during the design and implementation stages of Web site engineering. During the implementation, the Web engineer has full control over the partitioning and selection of information.

The framework also introduces a novel technique called *XSL stylesheet pre-processing* that allows the reuse of *existing* XSL stylesheets when adding new devices to a Web service. The approach, advocated by the W3C, of having a new XSL stylesheet for every supported device does not work effectively and there is often quite a lot of duplication in the stylesheets. As a consequence, the maintenance overhead increases. Stylesheet pre-processing significantly reduces the maintenance overhead because a lesser number of XSL stylesheets are needed.

The chapter is structured as follows: First, the Web service life cycle discussed in Chapter 2 is revisited and device-independence considerations are integrated. Second, an overview of the conceptual framework is given. Third, the concepts of page splitting, process partitioning and XSL stylesheet pre-processing are presented and discussed.

4.1 Rethinking the Web Service Life Cycle

Adaptability is an important issue when building software of any sort [GJM91]. Requirements change between the time when the customers say what they want and the time when the software is actually delivered. Fayad states in [FC96] that software that is being built

must be adaptable with respect to the ability to change the system's capabilities in amount and in kind, and the ability to fix the system without "breaking" other parts.

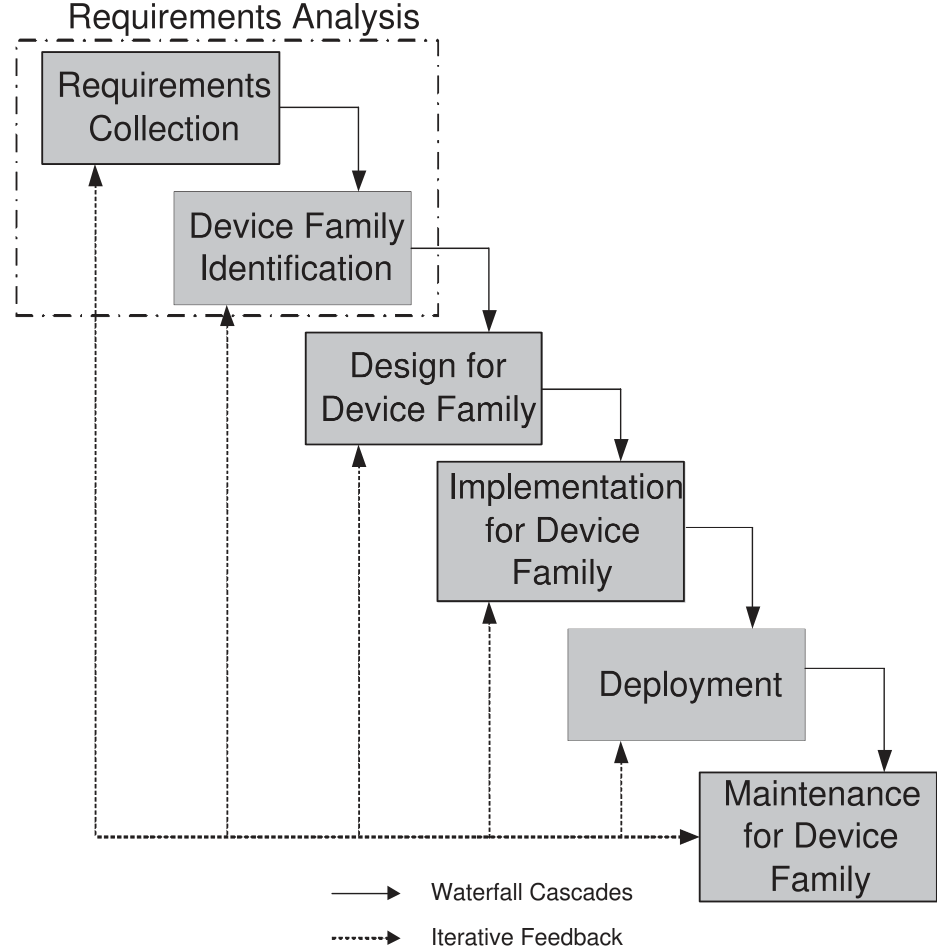


Figure 4.1: Life Cycle of a device-independent Web Service

As motivated in the previous chapters, one more adaptability requirement must be included for Web sites: *device-independence*.

Figure 4.1 depicts the WWW Service Life Cycle model with the integrated device-related processes. The requirements analysis includes traditional steps such as identifying the target audience, the functionality goals of the service and quality parameters. *Devices families* need to be identified that the service will support. A device family is made up of a collection of Web devices that have similar characteristics. PDAs with high memory capacity and a display size larger than 200x300 pixel size, for example, could make up a device family for a particular service. Another example of a device family is the collection of WAP-enabled phones and PDAs.

Each Web service will at least support one device family during its life time: the *default* device family. For example, a typical decision in a cultural event Web site could be to support a full HTML browser interface for the entire site as the default device family and a WAP-based mobile phone interface for the ticketing service only.

The main difference in the design, implementation and maintenance stages in comparison to the traditional Web service life cycle model is that these phases differ individually according to the device families that are to be supported. The WAP design, for example, will show differences to the HTML design: The navigation will be different and due to the memory limitations of mobile phones, the amount of information per page that can be displayed will also differ. The maintenance overhead is clearly higher than in traditional, HTML-only Web services because of the higher number of formats and devices that need to be supported. At the same time, changes may occur that *only* affect one device family and have no effect on the others. For example, changing the HTML layout to give the site a more appealing look-and-feel will not affect the WAP pages.

One important difference in the model is the introduction of a *deployment* phase. The deployment phase is ignored by well-known Web service life cycle models (e.g. [Sch98b, TL97]). Deployment is especially important when more than one Web device has to be supported and requires a significant planning, coordination and configuration effort.

As requirements change and new devices have to be supported, the Web engineer will often go back to the device family identification stage in the requirements analysis phase and iteratively design and implement support for a new device family.

The XML/XSL-based solution proposed in this dissertation is a flexible approach that eases the implementation stage and attempts to reduce the overall design, maintenance and deployment effort in engineering device-independent Web sites by reusing stylesheets and application logic.

4.2 Basis of solution: Separation of Layout, Content and Logic (LCL)

The basis of a solution to the device-independence Web site engineering problem is to find a way to effectively separate the layout (i.e., user interface) from the application logic.

The idea of separating the user interface from the application logic for achieving flexibility is not new and well-known (e.g., [Coc96]). User interfaces in software systems change frequently. Keeping the user interface "outside" the system and making the system program-driven has been a discussion issue in software engineering for many years. This separation is not always easy to achieve in traditional, large and complex software systems. The question which modules belong to the user interface and which do not cannot always be answered with ease. For example, should keyboard inputs be handled by the user interface component, or are they part of the application logic?

Because Web services are *event-driven*, it is easier to separate the layout from the application logic. A Web service reacts to user input by returning HTML that is then displayed on the user's browser so the interaction of the user with the service is session oriented. Every time the user gives some sort of input to the system, a connection to the service is built from the user's browser.

This layout and logic separation by itself, however, is not enough to enable the engineering of truly device-independent Web sites. An interfacing mechanism is needed for interactions that allows layouts of *varying* sizes to be supported with the same application logic.

Furthermore, the content needs to be separated from the layout and the logic as well to increase maintainability and flexibility. Clearly, a full *Layout/Content/Logic (LCL) separation* is needed for achieving flexible, maintainable device-independent Web sites.

Although the XML and XSL technologies solve the layout and content separation problem, they do not address application logic separation in Web sites. The concepts presented in this chapter fill this gap.

The next section discusses and summarizes the main requirements for a conceptual framework that supports device-independent Web site engineering.

4.3 Main requirements for a device-independent Web engineering framework

There are four important requirements that a device-independent Web engineering framework should meet: It should use *industrial standards* to enable the use of existing tools, it should be *platform and implementation language-independent*, it should support the *definition and generation of content and layout in XML* for non-HTML Web devices and most importantly, it *should not increase the maintenance effort significantly*.

The design of the DIWE framework presented in this chapter was guided by the following goals:

- *Support should be provided for the design, implementation, deployment and maintenance phases of a device-independent Web site.*
- *The XML and XSL standards should be used as core underlying technologies.* Many Web developers are already familiar with XML and XSL and there is wide third party tool support.
- *Both static and dynamic content should be supported.* The framework should enable the construction of interactive Web sites as well as Web pages that are static in nature.
- *The integration of content in Relational Database Management Systems (RDBMSs) should be supported.* RDBMSs are widely used in Web sites to store and manage content. Providing RDBMS support is essential.
- *Adaptation of the application logic in the Web site for a new device should not be necessary.* The same logic needs to work for any device (independent of its display and memory size) so that application logic maintenance is eased.
- *Layout adaptation should be possible.* A page that can be displayed without problems on a device with a large display may be too large for other devices and needs to be split. Support should be provided for splitting pages.
- *The use of stylesheets and separation of Layout, Content and Logic (LCL) should not increase the maintenance effort significantly.* A typical consequence of LCL separation may be that the number of project files and resources increase.

4.4 Overview of the DIWE framework

The DIWE framework consists of the *MyXML language*, a compiler that can interpret the language, and four basic run-time *processors* that are configured and deployed on the Web server at run-time to provide device-independence support. These processors are Web services themselves.

A device-independent Web service in the DIWE framework is a Web service that can be *extended* to support different Web devices of widely varying technical capabilities. The first step in constructing a *device-independent* Web service, hence, is to construct a *flexible* Web service with the MyXML language. The language provides support for LCL separation and is used by the Web developers to design and define the content and the interfaces to the application logic. A MyXML language compiler integrates the layout and generates static content embedded in HTML or XML, or source code that provides interactive functionality.

A Web device that accesses the Web server interacts with the instance of the run-time processors that filter and adapt the output produced by the MyXML-generated Web services. If no layout adaptation is required, the device may also be configured to directly access a Web service.

4.4.1 Web service design, implementation, deployment and maintenance

Figure 4.2 illustrates the usage of the framework in the design, implementation, deployment and maintenance stages of Web services.

During the design stage of a Web service, the content, layout and the application logic are designed and defined. The application logic is written using a technology of choice such as Java servlets. The MyXML language is used to define the content and the interfaces to the application logic and the layout is defined using XSL stylesheets.

To benefit from the advantages of XSL, the developer needs to follow traditional XSL-based Web engineering guidelines such as analyzing the commonalities of the pages and not encoding any content into the stylesheets to enable reuse (e.g.,[KKJK01]).

Content definition covers the structuring of information to be displayed in the Web service so that it can be adapted according to the characteristics of different device families. The content needs to be designed carefully to make it accessible from heterogeneous devices. Having the content in XML does not necessarily guarantee that it will be automatically accessible by all devices. The *description granularity*, the degree the content is described in XML, has to have the correct depth. If the description granularity is not deep enough, it will not be possible for some devices to select it using XSL. More descriptive tags will have to be inserted into the XML content later and the maintenance overhead will increase.

Figure 4.3 illustrates the description granularity problem. The content definition on the left has a lower description granularity than the definition on the right. Suppose only one sentence per page can be displayed because the Web device is a watch with a mini browser (e.g., a device such as IBM's Linux watch [NKR+02]). The content definition on the left would cause problems because the *entire* content is wrapped up in a single <text> tag. The definition on the left, in comparison, marks each sentence using extra <sentence> tags and

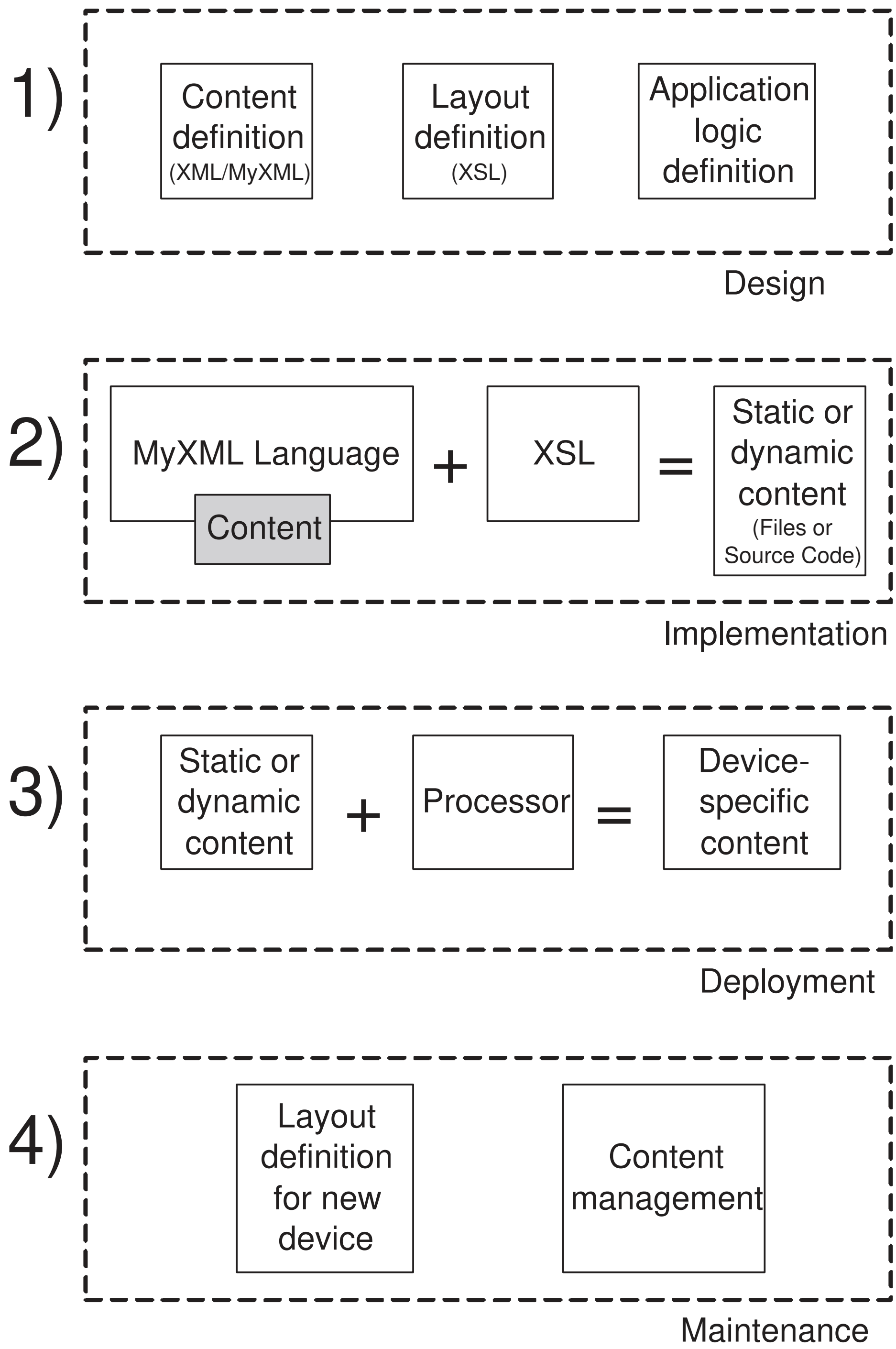


Figure 4.2: Web service design, implementation, deployment and maintenance

```
<text>
This is a long example text:

To be or not to be. To be or
not to be. To be or not to be.
To be or not to be.
be or not to be. To be or not
to be. To be or not to be.

To be or not to be. To be or
not to be.
</text>
```

```
<text>

<sentence>
This is a long example text:
</sentence>

<sentence>
To be or not to be. To be or not
to be. To be or not to be. To be
or not to be.
</sentence>

<sentence>
be or not to be. To be or not to
be. To be or not to be.
</sentence>

<sentence>
To be or not to be. To be or not
to be.
</sentence>

</text>
```

Figure 4.3: Differences in description granularity

thus, allows the selection of sentences one by one. The more descriptive the content is, the better it is for device-independent access.

During the implementation stage, a compiler that interprets the MyXML language is used to process the content and add an XSL layout to it. The resulting dynamic or static content is processed by the run-time processors during the deployment phase and device-specific content is generated.

During the maintenance phase, the XSL layout is extended for new devices (e.g., using new XSL stylesheets or XSL stylesheet pre-processing) and the XML content is maintained.

4.4.2 Processors

The four default run-time processors in the DIWE framework are: The *device detection*, *logic interfacing*, *page splitting* and *process partitioning* processors. These processors are instantiated and used at run-time in combination with the static and dynamic Web services defined by the MyXML language and generated with a MyXML language compiler. The Web developer can optionally construct and deploy application-specific processors that can process the content produced by the MyXML-generated Web services (e.g., to generate PDF receipts for an e-commerce order).

The *device detection* processor is responsible for device detection and identification. It can be configured to detect the device a user is using based on the HTTP request header and respond accordingly.

The *logic interfacing* processor provides device-independent application logic interfacing support to the services specified by the MyXML language. It allows the application logic

to be written once and used for multiple device-specific MyXML-generated Web services without any modifications.

The *page splitting* and *process partitioning* processors provide *layout adaptation* support. Layout adaptation in Web site construction deals with the problem of displaying pages on device families with small display and memory sizes. It also deals with the problem of providing Web form-based interaction support to users on devices with limited capabilities. An e-commerce application, for example, may collect information from the user such as her name, address and credit card number in a *single* HTML page. This information may be too large for a weak device such as a WAP phone.

The *page splitting* technique deals with the page size problem by using a combination of special tags that are encoded into the XSL stylesheets that are interpreted by the page splitting processor at run-time. Based on this "splitting" information, the content of a single page can be incrementally displayed on the target device family over a number of steps.

The process partitioning processor applies the *process partitioning* technique to deal with Web form-based input and interactions on devices with small displays and limited memory. It collects the required input from the user partially over many smaller pages. The application logic is invoked once all the information has been submitted. The process partitioning technique uses the page splitting technique to adapt the layout to the device.

4.5 Flexible Web service construction in three steps

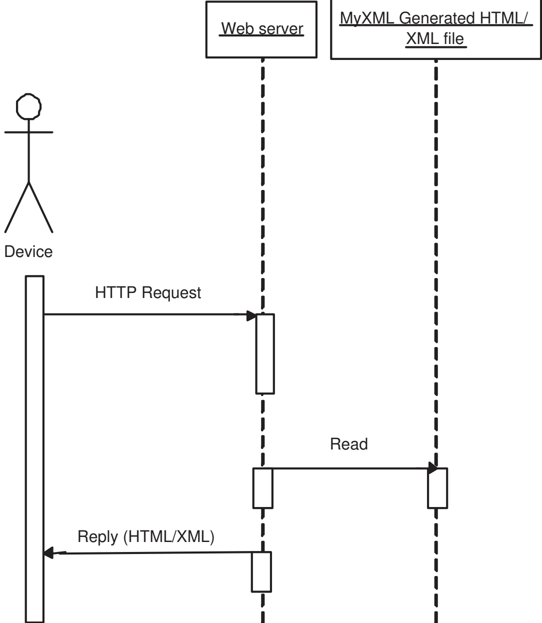


Figure 4.4: Interactions between the user's device, the Web server and the generated static content

The first step in creating a flexible Web service that is extensible and supports LCL separation is to define the content in a so called *MyXML document*. These documents are

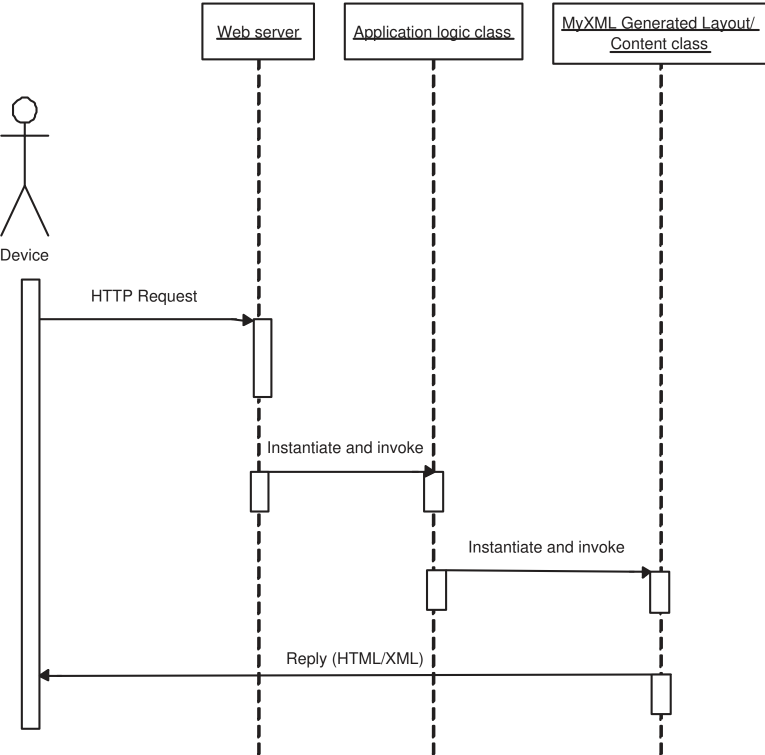


Figure 4.5: Interactions between the user's device, the Web server, the application logic and the generated functionality that produces the dynamic content

well-formed XML documents that contain the structured content and can also be based on a document type definition (DTD) that defines the content's overall structure. MyXML documents consist of XML content enriched with XML tags from the *MyXML namespace*. The MyXML namespace defines the elements in the MyXML language. The language enables the developer to add database integration functionality and dynamic content to a Web service.

In the second step, all the necessary layout information is added to the content defined in MyXML documents as separate XSL documents. Context information can be used in the layout definition rules and enables the processing of elements only if they appear in a predefined context (e.g., if they have a certain parent element, if they have an attribute with a given value etc.). XSL stylesheets can also be used to add static content, such as common headers and footers to the documents.

If the service being constructed produces static content, a MyXML language compiler is used to process the MyXML document and the XSL layout definition and generate an HTML or XML file. The generated files are then deployed on the Web server. Figure 4.4 shows a sequence diagram describing the interactions between the user's device, the Web server and the generated static content.

If the service is dynamic, source code that encapsulates the content and layout information is generated. The reference implementation produces Java sources and this source code provides *hooks* that the application logic can use to instantiate and invoke it. The Web developer then provides the application logic in the third step and uses the generated sources

to produce the dynamic content at run-time. Figure 4.5 shows a sequence diagram describing the interactions between the user's device, the Web server, the application logic and the generated functionality that produces the dynamic content.

4.6 Device-independent Web service construction in three steps

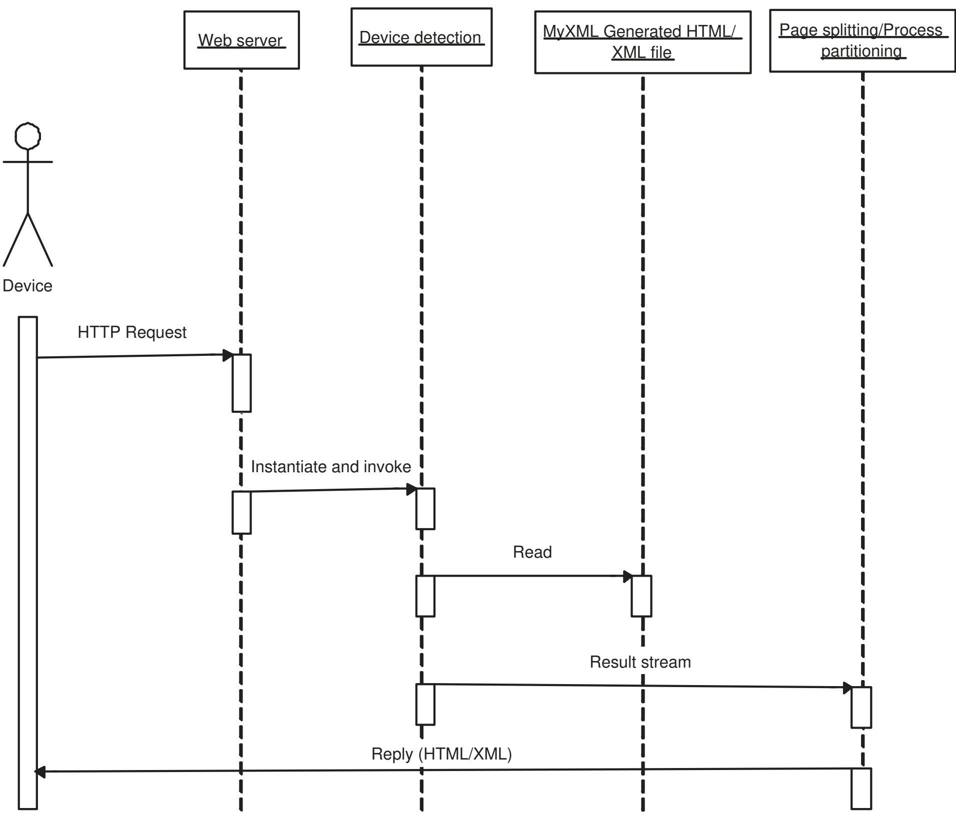


Figure 4.6: Sequence diagram showing the interactions between the device-independence components for static content

The first step in creating a device-independent Web service is to create a flexible Web service as described in the previous section. The Web service can later be extended to support multiple layouts with the same content by either using different XSL stylesheets or XSL stylesheet pre-processing. Page splitting and process partitioning information is embedded into the stylesheets during service definition.

In the second step, the *device detection* processor is configured and deployed on the Web server. At run-time, based on the request and the device the user is using, the *device detection* processor responds by dispatching the HTTP request to the corresponding device-specific pages that have been prepared by using the MyXML language.

If the service the user is accessing is static, the *device detection* processor reads the MyXML-generated HTML/XML file and passes the result stream to the *page splitting* and

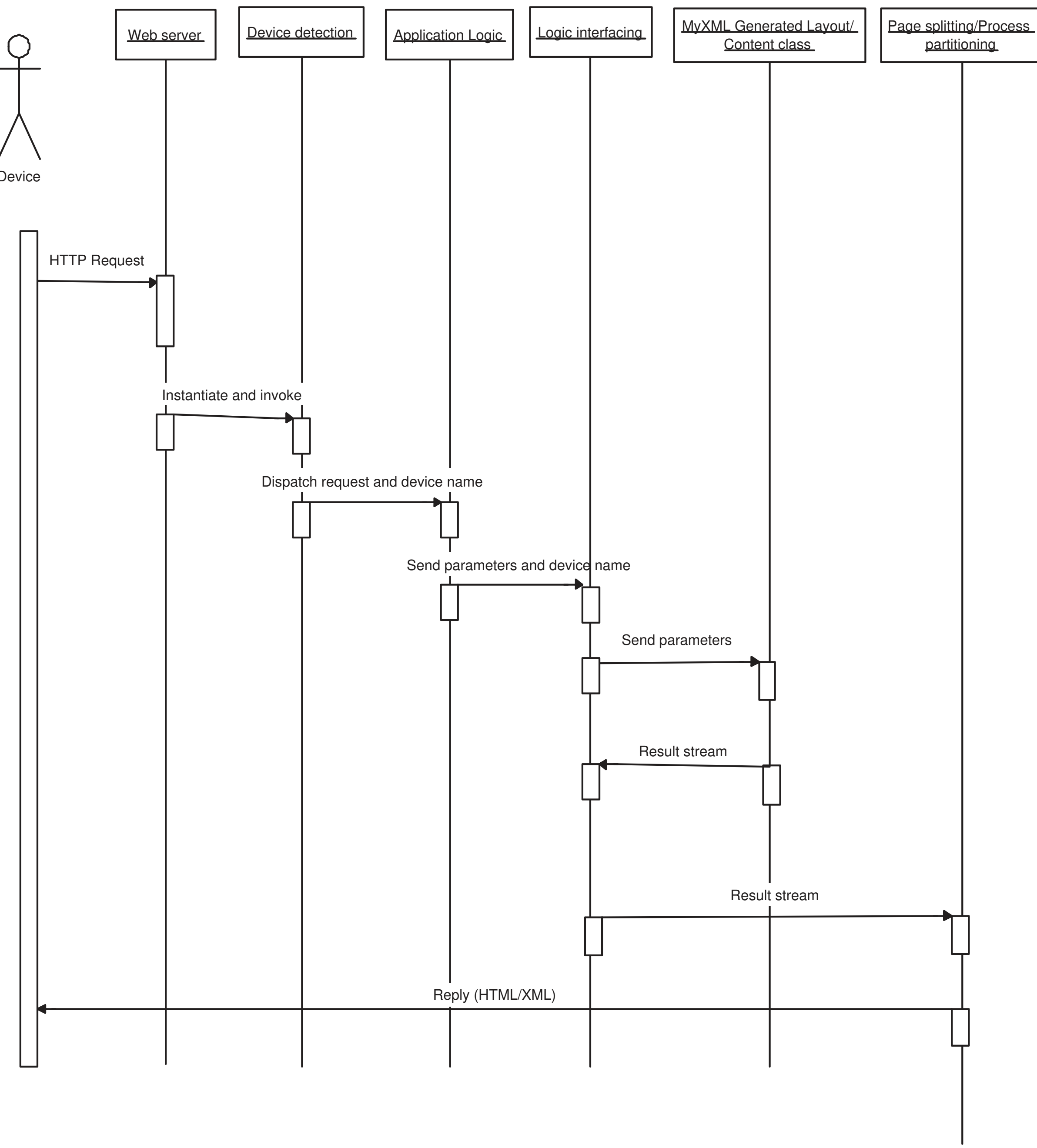


Figure 4.7: Sequence diagram showing the interactions between the device-independence components for dynamic content

process partitioning processors. These processors process the result stream and split the pages and interactions accordingly. They return HTML/XML to the requesting client device. Figure 4.6 shows the sequence diagram that illustrates the interactions between the default processors in the framework for processing static content.

The third step is only needed if the service being constructed is dynamic. In this step, the *logic interfacing* processor is configured and deployed on the Web server. The *logic inter-*

facing processor transparently and automatically invokes the corresponding layout/content classes based on the name of a device family.

Figure 4.7 shows a sequence diagram that illustrates the interactions between the default processors in the framework at run-time for processing dynamic content. The *device detection* processor receives the HTTP request and invokes the application logic. The application logic instantiates and invokes the *logic interfacing* processor with the device name it has received from the *device detection* processor and parameters it would like to pass to the layout/content class. The *logic interfacing* processor then instantiates the corresponding layout/content class and invokes it. It passes the result stream returned from the layout/content class to the *page splitting* and *process partitioning* processors. These processors process the page splitting and process partitioning information in the stream and return HTML/XML to the calling client device.

4.7 The MyXML language

The MyXML language used in MyXML documents is a simple XML-based language that uses loops, variables and database access functions.

One of main advantages of an XML-based Web language is that it allows the definition of functionality that is platform and technology independent. Although the reference implementation is based on the generation of Java sources, any popular programming or scripting language can be generated from the MyXML documents and XSL specifications with an appropriate MyXML language compiler.

4.7.1 Overview

Each element in the MyXML language has a special meaning and is processed accordingly by the MyXML language compiler. Variable definitions are the most important elements in the MyXML namespace because they define the interface between the application logic and the generated sources containing the layout and the content.

There are two types of variables in the language: *Singles* and *Multiples*. *Single* variables map to String objects in Java (i.e., character arrays in C) and *Multiple* variables map to arrays of String objects (i.e., n dimensional character arrays in C).

A *Loop* in the MyXML language defines a block of content and MyXML elements that are iterated according to the number of elements in the *Multiple* variable in the *Loop*. Each *Loop* has to have at least one *Multiple* variable in it and *Multiples* cannot exist without an encapsulating *Loop* block as its parent. A *Loop* block, for example, that contains a *Multiple* variable **names** will be processed by the MyXML language compiler to produce Java source code (i.e., in a Java implementation) that looks like the following (pseudo code):

```
for(i=0;i<=names.length;i++) {
        ... DO SOMETHING ...;
        <print out> names[i] ...;
        ... DO SOMETHING ...;
}
```

The *Loop*, *Single* and *Multiple* statements are all a Web developer needs to successfully separate the application logic from the content and the layout. The MyXML language, however, also provides CGI and database functionality that eases the construction of interactive, database-backed Web services.

4.7.2 MyXML Namespace

The MyXML namespace describes 18 elements that belong to the MyXML language. The language has 8 core elements that are needed for constructing flexible Web services. Furthermore, it provides 10 general utility elements for tasks such as accessing and embedding the current date and time into the content and formatting functionality for eliminating carriage returns and spaces. The XML syntax of the language allows the easy definition and extension of the general utility functionality.

The 8 core elements in the MyXML namespace are the *<myxml:single>*, *<myxml:multiple>*, *<myxml:loop>*, *<myxml:cgi>*, *<myxml:sql>*, *<myxml:dbcommand>*, *<myxml:dbitem>* and *<myxml:attribute>* elements.

The <myxml:single> element describes a *Single* variable that can be used arbitrary times in a MyXML document. The value of a <myxml:single> element is determined at run-time (i.e., provided by the application logic) and the same value is used whenever the element appears. A possible use of the <myxml:single> element is to print a customized welcome text depending on who is currently logged in. For example, the MyXML document:

```
<?xml version="1.0"?>
<welcome_text> Welcome to this site </welcome_text>
<myxml:single> name </myxml:single>
```

defines a welcome text and a *Single* variable **name** that is instantiated by the application logic at run-time (e.g., **name**=”Engin”, producing “Welcome to this site Engin”).

The <myxml:loop> and <myxml:multiple> elements provide the *Loop* and *Multiple* variable functionality. For all values provided as input for the <myxml:multiple> element, the part of the document enclosed in the <myxml:loop> element is processed. For example, the MyXML document:

```
<?xml version="1.0"?>
<myxml:loop>
    <welcome_text> Welcome to this site </welcome_text>
    <myxml:multiple> names </myxml:multiple>
</myxml:loop>
```

defines a welcome text for every name in the *Multiple* variable **names** that is instantiated by the application logic at run-time (e.g., **names**={“Engin”,”John”}, producing “Welcome to this site Engin”, “Welcome to this site John”). <myxml:loop> elements can be cascaded. Loops within other loops can be used, for example, to print a table containing all the books in a bookstore along with a list of authors for each book. The dimension of the *Multiple* variable is determined based on its position within the loop (i.e., dimension of 1 within the first loop, dimension of 2 within the second loop and so on).

The <myxml:cgi> element supports direct access to HTTP CGI parameters within a MyXML document. The definition of the <myxml:cgi> element has to correspond to the name of the CGI parameter it refers to (e.g., the name of the input field in an HTML form). For example, the MyXML document:

```
<?xml version="1.0"?>
<welcome_text> Welcome to this site </welcome_text>
<myxml:cgi> name </myxml:cgi>
```

defines a welcome text for a user who's name is received by a CGI parameter posted by a Web form.

The <myxml:attribute> element is used to define an attribute of a parent element that is not in the MyXML namespace. The usage and functionality of this element is similar to the <xsl:attribute> element from the XSL namespace. The main difference is that <myxml:attribute> can be used for *dynamic* content whereas <xsl:attribute> is only for static content. For example, the <myxml:attribute> element in the MyXML document:

```
<?xml version="1.0"?>
<a> Click here
   <myxml:attribute name="href">
       <myxml:single> url </myxml:single>
   </myxml:attribute>
</a>
```

defines a hypertext link and sets the *href* attribute of the HTML *a* element to the value of the *Single* variable **url** at run-time.

The *<myxml:sql>* element represents a database query. It is similar to the <myxml:loop> element. The document fragment enclosed by the <myxml:sql> element is processed for every record in the query's result set. Access to database fields is provided by the <myxml:dbitem> element. The query to be executed is defined by the <myxml:dbcommand> element and can contain other MyXML elements from the MyXML namespace such as *Single* variables. A possible use of the <myxml:sql> element is to generate XML from the content stored in a relational DBMS. For example, the MyXML document:

```
<?xml version="1.0"?>
<myxml:sql>
    <myxml:dbcommand>
         select * from names
    </myxml:dbcommand>
     <theName>
         <myxml:dbitem> name </myxml:dbitem>
    </theName>
</myxml:sql>
```

defines an SQL query in the database that selects all records in the table **names** and wraps the contents of the field **name** in XML *theName* tags. The resulting static content produced by the MyXML language compiler could look like this:

```
<?xml version="1.0"?>
<theName>
    Engin
</theName>
<theName>
    John
</theName>
...
```

The <myxml:currentdate> element is a good representative of the functionality provided by the general utility elements in the MyXML namespace. For example, the MyXML document:

```
<?xml version="1.0"?>
<date> Today's date is:
  <myxml:currentdate/>
</date>
```

defines the content "Today's date is:" enriched with the system date functionality. The MyXML language compiler inserts the necessary date for static content or produces the system date source code for dynamic content.

Other general utility functions allow the Web developer to control the parsing and formatting of the content and insert output produced by external system scripts and commands into it (e.g., inserting the output of the UNIX *ls* command into the content).

4.7.3 A simple MyXML example: Searching for musicals

Suppose a search form needs to be implemented. The form lets the user search for musicals in the Web site of a cultural organization that specializes in selling musical tickets. All musicals containing a certain keyword need to be retrieved from the database and have to be displayed in a Web page in a given layout. The content is dynamic because it is generated according to user input. The user enters a keyword that is then transmitted to the Web page.

Figure 4.8 shows the MyXML document that defines this functionality. There is a strict separation of content and layout as only the content and its structure are defined in the MyXML document. The example illustrates the use of CGI parameters and the handling of SQL queries with the MyXML language. The CGI parameter is used to construct the query string (see lines 6 and 7) and after the query is executed, the **title** field is extracted from the result set (see lines 9-11).

After the content has been defined, an XSL stylesheet is used to add a simple layout to the content. The search result is displayed in a table. Figure 4.9 depicts the XSL stylesheet used to format the output.

The XSL stylesheet generates HTML output and adds a heading to the document. For every record in the query's result set, a new row is added to the table. Of course, real world stylesheets would contain more complex rules and a more sophisticated layout would be defined.

```
1. <?xml version="1.0"?>
2. <!DOCTYPE VIF>
3. <VIF xmlns:myxml=".../ns/myxml">
4.  <query>
5.    <myxml:sql>
6.      <myxml:dbcommand>SELECT * FROM VIF_EVENTS WHERE title LIKE
7.        <myxml:cgi>musical_title</myxml:cgi>
8.      </myxml:dbcommand>
9.      <db_title>
10.        <myxml:dbitem>title</myxml:dbitem>
11.      </db_title>
12.    </myxml:sql>
13.  </query>
14. </VIF>
```

Figure 4.8: Example MyXML file to search in a database

```
1. <?xml version="1.0"?>
2. <xsl:style sheet version="1.0"
3.  xmlns:xsl=".../Transform"
4.  xmlns:myxml=".../ns/myxml">
5.  <xsl:import href="myxml.xsl"/>
6.  <xsl:output method="html" indent="yes"/>
7.
8.  <xsl:template match="query">
9.  <html><h2>The result of your search is:</h2>
10.    <table><xsl:apply-templates/></table>
11.  </html>
12.  </xsl:template>
13.
14.  <xsl:template match="db_title">
15.    <tr><td><xsl:apply-templates/></td></tr>
16.  </xsl:template>
17. </xsl:style sheet>
```

Figure 4.9: XSL stylesheet for formatting the output

```
1. public class VIF {
2.  protected HttpServletRequest request = null;
3.  protected ResultSet SQL0 = null;
4.  public VIF(HttpServletRequest request) {
5.    this.request = request;
6.  }
7.  protected String getCGIParameter(String paramName) {
8.    return request.getParameter(paramName);
9.  }
10.  protected ResultSet processSQLStatement(
11    String select, String user,
12.       String pwd, String url, String driver) {
13.         // do sql query using JDBC here!
14.  }
15. public void printContents(PrintWriter pw) {
16.    pw.println("<html>");
17.    pw.println("  <h2>");
18.    pw.println("    The result of your search is:");
19.    pw.println("  </h2>");
20.    pw.println("  <table>");
21.    printHTMLSQL0(pw);
22.    pw.println("  </table>");
23.    pw.println("</html>");
24.  }
25.  public void printContentsSQL0(PrintWriter pw) {
26.    try {
27.      SQL0 = processSQLStatement(
28.   "SELECT title, isbn_nr FROM VIF_EVENTS WHERE title LIKE"
29.         +getCGIParameter("musical_title")
30.              +";", "user", "pwd", "connect", "dbdriver");
31.      while (SQL0.next()) {
32.        pw.println("  <tr>");
33.        pw.println("    <td>");
34.        pw.println(SQL0.getString("title"));
35.        pw.println("    </td>");
36.        pw.println("  </tr>");
37.      }
…
```

Figure 4.10: Part of the generated Java Source Code

The Java source code that is generated from the MyXML document and the XSL stylesheet is shown in Figure 4.10. This generated Java class encapsulates the layout and the content information that was separately defined in the MyXML document and the XSL stylesheet file. Whenever a new layout is needed, only the XSL stylesheet has to be adapted.

The application logic can now create a new instance of this layout/content class and call its *printContents()* method (see line 15). The output produced by the method (e.g., see lines 16-23) is usually directly sent back to the calling client.

4.7.4 Another MyXML example: Shopping Cart

Suppose a flexible e-commerce Web service needs to be built for the cultural organization that specializes in selling musical tickets. The shopping cart is to allow users to manage tickets that they wish to buy.

```
1. <?xml version="1.0" encoding="US-ASCII"?>
2. <!DOCTYPE cart>
3. <cart xmlns:myxml="http://www.infosys.tuwien.ac.at/ns/myxml">
4.  <title>
5.   This is a simple shopping cart example using MyXML.
6.  </title>
7.
8.  <user>
9.  <myxml:single>username</myxml:single>
10. </user>
11.
12. <items>
13.  <myxml:loop>
14.   <item>
15.     <myxml:multiple name="id" create_name_element="no">
16.   product_id</myxml:multiple>
17.     <myxml:multiple name="name" create_name_element="no">
18.   product_name</myxml:multiple>
19.     <myxml:multiple name="quantity" create_name_element="no">
20.   product_quantity</myxml:multiple>
21.     <myxml:multiple name="price" create_name_element="no">
22.   product_price</myxml:multiple>
23.   </item>
24.  </myxml:loop>
25. </items>
26. </cart>
```

Figure 4.11: MyXML content definition for a shopping cart

Again, the content and the layout are defined in separate files. The dynamic content of the shopping cart is provided by the application logic at run-time. The application logic determines from user input which tickets have been booked and gives out the contents of the shopping cart using the MyXML generated layout/content encapsulating Java class.

Figure 4.11 depicts the MyXML document file for the shopping cart application. A <myxml:single> variable provides the value for the name of the user currently logged in. A <myxml:loop> element is used to iteratively step through the contents of the user's shopping cart. In the loop, <myxml:multiple> elements access the contents of the user's shopping cart. An item in the cart, consists of an ID number, a name, a price and the quantity of tickets the user wishes to order.

The layout of the shopping cart is defined with an XSL stylesheet. Figure 4.12 shows the XSL layout definition for the shopping cart. In this simple example, a welcome message is printed for the user and a table contains all the items currently stored in the shopping cart.

Note that a special *myxml.xsl* stylesheet is imported in the layout definition (see line 5). The imported stylesheet contains the default set of XSL processing rules for the MyXML namespace.

The rule for the <cart> element provides a basic HTML structure (see line 9-16). The content of the <user> element (see lines 8-10 in Figure 4.11) is used to print an introductory message including the user's name (see lines 18-20 in Figure 4.12). In the HTML document's body, a simple table is constructed containing the shopping cart's contents. For every <item> element (see lines 12-23 in Figure 4.11), a new table row is added that contains

```
1. <?xml version="1.0"?>
2.
3. <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4.    version="1.0">
5. <xsl:import href="myxml.xsl"/>
6. <xsl:output method="html" indent="yes"/>
7.
8. <!-- root element: create HTML skeleton -->
9. <xsl:template match="cart">
10.  <html><head><title>
11.     <xsl:value-of select="title"/>
12.    </title></head><body>
13.    <xsl:apply-templates/>
14.   </body>
15.  </html>
16. </xsl:template>
17.
18. <xsl:template match="user">
19.  <b>Shopping cart for user <xsl:apply-templates/></b>
20. </xsl:template>
21.
22. <xsl:template match="items">
23.  <table border="1"> <xsl:apply-templates /></table>
24. </xsl:template>
25.
26. <xsl:template match="item">
27.  <tr><td><xsl:apply-templates select="*[@name='id']" /></td>
28.   <td><xsl:apply-templates select="*[@name='name']" /></td>
29.   <td><xsl:apply-templates select="*[@name='quantity']" /></td>
30.   <td><xsl:apply-templates select="*[@name='price']" /></td>
31.  </tr>
32. </xsl:template>
33. </xsl:stylesheet>
```

Figure 4.12: XSL layout definition for the shopping cart

```
1. import java.io.*;
2. public class cart {
3. ...
4.  public cart(String username, String[] product_quantity,
5.       String[] product_id,
6.              String[] product_name, String[] product_price) {
7.  this.username = username;
8.  this.product_quantity = product_quantity;
9.  this.product_id = product_id;
10.  this.product_name = product_name;
11.  this.product_price = product_price;
12. }
13. public void printContents(PrintWriter pw) {
14.    pw.println("<html><head><title>");
15.    pw.println(
16.         "This is a simple shopping cart example using MyXML.");
17.   pw.println("</title></head><body>");
18.    pw.println(
19.         "This is a simple shopping cart example using MyXML.");
21.    pw.println(”<b>Shopping cart for user ");
22.    pw.println(username);
23.    pw.println("</b><table border=\"1\">");
24.    printContents0(pw);
25.    pw.println(”</table></body></html>");
26. }
27. public void printContents0(PrintWriter pw) {
28.  for(int i=0; i<product_id.length; ++i) {
29.    pw.println("<tr> <td>");
30.    pw.println(product_id[i]);
31.    pw.println("</td> <td>");
32.    pw.println(product_name[i]);
33.    pw.println("</td> <td>");
34.    pw.println(product_quantity[i]);
35.    pw.println("</td> <td>");
36.    pw.println(product_price[i]);
37.    pw.println("</td> </tr>");
38.  }
39. }
40. }
```

Figure 4.13: Part of the generated shopping cart Java code encapsulating the HTML code

```
1. // test application for cart.java
2.
3. import java.io.*;
4.
5. public class carttest {
6.  public static void main(String args[]) {
7.
8.    // provide shopping cart values
9.    String username = ”Engin Kirda";
10.    String[] id = {"1", "3", "4"};
11.    String[] name = {"PS/2 Mouse", "Cherry Keyboard",
12.         "Logitech Wingman"};
13.    String[] quantity = {"2", "1", "1"};
14.    String[] price = {"399", "599", "799"};
15.    cart c = new cart(username, quantity, id, name, price);
16.    PrintWriter pw = new PrintWriter(System.out);
17.    c.printContents(pw);
18.    pw.flush();
19.    pw.close();
20.    System.out.println("Done.");
21.  }
22. }
```

Figure 4.14: Invoking the generated code

table data elements for all the characteristics of the items in the shopping cart (i.e., ID, name, quantity and price) (see lines 22-24 and 26-32 in Figure 4.12).

Figure 4.13 shows the Java code that the MyXML language compiler generates (i.e., in a Java implementation) from the content and layout definitions and Figure 4.14 depicts how the generated code is invoked from the application logic (see lines 15-17 in Figure 4.14). Item information in the shopping cart such as the ID, name and quantity are passed to the layout/content code using string arrays (see lines 9-14 in Figure 4.14).

4.7.5 Post XSL stylesheet application

In some cases, it is not advantageous to generate static content in HTML/XML or source code functionality that produces dynamic content. There are situations when the developer does not wish to add a layout to the content during the implementation, but would like to keep it flexible and add it when the service is run. For example, the layout may be changing often and generating HTML or source code might be costing too much time.

Post stylesheets are XSL stylesheets that can be applied to the content at run-time. For example, if the first XSL stylesheet produces XML data instead of adding a layout to the content, a second stylesheet, the post XSL stylesheet, can be used to process it and add a layout. When the MyXML-generated source code is compiled and run, the specified post stylesheet is automatically applied to the generated content.

4.8 XSL stylesheet pre-processing for stylesheet reuse

The examples given in the previous sections used a single XSL definition to add a layout to the content. Clearly, if the aim is to support multiple devices, it is possible to use a different stylesheet for every device.

The problem is that as the number of devices increase, the number of stylesheets increase proportionally. The stylesheets are often quite similar with respect the processing rules. Often, the only difference is the formatting specifications and the Web format being used (e.g., HTML, WML, etc.). In a service with 4 stylesheets, for example, 12 stylesheets would be necessary to support 3 different devices. There would be much redundancy and hence, the maintenance overhead would significantly increase.

Before a layout is added to the content, a technique called *XSL stylesheet pre-processing* is used to eliminate the described duplication and enable the reuse of existing XSL stylesheets: Instead of the traditional approach of using a new XSL stylesheet for every new device, the information necessary for the device is *integrated* into the existing stylesheets using special descriptors that help differentiate between the device-specific layout in the stylesheets. The MyXML language compiler processes these specifications and generates the appropriate XSL stylesheet for a particular device.

Figure 4.15 depicts a portion of an XSL stylesheet from a commercial Web site. The single XSL match template (see lines 1-42) defines an HTML layout for traditional full-fledged browsers and a simpler HTML version of the page for PDAs. The *@myxml:device* descriptors (e.g., see lines 3 and 9) are used to define device-specific output. In the example, HTML for traditional browsers is the default device family (see Section 4.1) and this output is marked in *@myxml:device:default* blocks (e.g., see lines 3-6). The PDA-specific output, on the other hand, is marked in *@myxml:device:pda* blocks (e.g., see lines 9-14). New devices can be added to the stylesheets by embedding layout content in blocks of the form *@myxml:device:<Name of device>*, where the name can be any string description of a device family.

In the example, a descriptor of the form *@myxml:device:default,pda* in the stylesheet (see lines 16-18) indicates that the layout is valid for the default device family as well as the PDA device family. A comma can be used between device names to signal the compiler that the following layout definition is valid for more than one device.

Figure 4.17 shows the XSL stylesheet for the traditional browser version of the page after pre-processing. Figure 4.16 shows the XSL stylesheet for the PDA version of the page after pre-processing. Pre-processing filters out the layout definitions that are not needed for the device for which the XSL stylesheet is being generated.

The XSL pre-processing technique eliminates the need to copy the stylesheets and adapt them for a new device. This is important because XSL stylesheets can become quite complex in real-world Web sites. Using a separate stylesheet for a new device only shifts the problem of *copying* and adapting source code to *copying* and adapting stylesheets.

```
1. <xsl:template match="event_list">
2.    <!-- %%%%%%%%%%%%% HTML %%%%%%%%%%%%%%%%%%%%%-->
3.    @myxml:device:default{
4.          <table border="0">
5.          <tr><td><b>Event</b></td><td><b>Location</b></td></tr>
6.    }@myxml:device
7.
8.    <!-- %%%%%%%%%%%%% PDA %%%%%%%%%%%%%%%%%%%%%%-->
9.    @myxml:device:pda{
11.         <xsl:apply-templates select="//explanation"/>
12.         <table border="1" cellspacing="0" cellpadding="2">
13.         <tr><td><b>Events</b></td><td><b>Location</b></td></tr>
14.   }@myxml:device
15.
16.   @myxml:device:default,pda{
17.         <xsl:apply-templates/>
18.   }@myxml:device
19.
20.   <!-- %%%%%%%%%%%%% HTML %%%%%%%%%%%%%%%%%%%%%-->
21.   @myxml:device:default{</table> }@myxml:device
22.
23.   <!-- %%%%%%%%%%%%% PDA %%%%%%%%%%%%%%%%%%%%%%-->
24.   @myxml:device:pda{
25          </table><br/> <table border="0" width="400">
26.         <tr><td alight="left">
27.   <a href="/wf/
28.         displayevents?device=pda">
29.         <img border="0" src="/images/english/buttons/
30.         vorige.gif"
31.         alt="Previous events"/></a></td>
32.   <td alight="right">
33.   <a href="/wf/
34.   displayevents?device=pda">
35.         <img border="0" alt="More events"
36.         src="/images/english/buttons/naechste.gif"/></a>
37.   </td></tr></table>
38.   <a href="/wf/collector?device=pda">
39.         Back to the
40.          first event </a>
41.   }@myxml:device
42. </xsl:template>
```

Figure 4.15: XSL Stylesheet reuse with pre-processing

```
1. <xsl:template match="event_list">
3.          <xsl:apply-templates select="//explanation"/>
4.    <table border="1" cellspacing="0" cellpadding="2">
5.    <tr><td><b>Events</b></td><td><b>Location</b></td></tr>
6.          <xsl:apply-templates/>
7.    </table>
8.    <br/>
9.    <table border="0" width="400">
10.   <tr><td alight="left">
11.   <a href="/wf/
12.   displayevents?device=pda">
14.         <img border="0" src="/images/english/buttons/vorige.gif"
15.         alt="Previous events"/></a>
16.   </td>
17.   <td alight="right">
18.   <a href="/wf/
19.   displayevents?device=pda">
21.         <img border="0" alt="More events"
22.         src="/images/english/buttons/naechste.gif"/></a>
23.   </td></tr></table>
26.   <a href="/wf/collector?device=pda"> Back to
27.   the first event </a>
29. </xsl:template>
```

Figure 4.16: XSL Stylesheet for PDA access after pre-processing

```
1. <xsl:template match="event_list">
2.    <table border="0">
3.    <tr><td><b>Event</b></td><td><b>Location</b></td></tr>
4.          <xsl:apply-templates/>
5.    </table>
6. </xsl:template>
```

Figure 4.17: XSL Stylesheet for full HTML access after pre-processing

4.9 Page splitting

The main idea behind page splitting in Web site construction is to split and partition the content in XSL layout files by *grouping* layout elements. A *group* identifies a single unit of information on the page that a device family is able to display. Groups can also be partitioned and split using *subgroups* and thus, different splitting granularities can be achieved.

Figure 4.18 illustrates the concept of grouping and subgrouping on a commercial Web page (belonging to the Vienna International Festival Web site) that displays a list of cultural events (i.e., exhibitions, ensembles, theaters, performances) and their locations. On the page, the entire event information has been marked as belonging to a group. Every two events on the page make up a subgroup.

Depending on the order they appear on a page, each group and subgroup implicitly receives an ID to make it uniquely identifiable (the ID count starts from 0). In the Figure, for example, there is one group with the ID 0, and the depicted subgroups have IDs 0, 1 and 2.

Each time the layout is presented, only the information in a single group is displayed and

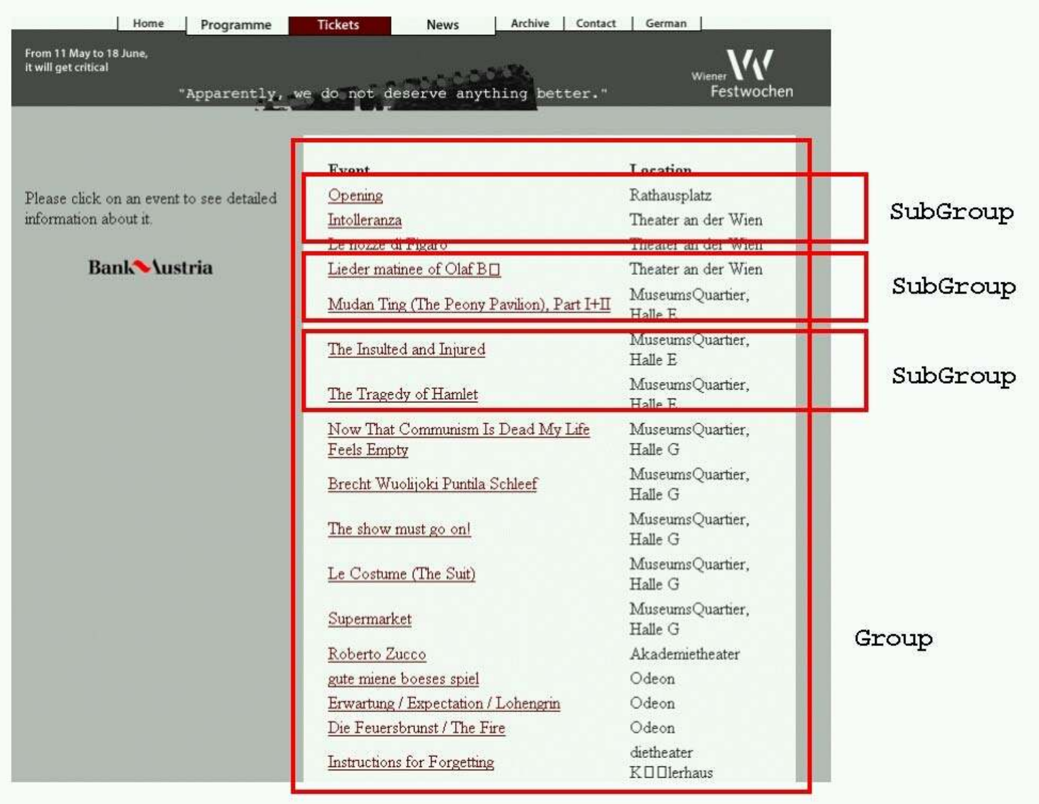


Figure 4.18: Page splitting using groups and subgroups

the other groups are ignored. If, for example, the layout in Figure 4.18 is presented with the group ID 1, no event information would not be displayed and one would only see layout elements that do not belong to any groups (i.e., the header, title, logo in the page).

If a group contains subgroups, similarly, the subgroups are displayed one after one. Only the group and subgroup with the given ID would be displayed. For example, to display the information in the second subgroup, the layout in Figure 4.18 would be presented with the group ID 0 and subgroup ID 1.

By setting a so called *step* value, the subgroups that are to be displayed can be further adjusted. For example, a step value of 2 and a subgroup ID of 0 would display the first and the second subgroup and then stop. The next time the layout is presented, the next two subgroups would be displayed. With a step value of 3, the first three subgroups would be displayed, then the next three and so on.

The described simple mechanism allows the selection of portions of a page during Web site construction so that they can be incrementally displayed on devices with small displays and limited memory sizes.

The *page splitting* processor in the DIWE framework is responsible for giving out the information partially over many smaller steps. It keeps track of the group and subgroup numbers and can receive commands on which splits (i.e., layout fragments) to give out.

4.9.1 Page splitting descriptors and parameters

Parameter	Description
ui	Indicates the group number to display
sg	Indicates the subgroup number to display
reset	Signals that all internal counters (e.g., subgroup and group count) should be reset

Table 4.1: Page splitting-related CGI parameters that the page splitting processor interprets

Descriptor	Functionality
@myxml:nextGroup	Substitute this descriptor with the next group number
@myxml:currentGroup	Substitute this descriptor with the current group number
@myxml:previousGroup	Substitute this descriptor with the previous group number
@myxml:currentSubgroup	Substitutes this descriptor with the current subgroup number
@myxml:previousSubgroup	Substitutes this descriptor with the previous subgroup number
@myxml:nextSubgroup	Substitutes this descriptor with the next subgroup number

Table 4.2: Descriptors that the page splitting processor substitutes at run-time

Group and subgroup information is inserted into the XSL stylesheets using @*myxml:group* and @*myxml:subgroup* descriptors. When processing an output stream at run-time, the *page splitting* processor looks for these descriptors to split a page.

Further, a Web page is constructed in such a way that when a user follows a link, the *page splitting* processor is invoked with CGI parameters that signal to it which group and subgroup number it should display. Table 4.1 lists the page splitting-related CGI parameters the *page splitting* processor understands. The *ui* parameter indicates the group and *sg*, the subgroup number to display. The *reset* parameter can be used to reset all internal group and subgroup counters. The *page splitting* processor can accept CGI parameters using both the GET and POST HTTP methods.

Because it is not always possible to know how many splits a page consists of and what the next group or subgroup number is, the *page splitting* processor is able to recognize and interpret descriptors at run-time that request group/subgroup information. Table 4.2 lists descriptors that the *page splitting* processor substitutes with appropriate values. The *@myxml:nextGroup* descriptor, for example, is substituted with the next group number in the *page splitting* processor's internal counters.

4.9.2 A simple page splitting example

Suppose the information on the HTML page in Figure 4.18 has to be displayed on a WAP device. Clearly, the information on the HTML page is too large and cannot be displayed in a single WML page.

```
1. <event_list>
2.   <myxml:sql>
3.    <myxml:dbcommand>
4.    select * from WF2001_EVENTSENGLISH as e, WF2001_LOCATION
5.     as l where (e.location_id=l.id)
6.    </myxml:dbcommand>
7.    <event>
8.          <title>
9.          <link>
10          <myxml:dbitem> title </myxml:dbitem>
11          </link>
12          </title>
13    </event>
14    </myxml:sql>
15. </event_list>
```

Figure 4.19: MyXML document for the events page

```
1.    <xsl:template match="event">
2.          <tr>
3.          <xsl:apply-templates/>
4.          </tr>
5.    </xsl:template>
6.
7.    <xsl:template match="title">
8.          <td>
9.          <xsl:apply-templates select="link"/>
10.         </td>
11.   </xsl:template>
12.
13.   <xsl:template match="link">
14.         <a> <xsl:apply-templates/> </a>
15.   </xsl:template>
```

Figure 4.20: XSL layout definition for HTML event page

```
1.    <xsl:template match="event">
2.          @myxml:group{
3.                <xsl:apply-templates/>
4.          }@myxml:group
5.    </xsl:template>
6.
7.    <xsl:template match="title">
8.          @myxml:subgroup{
9.                <p>
10.               <xsl:apply-templates select="link"/>
11.               </p>
12.               <a href="/collector?ui=@myxml:nextGroup
13.               &amp; sg=@myxml:nextSubGroup">
14.               Next Page
15.               </a>
16.         }@myxml:subgroup
17.   </xsl:template>
18.
19.   <xsl:template match="link">
20.         <a> <xsl:apply-templates/> </a>
21.   </xsl:template>
```

Figure 4.21: XSL layout definition for WML event page

Figure 4.19 shows the simplified MyXML document for the page. A <myxml:sql> element selects the event information from a relational database. The event names in the database column “title” are picked using the <myxml:dbitem> element.

Figure 4.20 shows a part of the XSL layout definition for the HTML version of the page. Figure 4.21 shows the layout definition for the WML version of the page. The stylesheets are quite similar except for the differences in HTML and WML tags and the grouping and subgrouping in the WML definition (e.g., compare the lines 1-5 in Figure 4.20 with the lines 1-5 in Figure 4.21).

In the WML stylesheet in Figure 4.21, groups and subgroups are defined using the descriptors *@myxml:group* and *@myxml:subgroup*. As in the page shown on Figure 4.18, the entire event information is defined in a group, but each event is marked as a subgroup (see lines 1-5 and 7-17). The *@myxml:nextGroup* and *@myxml:nextSubGroup* descriptors in the stylesheet are automatically replaced by the next group and subgroup numbers by the *page splitting* processor at run-time. Whenever the user clicks the “Next Page” link, the next subgroup in the current group is presented by the *page splitting* processor. If there are no more subgroups, the next group and the first subgroup in the group are fetched and the information is incrementally given out to the device. Setting the stepping count to 2, hence, would give out the subgroups at 2 step intervals.

In the example, the *page splitting* processor is accessed via the URL */collector* (see line 12 in Figure 4.21) and receives two parameters: *ui* for the group and *sg* for the subgroup number to display.

4.10 Process partitioning

The page splitting concept solves the problem of dealing with different page sizes various Web devices are able to display. A database query, for example, can be made to display three results per page for WAP devices and ten results per page for desktop browsers.

An important problem page splitting does not solve, however, is how to deal with interactive Web pages that use Web forms. If a Web form in a page is split and distributed over two other pages, for example, it will not work because every Web form has a corresponding *target URL* (i.e., the application logic) that it invokes with the parameters it collects. Hence, if the parameters on the first page are submitted, the information in the following pages will be missing.

Process partitioning is a technique that allows Web developers and designers to deal with Web form-based dynamic interactions on devices with display and memory size limitations. Process partitioning uses the page splitting technique to incrementally display Web forms and provides a mechanism to partition the interactive process over a number of independent steps.

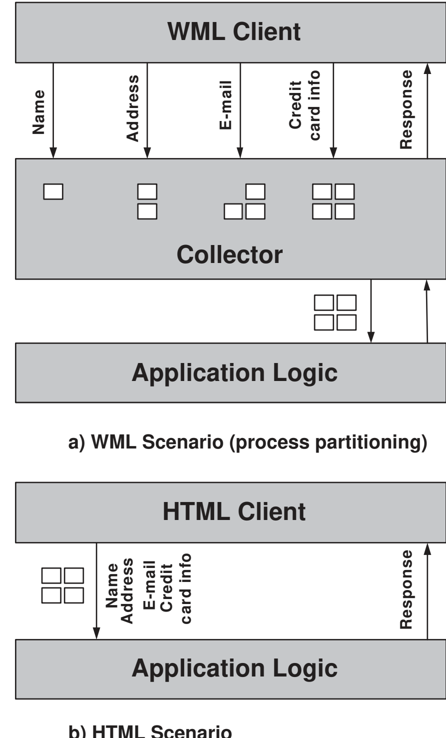


Figure 4.22: An online WML-based order with process partitioning compared to a traditional HTML-based order

Using process partitioning in a WAP e-commerce application for selling cultural event tickets, for example, the ordering process would be distributed over several WML pages. Each time a part of the required information would be collected (e.g., customer's name in the first step, her address in the second, and so on) and sent to an intermediary processor that temporarily stores the input. The intermediary processor would invoke the application logic

with the input data it has collected when all necessary data is submitted. In the DIWE framework, the functionality of the intermediary processor is provided by the *process partitioning* processor.

Figure 4.22 depicts the ordering of an event ticket using HTML and WML. The HTML page is able to collect all of the information in a single page, but the process is partitioned into multiple steps for WML.

4.10.1 Process partitioning parameters

Parameter	Description
colstat	Signals that the collection process is finished
target	Indicates the target URL to invoke once collection process is finished

Table 4.3: Table showing process partitioning-related CGI parameters the Collector component understands

Two CGI parameters are used by the *Collector* component that help control the input collection process over several steps. Table 4.3 lists these parameters and describes their functionality. The *colstat* CGI parameter is used to signal the *process partitioning* processor that the collection is finished. The *process partitioning* processor then invokes the URL that it receives with the *target* parameter.

4.10.2 A simple process partitioning example

Figure 4.23 depicts the XSL stylesheet for a simple HTML Web form that displays four input fields and collects the user's name, age and address information and some miscellaneous comments. A button is placed at the bottom of the form (see line 5) and the user has to press it to submit the information. The POST method is used to submit the values in the input fields and the target URL that processes the results is a program (i.e., servlet, script, etc.) */show* in the example (see line 3). Figure 4.24 shows the Web form as seen on a desktop browser.

Figure 4.25 depicts the XSL stylesheet for the same HTML Web form that has been partitioned into two HTML pages using page splitting. The target URL has been changed from */show* to */collector* (see line 3) which is the URL for the *process partitioning* processor in this example. The *process partitioning* processor accepts a command parameter *ui* that indicates which group in the page should be displayed next. This command is embedded into the form as a hidden input element and uses a @*myxml:nextGroup* identifier to retrieve the next group number (see line 5).

```
1. <xsl:template match="page">
2.    <html><head> <title> Page </title> </head> <body>
3.    <form action="/show" method="post">
4.          <xsl:apply-templates/>
5.    <input type="submit"/>
6.    </form>
7.    </body></html>
8. </xsl:template>
9.
10. <xsl:template match="name">
11.   <h2>
12.         <xsl:apply-templates/>
13.         <input type="text" name="name"/>
14.   </h2>
15. </xsl:template>
16.
17. <xsl:template match="age">
18.   <h2>
19.         <xsl:apply-templates/>
20.         <input type="text" name="age"/>
21.   </h2>
22. </xsl:template>
23.
24. <xsl:template match="address">
25.   <h2>
26.         <xsl:apply-templates/>
27.         <input type="text" name="address"/>
28.   </h2>
29. </xsl:template>
30.
31. <xsl:template match="misc">
32.   <h2>
33.          <xsl:apply-templates/>
34.         <input type="text" name="misc"/>
35.   </h2>
36. </xsl:template>
```

Figure 4.23: XSL layout definition for HTML Web form

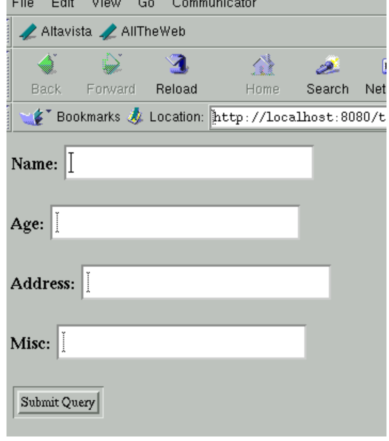


Figure 4.24: Screenshot of simple HTML Web form

```
1. <xsl:template match="page">
2.    <html><head> <title> Page </title> </head> <body>
3.    <form action="/collector" method="post">
4.          <xsl:apply-templates/>
5.    <input type="hidden" name="ui" value="@myxml:nextGroup"/>
6.    <input type="submit"/>
7.    </form>
8.    </body></html>
9. </xsl:template>
10.
11. <xsl:template match="main">
12.   @myxml:group{
13.         <xsl:apply-templates select="name"/>
14.         <xsl:apply-templates select="age"/>
15.         <input type="hidden" name="target" value="/show"/>
16.   }@myxml:group
17.
18.   @myxml:group
19.         <xsl:apply-templates select="address"/>
20.         <xsl:apply-templates select="misc"/>
21.         <input type="hidden" name="colstat" value="true"/>
22.   }@myxml:group
23. </xsl:template>
24.
25. <xsl:template match="name">
26.   <h2>
27.          <xsl:apply-templates/>
28.         <input type="text" name="name"/>
29.   </h2>
30. </xsl:template>
31.
32. <xsl:template match="age">
33.   <h2>
34.         <xsl:apply-templates/>
35.         <input type="text" name="age"/>
36.   </h2>
37. </xsl:template>
38.
39. <xsl:template match="address">
40.   <h2>
41.         <xsl:apply-templates/>
42.         <input type="text" name="address"/>
43.   </h2>
44. </xsl:template>
45.
46. <xsl:template match="misc">
47.   <h2>
48.         <xsl:apply-templates/>
49.         <input type="text" name="misc"/>
50.   </h2>
51. </xsl:template>
```

Figure 4.25: XSL layout definition for the partitioned HTML Web form

In the XSL template for *main* (see lines 11-23), two groups have been defined with *@myxml:group*. The *name* and *age* input fields are in the first group and the *address* and *misc* input fields are in the second (see lines 12-16 and 18-22).

There is a hidden input named *target* in the first group (see line 15). This is a special parameter that is passed to the *process partitioning* processor and that identifies the target URL for this collection session. In the case of the example, this is the URL */show*. The *process partitioning* processor, hence, knows that it has to invoke this URL once it has collected all parameters from both groups. The *colstat* hidden input in the second group (see line 21) signals the *process partitioning* processor that it can invoke the target URL after it has received the results of the second group. It indicates that there are no more groups and input parameters in the page.

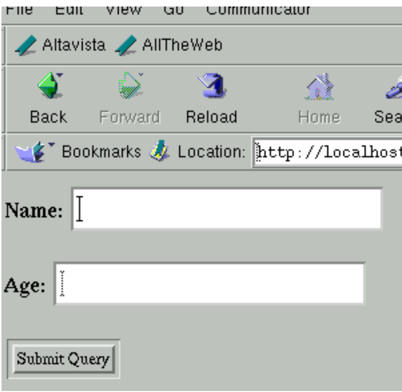


Figure 4.26: Screenshot of the partitioned HTML Web form – First group

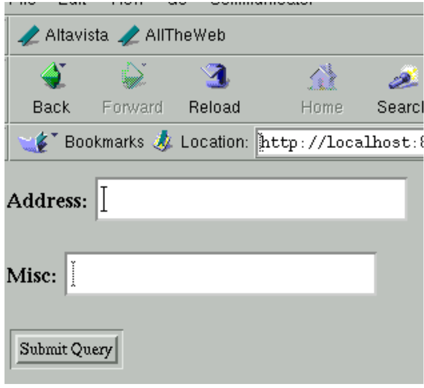


Figure 4.27: Screenshot of the partitioned HTML Web form – Second group

Figures 4.26 and 4.27 depict screenshots of the partitioned HTML Web forms as seen on a browser. Once the information in the first group has been submitted (i.e., first page), the second group is displayed (i.e., second page) and the user is prompted for input. Pressing the submit button in the second group invokes the application logic at the URL */show*.

4.11 Device-independent application logic interfacing

The traditional approach to supporting different layouts with the same application logic is to build conditional (e.g., *if/then/else*) statements into the code and to present the layout based on some criteria (e.g., the user chooses a device name from a list). For example, the following pseudo application logic code:

```
  ... do some domain specific task ...
if (device="html") present HTML_LAYOUT
else if (device="wap") present WAP_LAYOUT
  ... do some domain specific task ...
```

checks the value of a variable named *device* and presents the appropriate HTML or WAP layout.

The disadvantage of this approach is that the application logic has to be modified and extended for every new device that is being supported. While writing the application code, the Web developer often needs to know in advance what type of devices will be supported. She has to try to design and optimize the code so that it can easily be extended: A task that is not always easy to achieve.

The *logic interfacing* processor provides a solution to this problem and allows the application logic to be reused for arbitrary devices. It acts as a wrapper to the layout/content and eliminates the need for the application logic to explicitly choose and invoke a MyXML-generated layout/content class.

4.11.0.1 Calling the logic in three steps

The first step in invoking a MyXML-generated layout/content class in a device-independent way is to create an instance of the *logic interfacing* processor.

In the second step, instead of directly instantiating the layout/content class with the parameters it requires (e.g., as in the MyXML Web service construction example presented in Section 4.7.4), the parameters are written in an array.

The *logic interfacing* processor provides a method *invoke()* that the application logic can use to invoke layout/content classes. In the third step, this method is used to present the output of the appropriate device-specific layout/content class.

The *invoke()* method has the following signature (i.e., in a Java implementation):

```
public void invoke(String name, Object array[]);
```

The method accepts two parameters: a string containing the name of the layout/content class to be invoked and an Object array that the application logic uses to pass the parameters that the layout/content class requires.

The *logic interfacing* processor uses a simple trick to enable a single *invoke()* method in the application logic to work for arbitrary layout/content classes: A class naming convention is used to identify layout/content classes that belong to the same page and this enables the *logic interfacing* processor to automatically instantiate and invoke the appropriate layout/content class. The default device family layout/content class name for a page

```
1.    Output output = new Output(new
2.          CheckoutRequestWrapper(request), response);
3.    Object[] params = new Object[20];
4.
5.    params[0] = cart.getTotalNumberOfTickets();
6.    params[1] = cart.getMinimumPrice();
7.    params[2] = cart.getMaximumPrice();
8.    params[3] = "";
9.    params[4] = "";
10.   params[5] = "";
11.   params[6] = "";
12.   params[7] = "";
13.   params[8] = "";
14.   params[9] = "";
15.   params[10] = "";
16.   params[11] = "";
17.   params[12] = cart.getEventName();
18.   params[13] = cart.getEventLocation();;
19.   params[14] = cart.getEventDate();
20.   params[15] = cart.getEventTime();
21.   params[16] = cart.getNumberOfTickets();
22.   params[17] = cart.getCategoryName();
23.   params[18] = cart.getCategoryInfo();
24.   params[19] = creditCards;
25.   output.invoke("Checkout",params);
```

Figure 4.28: Invoking the *Checkout* layout/content class from the application logic

```
1. public Checkout(String totalNumberOfTickets,String minimumPrice,
2. String maximumPrice,String errorMessage,String name,String address,
3. String phonePrivate,String phoneWork,String email,String comments,
4. String cardNumber,String validThru,String event_name[],
5. String event_location[],String event_date[],String event_time[],
6. String number_of_tickets[][],String category_name[][],
7. String category_info[][],String creditCard[]) {
8.          this.totalNumberOfTickets=totalNumberOfTickets;
9.          this.minimumPrice=minimumPrice;
10.         this.maximumPrice=maximumPrice;
11.         this.errorMessage=errorMessage;
12.         this.name=name;
13.         this.address=address;
14.         this.phonePrivate=phonePrivate;
15.         this.phoneWork=phoneWork;
16.         this.email=email;
17.         this.comments=comments;
18.         this.cardNumber=cardNumber;
19.         this.validThru=validThru;
20.         this.event_name=event_name;
21.         this.event_location=event_location;
22.         this.event_date=event_date;
23.         this.event_time=event_time;
24.         this.number_of_tickets=number_of_tickets;
25.         this.category_name=category_name;
26.         this.category_info=category_info;
27.         this.creditCard=creditCard;
28. }
```

Figure 4.29: The MyXML-generated *Checkout* layout/content class

is taken as the base identifier and device names are apprehended to this identifier for all other device-specific layout/content classes. For example, if the name of the layout/content class is *homepage* for the default device family, for the PDA layout/content class this would be *homepagepda* (i.e., device name is "pda"), for the WAP layout/content class this would be *homepagewap* (i.e., device name is "wap") and so on. A CGI parameter called *device* signals the *logic interfacing* processor the name of the device the application logic is being invoked for. For example, if the user is visiting the home page that is available at the URL */homepage*, calling the URL as */homepage?device=pda* would make the *logic interfacing* processor invoke the PDA layout/content class *homepagepda*. The component would add the "pda" device name to the default family layout/content class name (i.e., *homepage*) and instantiate that class with the parameters.

The advantage of this simple approach is that the application logic is device-independent: It can be used for many devices as long as the developer keeps to simple naming conventions that the *logic interfacing* processor can correctly interpret.

4.11.0.2 A simple example

Figure 4.28 depicts a part of the Java application logic from an e-commerce Web application. First, an instance of the *logic interfacing* processor is created (a Java implementation called Output in this case – see lines 1-2). Then, a Java Object array is created that accepts 20 parameters (see lines 3-24). Finally, the layout/content class is invoked using its class name (i.e., *Checkout* in this case) and the parameters it requires (see line 25). Figure 4.29 depicts the constructor of the *Checkout* layout/content class.

Suppose an alternative PDA layout has to be provided. To cover this requirement, first, a PDA layout/content class would be created using the MyXML compiler. Following the layout/content class naming conventions, the device family name would be apprehended to the name of the default device family layout/content class. The PDA layout/content class, hence, would be called *Checkoutpda*. The *logic interfacing* processor would instantiate and invoke the appropriate layout/content class based on the name of the device family the application logic is being invoked for.

4.12 Summary

This chapter introduced a novel conceptual framework for device-independent Web engineering. The Device-Independent Web Engineering (DIWE) framework consists of the *MyXML language*, a compiler that can interpret the language, and four basic run-time *processors* that are configured and deployed on the Web server at run-time to provide device-independence support. These processors are Web services themselves. The framework introduces two novel techniques, *page splitting* and *process partitioning* by layout marking, that allow the Web developer to tune the selected information and the sizes of generated pages according to the characteristics of a device that is being targeted. The framework also introduces a novel technique called *XSL stylesheet pre-processing* that allows the reuse of *existing* XSL stylesheets when adding new devices to a Web service.

Chapter 5

The MyXML tool suite: A prototype implementation

This chapter presents the MyXML tool suite, an implementation of the Device-Independent Web Engineering (DIWE) framework discussed in the previous chapter. The tool suite consists of the MyXML compiler, three configurable run-time device-independence components and a visual Integrated Development Environment (IDE).

5.1 The MyXML tool suite

The MyXML compiler and the MyXMLDesigner IDE are development tools used to construct flexible, XML/XSL-based Web services using the MyXML language. The configurable device-independence components in the tool suite are implementations of the *device detection*, *logic interfacing*, *page splitting* and *process partitioning* processors discussed in the previous chapter. These components are configured and deployed on the Web server at run-time to provide device-independence support.

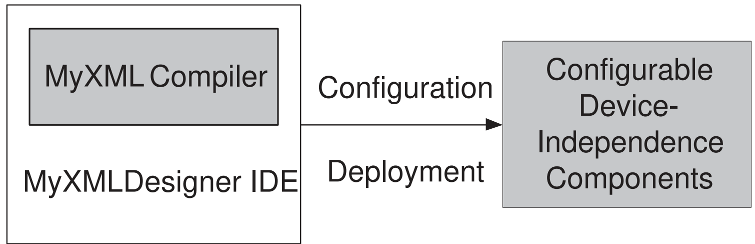


Figure 5.1: Relations between the tools in the MyXML tool suite

Each tool in the tool suite addresses a specific part of the Device-Independent Web Engineering (DIWE) framework. Table 5.1 shows the Web service life cycle phases each tool in the tool suite supports and Table 5.2 shows the functionality each one provides.

Figure 5.1 depicts the relations between the tools in the suite. MyXMLDesigner is a visual development environment and a user-friendly graphical front-end to the functionality

Phase/Tool	MyXML Compiler	Device-Independence Components	MyXMLDesigner IDE
Design			X
Implementation	X	X	X
Deployment			X
Maintainence			X

Table 5.1: The Web service life cycle phases each tool in the MyXML tool suite supports

Functionality/Tool	MyXML Processor	Device-Independence Components	MyXMLDesigner IDE
LCL Separation (with XML/XSL)	X		
Logic Reuse		X	
XSL Reuse	X		
RDBMS Integration	X		
User-friendly IDE			X
Device Detection		X	
Device Configuration		X	X
Device Management			X
XML Content and Layout Generation	X		
Layout Adaptation		X	

Table 5.2: The functionality provided by the tools in the MyXML tool suite

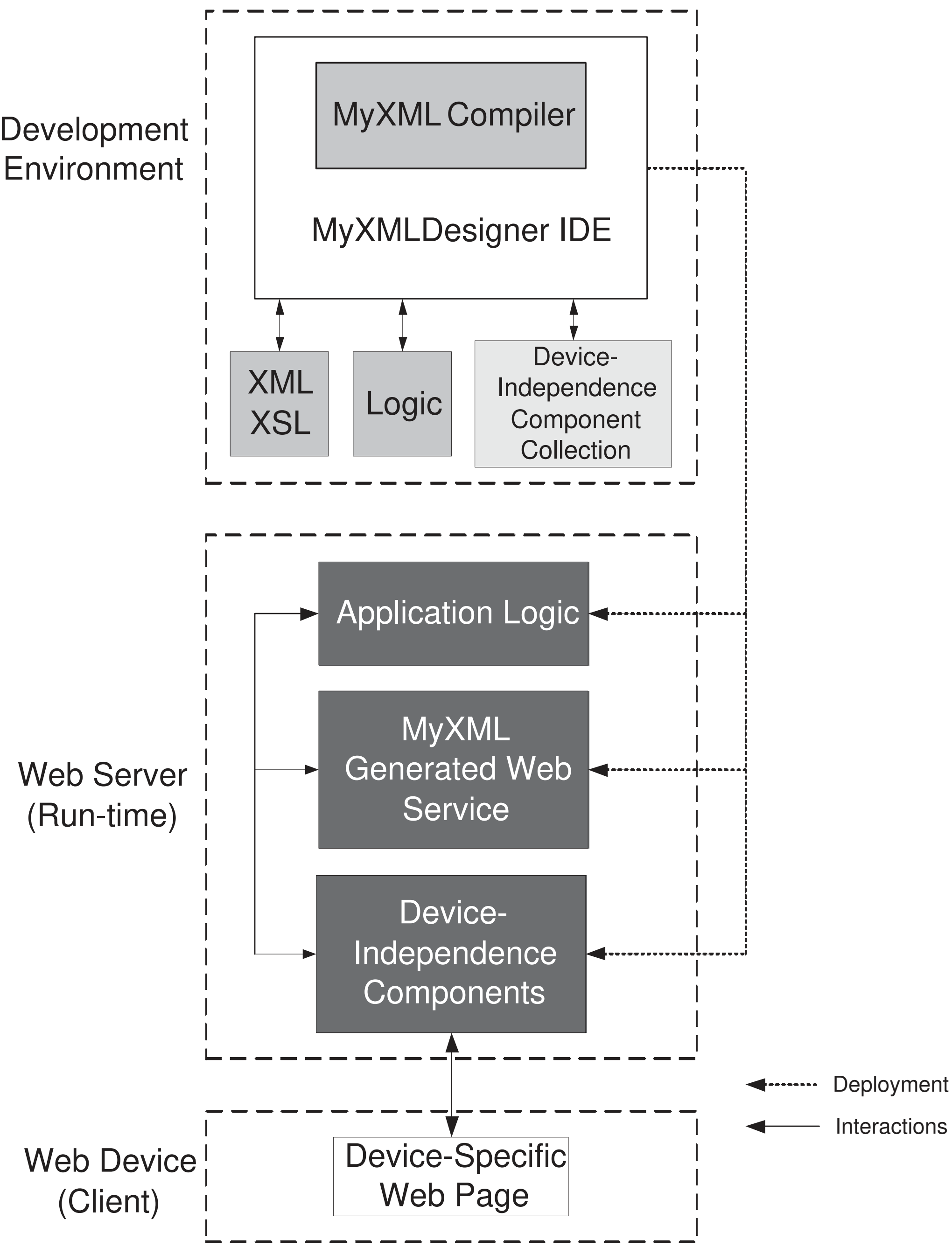


Figure 5.2: The MyXML tool suite in Web service construction and operation based on the DIWE framework

provided by the tools in the tool suite. Device-independent Web sites can be created and maintained with MyXMLDesigner and interactive functionality can be constructed.

Based on MyXML language specifications, MyXMLDesigner uses the MyXML compiler to generate static content embedded in HTML or XML, or Java source code that provides interactive functionality.

Although the MyXML compiler and the device-independence components can be configured and deployed manually, the MyXMLDesigner IDE has integrated support for their automated, user-friendly configuration, deployment and usage.

Figure 5.2 illustrates the role of the MyXML tool suite in Web service construction and operation based on the DIWE framework discussed in the previous chapter. A typical development environment consists of MyXMLDesigner and the MyXML compiler. The device-independence components are stored in a repository (i.e., component collection) integrated into the MyXMLDesigner IDE. The developer creates (or integrates) XML content and XSL layout definitions. If static layout is being generated, there is no need for application logic. If dynamic content is being created, however, an application logic (i.e., Java source code) is created (or integrated) using editors in MyXMLDesigner. The application logic, the generated layout and source code files, and the configured device-independence components are automatically compiled, configured and deployed on the Web server.

5.2 The MyXML compiler

The implementation of the MyXML language compiler in the MyXML tool suite is a pluggable, stand-alone application. As a part of this dissertation, three versions of the MyXML compiler have been developed since early 2000: rudimentary prototypes to estimate the feasibility of the tool (e.g., [KK01, KK00]) and the final version that is pluggable into external applications and that can support arbitrary content types and XML content. This section focuses on the final version (called MyXML version 1.3 Xenon).

5.2.1 Usage

The MyXML compiler has a command-line interface that can be used to invoke it by hand or from scripts. It can be started with the syntax:

```
java myxml.Xenon <MyXML File> <XSL File>
       -p <XSL Post Style> <Class/Document Name>
       <ConnectURL> <User name> <Password>
       <Device Name>
```

The user provides:

- A MyXML document file
- An XSL stylesheet that defines the layout
- An optional XSL post stylesheet

- A Java class name for the generated layout/content class or a name for the generated static content HTML/XML file
- A connection URL, user name and password for the relational database
- A device name for XSL stylesheet pre-processing

The compiler also provides a Java Application Programming Interface (API) to its functionality and can be configured and invoked from inside programs. The MyXMLDesigner visual IDE uses this API to start the MyXML compiler.

5.2.2 Implementation

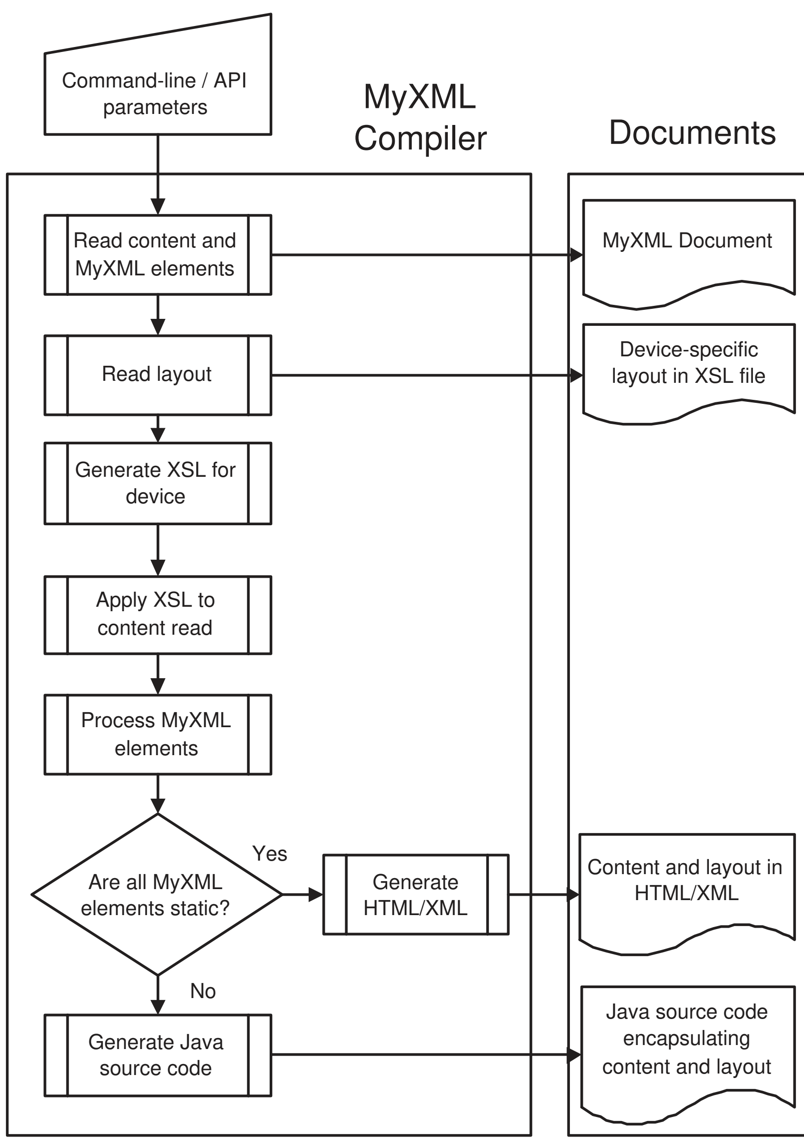


Figure 5.3: Flowchart showing the main steps taken by the MyXML compiler

The flowchart in Figure 5.3 shows the main steps taken by the MyXML compiler during the processing of MyXML documents and XSL specifications.

The compiler is invoked using either command-line parameters or its Application Programming Interface (API). First, the compiler reads the MyXML document it is given. Then, it reads the XSL layout definition. Based on the device for which the content and layout is being generated, XSL stylesheet pre-processing is performed and an XSL stylesheet is generated for the target device.

The generated device-specific XSL stylesheet is applied to the MyXML document and the elements from the MyXML namespace are parsed and interpreted. If all MyXML elements are static (i.e., do not contain any variables, loops, CGI elements that need to be instantiated at run-time), an HTML or XML file is generated based on the layout information in the XSL definition. If dynamic MyXML elements exist, on the other hand, a Java source code file (i.e., Java class) is generated that encapsulates the content and the layout.

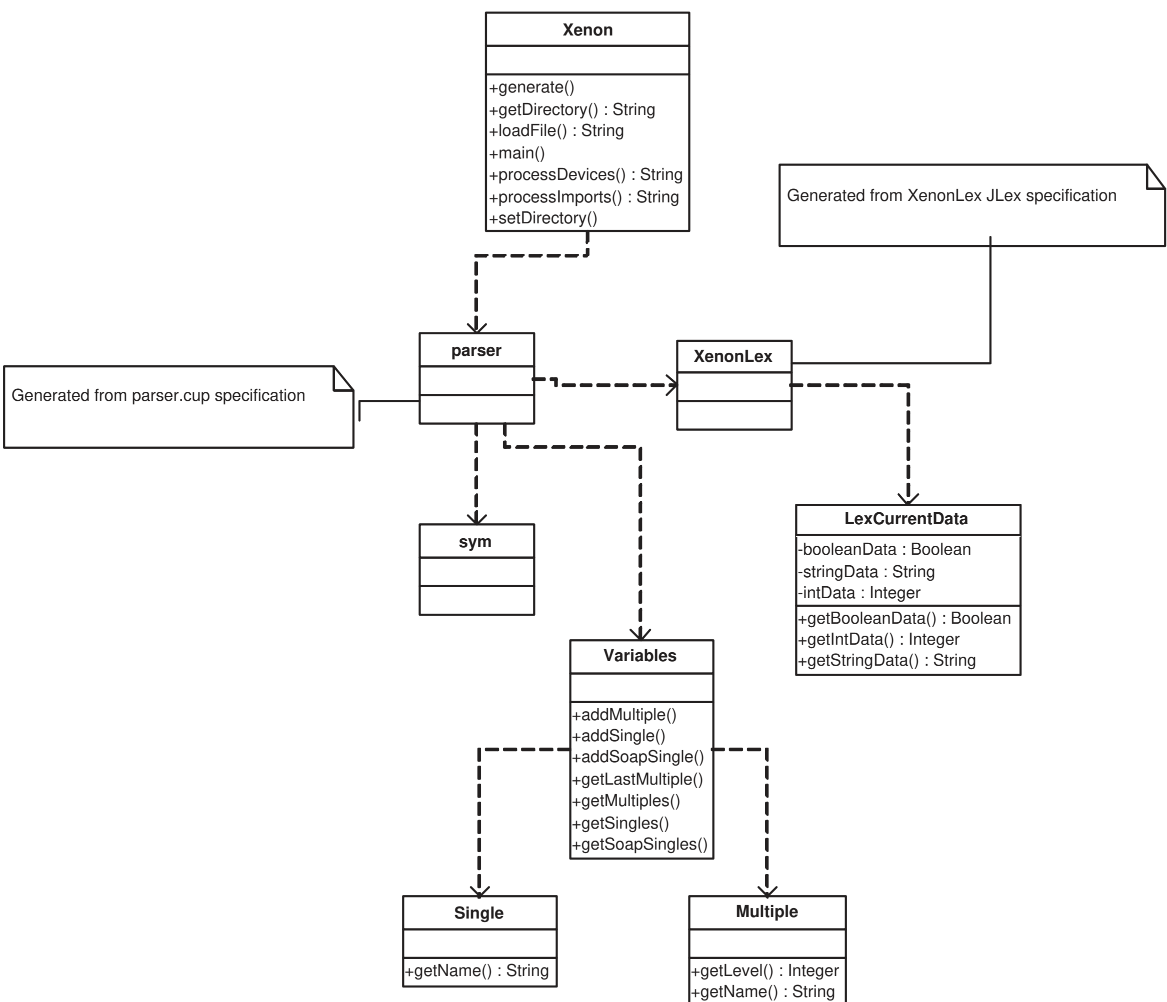


Figure 5.4: UML class diagram describing the architecture of the MyXML compiler

The MyXML compiler reference implementation has been written in Java (JDK Version 1.2). The compiler first uses the Apache Xalan [Apa01a] XSL processor and the Apache

Xerces [Apa01b] XML parser to parse MyXML documents (i.e., XML content and MyXML elements) and to add a layout to them. The resulting documents are then processed by the JLex lexical analyzer [Ber01] and the JCup code generator [Ani01] and the embedded MyXML elements are interpreted and resolved.

Figure 5.4 shows the architecture of the MyXML compiler with simplified UML class diagram. The classes *parser* (i.e., code generator), *sym* and *XenonLex* (i.e., lexical analyzer) are generated from JLex and JCup lexical analysis and grammar specifications and are used for content and code generation.

The class *Xenon* provides a command-line interface and an API to the compiler. The classes *XenonLexCurrentData*, *Variables*, *Single* and *Multiple* are used to pass information from the lexical analyzer to the code generator and to keep track of MyXML variables during the parsing.

5.3 Configurable device-independence components

There are three components in the MyXML tool suite that provide device-independence support: The *Dispatcher*, *Output* and *Collector* components. These components are implementations of the default device-independence run-time processors in the DIWE framework that were discussed in Chapter 4. The components are configurable and are instantiated and used at run-time in combination with the static and dynamic Web services generated by the MyXML compiler based on MyXML language specifications.

Table 5.3 shows the *Dispatcher*, *Output* and *Collector* device-independence components and lists the functionality each one provides.

Functionality/ Component	Dispatcher	Output	Collector
Device Detection	X		
Application Logic Interfacing		X	
Page Splitting			X
Process Partitioning			X

Table 5.3: Table showing the device-independence components and the functionality they provide

The *Dispatcher* component is responsible for device-detection and is a Java implementation of the *device detection* processor. It can be configured to detect the device a user is using based on the HTTP request header and respond accordingly.

The configurable *Output* component provides device-independent application logic interfacing support to the services generated by the MyXML compiler and is an implementation of the *logic interfacing* processor discussed in the previous chapter. It allows the application logic to be written once and used for multiple device-specific MyXML-generated Web services without any modifications.

The *Collector* component is one of the most important run-time tools in the MyXML tool suite. It provides layout adaptation support and is an implementation of the *page splitting* and *process partitioning* processors in the DIWE framework.

5.3.1 The Dispatcher component

The *Dispatcher* component detects devices by analyzing the *User-Agent* attribute that is sent by most clients (i.e., browsers) in the HTTP request header. This attribute provides information about the client the user is using to access the Web service such as its name and version number.

By maintaining a list of clients and URLs they should be "mapped" to (i.e., the appropriate response), the *Dispatcher* component allows two users on two different devices to access the *same* URL, but see two *differing* pages that have been custom-tailored for the device.

Detecting devices based on the *User-Agent* attribute is not a new idea. Other systems and programs have been using this attribute for various purposes (e.g., collecting statistics on browser usage) since the early days of the Web. One limitation of the approach is that not all clients may send the *User-Agent* attribute with HTTP requests. Therefore, the component allows the configuration of a default action if it cannot detect the client agent.

A second limitation of detecting devices based on the *User-Agent* HTTP attribute is that a list of known devices have to be maintained. If the user is using an unknown device that is not in the list (e.g., a new micro-browser for the Compaq iPAQ PDA), the *Dispatcher* will not be able to detect it. Nevertheless, by analyzing the Web access logs, it is possible to find out what devices users are using to access a particular service. The configuration of the *Dispatcher* component, thus, can be adjusted for each service.

5.3.1.1 Configuration grammar

The *Dispatcher* component provides an XML configuration language. Figure 5.5 depicts the DTD that defines the configuration grammer of the *Dispatcher* component.

A typical configuration consists of a list of user agents and a default agent in case there are no matches (see line 3). Each agent entry is accompanied by a name and a mapping URL (i.e., <name> and <map_to> elements – see lines 5-6 and 12-15). The name entry defines a string that should be matched to the contents of the *User-Agent* attribute in the HTTP request header.

The *Dispatcher* component can dispatch or redirect requests. Redirecting requests means that the *Dispatcher* component forwards the request to another URL via HTTP. Dispatching requests, on the other hand, means that the *Dispatcher* component *invokes* another component *internally* with the parameters it has received. Each agent entry in the configuration

```
1. <?xml encoding="UTF-8"?>
2.
3. <!ELEMENT agents ((agent)+,default>
4.
5. <!ELEMENT agent (name,map_to)>
6. <!ATTLIST agent action CDATA #IMPLIED>
7.
8. <!ELEMENT default (target,action)>
9. <!ELEMENT target (#PCDATA)>
10. <!ELEMENT action (#PCDATA)>
11.
12. <!ELEMENT name (#PCDATA)>
13.
14. <!ELEMENT map_to (#PCDATA)>
15. <!ATTLIST map_to static CDATA #IMPLIED>
```

Figure 5.5: The Dispatcher component configuration DTD

definition accepts an *action* attribute (see line 6). This attribute defines if the request should be dispatched or redirected (i.e., its value can be “dispatch” or “redirect”).

The <map_to> element accepts an attribute *static* (see line 15). The attribute signals the *Dispatcher* component that the service that is being configured is static. It is assumed per default that the service being configured is dynamic.

5.3.1.2 A configuration example

Imagine device detection support is needed for a service that is accessible at the URL *http://kirda.com/welcome/*. There are users that access the service with traditional desktop HTML browsers and users that access it using micro-browsers on PDAs.

The goal is that users on PDAs should automatically see the contents in the URL *http://kirda.com/welcome/pda/* and the desktop browser users should see the content at the URL *http://kirda.com/welcome/pc/*.

First the Web server has to be configured to divert any requests that come to the URL *http://kirda.com/welcome/* to the *Dispatcher* component. Web servers offer configuration facilities with which this is easily done. Then, based on the *User-Agent* attribute, the *Dispatcher* component has to be configured to dispatch the request to the device-specific URLs listed above.

Figure 5.6 depicts the XML *Dispatcher* component configuration for the service. The *action="dispatch"* attributes in the *agent* entries (see lines 4,9,14) signal to the *Dispatcher* component that it should dispatch a request instead of redirecting it. The *name* and *map_to* tags in the agent entries define the mapping between the name of a user agent (i.e., device) and the URL it should be mapped to (e.g., see lines 5-6). In the example, two user agents, *Windows CE* and *Palm*, are mapped to the */welcome/pda* URL (see lines 5-6 and 10-11). PDAs running the Windows CE and Palm operating systems usually send these strings in

```
1.    <?xml version="1.0" encoding="UTF-8"?>
2.    <!DOCTYPE agents SYSTEM "agents.dtd">
3.     <agents>
4.          <agent action="dispatch">
5.                 <name> Windows CE </name>
6.                 <map_to> /welcome/pda </map_to>
7.          </agent>
8.
9.          <agent action="dispatch">
10.               <name> Palm </name>
11.               <map_to> /welcome/pda </map_to>
12.         </agent>
13.
14.         <agent action="dispatch">
15.               <name> Mozilla </name>
16.               <map_to> /welcome/pc </map_to>
17.         </agent>
18.
19.         <default>
20.               <map_to> /welcome/pc </map_to>
21.         </default>
22.   </agents>
```

Figure 5.6: A Dispatcher configuration for a service

the HTTP requests they make. When the *Dispatcher* component receives an HTTP request header *User-Agent* attribute that contains these strings, it dispatches the request to the URL designated for the PDA.

In the example, the *Dispatcher* component detects Mozilla-based browsers and dispatches them to the */welcome/pc* URL (see lines 14-17). The default rule in this configuration is to dispatch all requests to the */welcome/pc* URL (see lines 19-21).

It is usually not necessary to configure a *Dispatcher* component for every single page in a service. The home page, for example, can act as an entry point into the device-specific pages.

5.3.1.3 Implementation

The *Dispatcher* component has been implemented as a stand-alone Java servlet and uses the Apache Xerces XML parser for processing configuration files. It is based on the Java Servlet API Version 2.3 and has been tested with the Tomcat Servlet Engine version 4.0 (Catalina).

The *Dispatcher* class is instantiated and invoked by the servlet engine (i.e., Web server). The *RequestWrapper* class is used in the Java Servlet API Version 2.3 to wrap and modify/extend an incoming HTTP request. It is usually used in request dispatching. The *ParseErrorHandler* class is used by the Xerces XML parser to process errors that are encountered during the parsing.

Figure 5.7 depicts a UML class diagram that describes the architecture of the *Dispatcher* component.

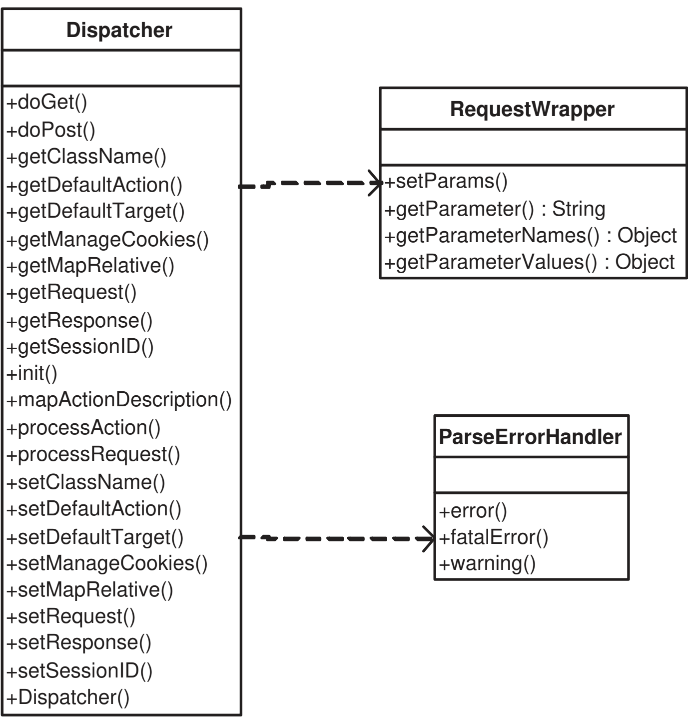


Figure 5.7: UML class diagram showing the architecture of the Dispatcher component

5.3.2 The Collector component

The *Collector* component in the MyXML tool suite is a configurable, stand-alone application that provides both the *page splitting* and *process partitioning* processor functionality discussed in the previous chapter. It is responsible for giving out the information partially over many smaller steps, keeps track of the group and subgroup numbers and can receive commands on which splits (i.e., layout fragments) to give out. Furthermore, it invokes the application logic with the input data it has collected when all necessary data has been submitted.

5.3.2.1 Configuration grammer

The *Collector* component provides an XML configuration language that allows the Web developer to define page splitting stepping values and *content types* for devices. A Web device requests information from the Web server with a specific content type HTTP attribute. A WAP phone, for example, signals the Web server with the content type *vnd.wap.wml* that it is awaiting a WML page.

Figure 5.8 depicts the DTD that defines the configuration grammer of the *Collector* component.

A typical configuration consists of a list of device names and the corresponding content type definitions and stepping values. There is also default device definition (see line 3).

The XML definition contains a list of <device> elements with <contentType> and <steps> elements (see line 5). Each device name is mapped to a content type definition

```
1. <?xml encoding="UTF-8"?>
2.
3. <!ELEMENT config ((device)+,default>
4.
5. <!ELEMENT device (name,contentType,steps)>
6.
8. <!ELEMENT default (contentType,steps)>
9.
10. <!ELEMENT contentType (#PCDATA)>
11.
12. <!ELEMENT steps (#PCDATA)>
```

Figure 5.8: The Collector component configuration DTD

and a stepping value.

5.3.2.2 A configuration example

```
1.    <?xml version="1.0" ?>
2.    <config>
3.
4.    <device>
5.       <name> pda </name>
6.       <contentType> text/html </contentType>
7.        <steps> 3 </steps>
9    </device>
10.   <device>
11.         <name> wap </name>
12.         <contentType> text/vnd.wap.wml </contentType>
13.         <steps> 3 </steps>
15.    </device>
16.   <default>
17.         <contentType> text/html </contentType>
18.         <steps> </steps>
19.    </default>
20.   </config>
```

Figure 5.9: A typical XML Collector component configuration

Figure 5.9 shows the *Collector* component configuration file for a Web service. The content type for the default device family is HTML (i.e., *text/html*, see line 17) and a stepping value is not given (i.e., no page splitting or process partitioning is required).

Two other devices, PDAs and WAP phones, are also supported. The content type definition for PDA devices is HTML (i.e., *text/html*, see line 6) and WML for WAP phones (i.e., *vnd.wap.wml*, see line 12). Both devices use a stepping value of 3.

5.3.2.3 Implementation

The *Collector* component has been implemented as a stand-alone Java servlet. It is uses the request dispatching and session management feature of the Java Servlet API Version 2.3. The component has been tested with the Tomcat Servlet Engine version 4.0 (Catalina).

Figure 5.10 shows a UML class diagram describing the architecture of the *Collector* component. The class *Collector* processes the result stream that the calling component passes to it. In a typical setting, the *Dispatcher* component instantiates and invokes this main class. The class *CollectorStore* is used to keep track of group and subgroup numbers and the target URLs for process partitioning. The class *ParameterStore* is used to keep track of all CGI parameters that the *Collector* component receives so that they can be forwarded to the target URL when the collection is finished.

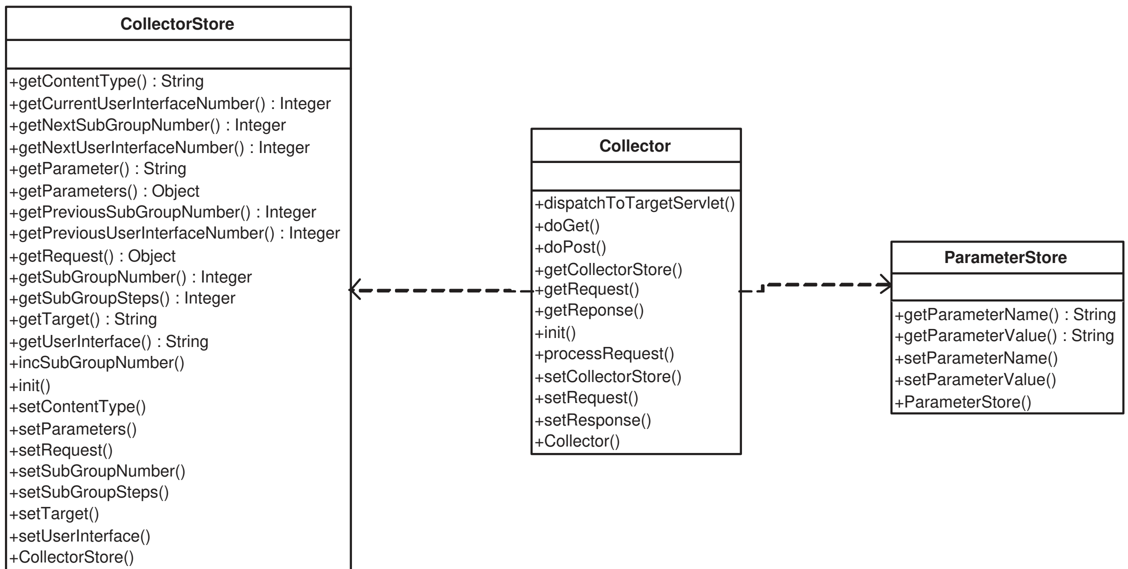


Figure 5.10: UML class diagram describing the architecture of the Collector Component

5.3.3 The Output component

The *Output* component in the MyXML tool suite is a stand-alone application that provides the functionality of the *logic interfacing* processor in the DIWE framework. Just like the other two device-independence components, it can be configured to adjust its behavior.

5.3.3.1 Configuration grammer

The *Output* component provides an XML configuration language that allows the Web developer to specify how the component should deal with the output that it receives from the layout/content classes. Figure 5.11 depicts the DTD that defines the configuration grammer of the *Output* component.

```
1. <?xml encoding="UTF-8"?>
2.
3. <!ELEMENT config ((device)+,default>
4.
5. <!ELEMENT device (name,processor)>
6.
7. <!ELEMENT default (#PCDATA)>
```

Figure 5.11: The Output component configuration DTD

A typical configuration consists of a list of device names and the appropriate processor that the *Output* component should invoke with its output. Usually, the *Output* component will invoke the *Collector* component with an output stream that should be processed for page splitting and process partitioning. However, the configuration mechanism of the *Output* component provides flexibility and allows other processors to be invoked as well. For example, an output stream for a device could be sent to a specific Java servlet developed by the Web developer for creating and saving PDF files.

Figure 5.11 shows the *Output* component configuration DTD. The XML definition contains a list of <device> elements with <name> and <processor> elements (see line 5). Each device name is mapped to a processor available at a specific URL. Furthermore, a default processor is also given for the default device family using the <default> element (see lines 3 and 7).

5.3.3.2 A configuration example

```
1.    <?xml version="1.0" ?>
2.    <config>
3.
4.    <device>
5.       <name> pda </name>
6.        <processor> /collector </processor>
7    </device>
8.    <device>
9.          <name> wap </name>
10.          <processor> /collector </processor>
11.    </device>
12.   <device>
13.         <name> pdf </name>
14.          <processor> /pdfgenerator </processor>
15.    </device>
16.   <default>
17.         /collector
18.    </default>
19.   </config>
```

Figure 5.12: A typical XML Output component configuration

Figure 5.12 shows the *Output* component configuration file for a Web service. The processor for the default device family is the *Collector* component available at the URL */collector* (see lines 16-18). The device families *pda* and *wap* have been configured to use the *Collector* component as well (see lines 4-7 and 8-11). The device family *pdf* has been configured in this case to use a processor available at the URL */pdfgenerator*. The Web developer has written this processor herself.

5.3.3.3 Implementation

The *Output* component has been implemented as a simple Java class and uses the Apache Xerces XML parser for processing configuration files. It uses the *Reflection* mechanism of Java to create instances of layout/content classes from their class names.

Figure 5.13 shows the UML class diagram of the *Output* component. The application logic creates an instance of the *Output* Java class. Errors in the configuration files are processed with the *ParseErrorHandler* class.

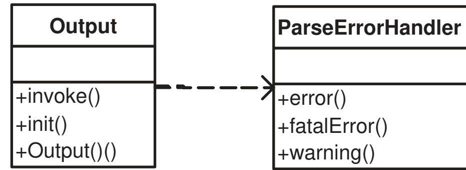


Figure 5.13: UML class diagram of the Output component

5.4 MyXMLDesigner

A user-friendly visual development environment is important for device-independent Web engineering because of the increased complexity of Web sites that are built with XML and XSL. The Web site planning, organization and maintenance overhead may increase significantly with the use of XML and XSL technologies [KKJK01]. Web sites may become even more complex when application logic separation support is also provided and separate layouts have to be managed for different Web devices. The MyXMLDesigner visual development environment attacks this problem and aims to ease device-independent Web site development and maintenance.

Compared to other visual Web site development tools and environments, one of MyXML-Designer's distinguishing features is its editing support for the separation of layout, content and logic during the implementation. Furthermore, MyXMLDesigner is one of the few visual development environments that aims to support the construction and maintenance of device-independent Web sites. It provides a user-friendly interface to the MyXML compiler and the device-independence components in the MyXML tool suite.

MyXMLDesigner provides the following functionality to Web developers:

- *Customizable, XML-based menus* for layout, content and logic separation, and page splitting and process partitioning support.

- *Configuration mechanisms* that allow the definition and management of devices and the configuration of the *Dispatcher*, *Collector* and *Output* device-independence components.
- *User-friendly code editing facilities* with syntax highlighting for MyXML, XML, XSL and Java documents.
- *Creation, organization and management* of Web projects and project files.
- *Visual definition and management* of Web pages that support multiple layouts.
- *Generation, deployment and compilation* of static and dynamic content (using the MyXML compiler).

5.4.1 Overview of the IDE

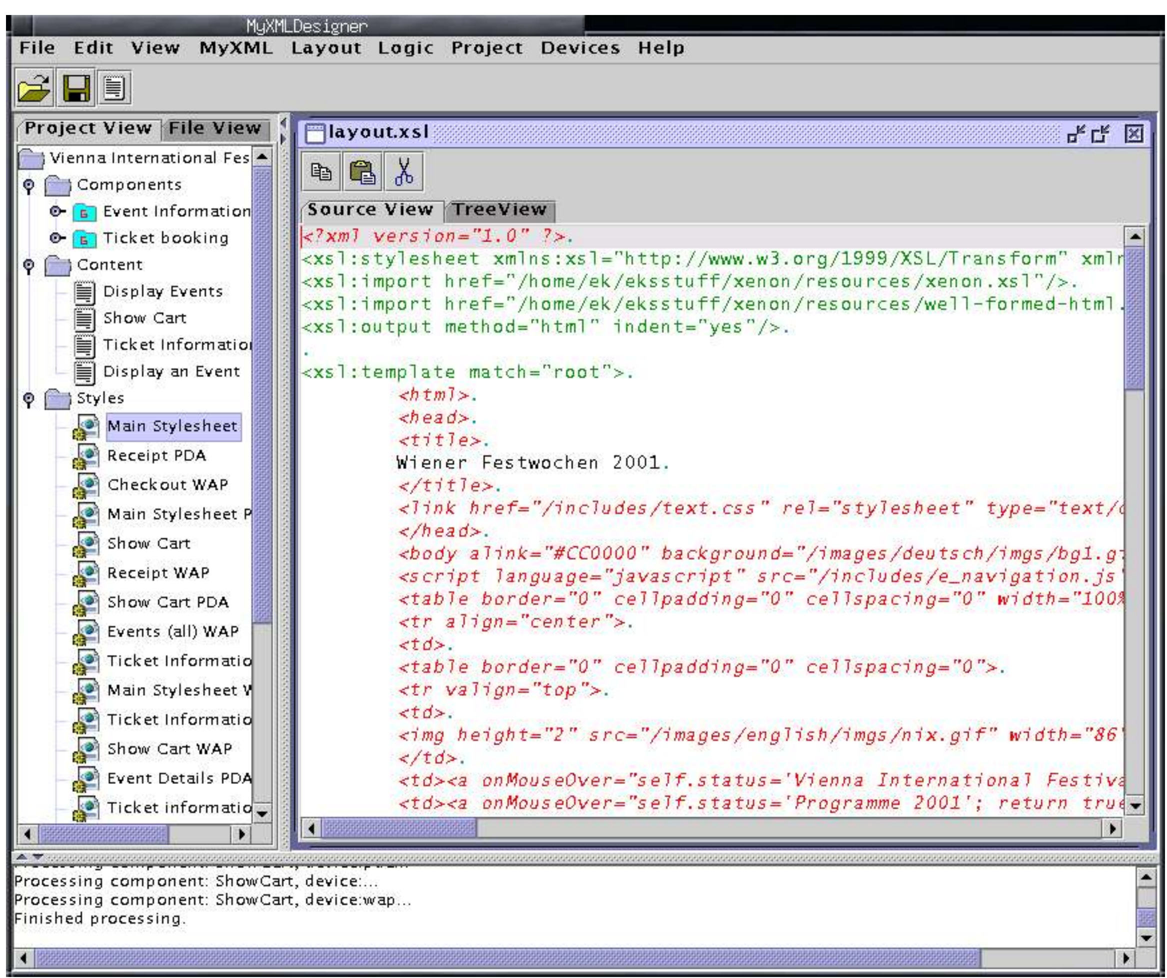


Figure 5.14: The MyXMLDesigner visual Integrated Development Environment (IDE)

The MyXMLDesigner IDE contains of a desktop that is able to display multiple documents. Figure 5.14 presents a screenshot of the application. The *MyXML*, *Layout* and *Logic*

menu items in the main menu are customizable by the user. To add and manage devices, the user chooses the *Devices* menu item. The *Project* menu contains items that allow the generation of pages, compilation of sources, managing of projects and configuration settings (e.g., setting the Java CLASSPATH Environment).

A message pane is embedded into the bottom of the desktop pane that displays system messages. In the screenshot, for example, the messages indicate that the user (i.e., Web developer) has processed and generated pages for the project using the MyXML compiler.

The project view on the left side of the desktop supports two views: the project view and the file view: The project view provides a collapsible tree view of the MyXML documents and XSL layout definitions in the project, the pages in the Web site and the devices each page supports. The file view provides a collapsible tree view of the files in the project and their physical locations.

By clicking on the nodes of the collapsible tree, page properties can be displayed, new pages can be created by visually combining MyXML documents and XSL definitions, and the contents of the files in a project can be opened as documents in the desktop. In the screenshot, for example, an XSL stylesheet *layout.xsl* has been opened and is being edited.

5.4.2 Support for design

One important feature of MyXMLDesigner is its support for data organization. Data organization is an old issue in Web site design. A frequent problem is that as the site grows, content managers lose track of the files and resources in the site. The results are often *broken (or dangling) links*, a growing need for extra storage space and files that are "forgotten" [KKJK01, RM98]. If XML/XSL technologies are deployed, data organization problems may worsen because the number of involved files and their dependencies increases. In a typical site, for example, an XML file may reference a DTD, import other XML files and point to a stylesheet that, yet again, imports other stylesheets. Data organization planning also includes writing makefiles and scripts that allow the easy compilation of sources and copying of files.

MyXMLDesigner decreases the data organization planning and management effort by automatically creating content, layout and source code directories and generating makefiles. Static and dynamic content can then be generated and deployed by calling these makefiles. Files that are being inserted into the project, as well as new content, layout and application logic files that are created are automatically stored in the corresponding locations.

In MyXMLDesigner, a *project* is the highest organizational unit. Web sites and Web services are treated as projects in MyXMLDesigner. The project in MyXMLDesigner defines the content, layout and application logic resources that are available and the necessary settings for the development environment such as the location of the deployment directories.

A Web site can constitute a single project in MyXMLDesigner. From a management and organization point of view, it is more practical to structure Web sites as a combination of separate projects. For example, a main project can be created that defines the main layout infrastructure and content in the Web site and other projects can then import and extend this functionality.

5.4.3 Support for implementation

HTML editors are quite popular in Web development. They often provide syntax highlighting for editing HTML content. One common feature of such editors is their ability to generate HTML elements: The developer can choose HTML layout elements such as <br> and <table> from a menu that are then inserted into the HTML document that is being edited. The customizable *development menus* in MyXMLDesigner are similar. The menus allow Web developers to encode layout, content and application logic-specific elements and code into MyXML, XSL and Java documents.

The advantage of having customizable development menus is that the Web developer can extend them to contain device and problem-specific code. The layout definition menu supports HTML, WML and MyXML layout code by default, but the Web developer, for example, can add VoiceXML elements to it simply by extending the XML menu definition.

Contrary to other Web site construction tools that intermix the layout, content and application logic information, MyXMLDesigner guides the Web development during the implementation and supports the layout, content and logic separation by enabling and disabling menu items. For example, when the Web developer is editing an XSL layout file, menu elements from the MyXML Namespace are disabled and cannot be automatically inserted into the document.

In real-world projects, it is sometimes necessary to mix layout and content to some degree (e.g., when embedding links). The separation mechanism, does not prevent the Web developer in inserting elements manually. It merely encourages the separation and provides some guidance.

MyXMLDesigner provides syntax highlighting and editing support for pure text, XML, XSL, MyXML and Java code files. By displaying the file contents in a combination of colors, the Web developer can distinguish between MyXML, general XML and XSL elements and identify the layout, content and logic during the development.

5.4.4 Support for configuration and deployment

Figure 5.15 shows a screenshot of MyXMLDesigner's device configuration dialogs. MyXMLDesigner provides a graphical user interface for the configuration of the *Dispacher*, *Collector* and *Output* device-independence components. Device families and their properties can be easily configured and managed without the need to edit the XML configuration files by hand. In the screenshot, for example, the Web service (the Vienna International Festival e-commerce component in this case) has been configured to support 5 device families: PDAs (device name *pda*), PDF generation (device name *fop*), speech-recognition interface using VoiceXML (device name *voice*) and WAP access (device name *wap*).

In the screenshot, the properties of WAP devices are currently being edited. The splitting step has been set to 3 and the *Collector* component available at the URL */collector* has been selected as the processor for the device.

Whenever a Web site is generated, MyXMLDesigner automatically instantiates, configures and deploys the device-independence components on the Web server based on the project settings.

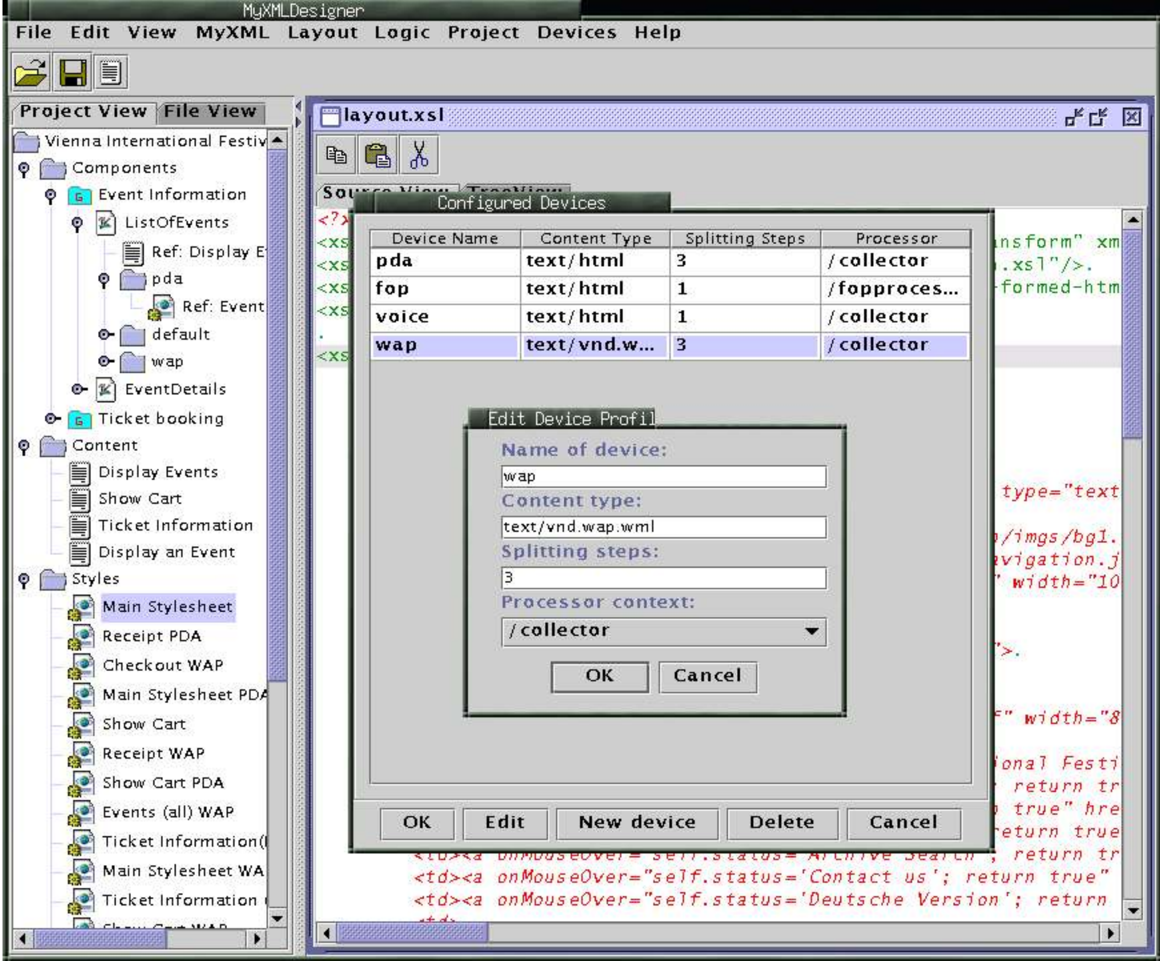


Figure 5.15: Configuring general device properties

5.4.5 Support for Web page creation and maintenance

Compared to other visual Web tools, one of the main distinguishing features of MyXMLDesigner is its support for device-independent Web page creation and management. The Web developer can add device-specific layouts to pages and multi-device support is part of the page creation and management process.

MyXMLDesigner provides visual mechanisms for:

- Listing the pages that constitute a service or a site (i.e., site overview).
- Displaying information about each device a page supports.
- Grouping of pages to ease organization and management.
- Displaying which MyXML documents and XSL stylesheets each page uses.

A page is created with a dialog that allows the Web developer to enter descriptive information about the page such as its name and purpose. The user is then presented a *page*

property dialog that displays the MyXML documents and layout stylesheets in the project. The minimum setting needed to create a page is to choose a MyXML document and a layout stylesheet for the default device family.

The developer can later choose a page and add arbitrary numbers of devices to it by simply specifying the stylesheet in the project that applies the suitable layout for the device (i.e., either a new stylesheet or an existing one that uses XSL stylesheet pre-processing).

The devices each page supports are listed in a collapsible tree in the project view. In the screenshot in Figure 5.15, for example, the project panel contains a page *ListOfEvents* that supports the PDA, WAP and HTML device families. By expanding each device node in the tree, the layout components that they support become visible.

5.4.6 Architecture and implementation

MyXMLDesigner has been implemented in Java and uses the Swing Graphical User Interface (GUI) classes. It accesses and uses the MyXML compiler using the processor's API. The device-independence components are stored and managed in a repository on the local file system.

MyXMLDesigner generates XML makefiles that can be processed by the Apache Jakarta Ant [ant02] tool. Furthermore, it uses the Ant libraries to compile Java sources generated by the MyXML compiler.

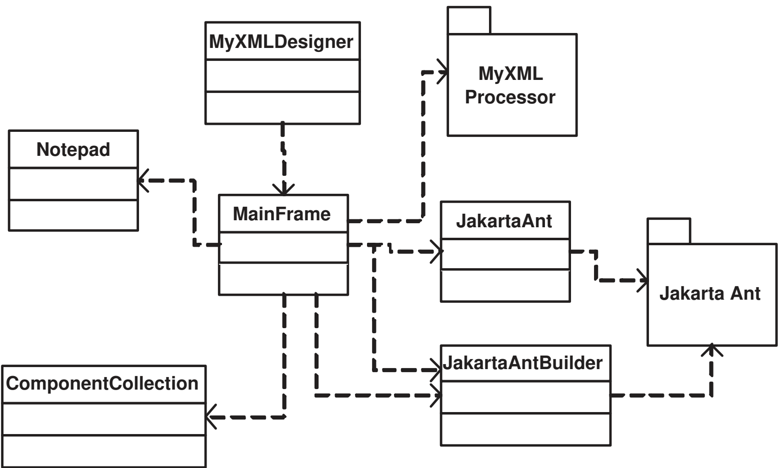


Figure 5.16: Simplified UML class diagram describing the architecture of MyXMLDesigner

Figure 5.16 shows a simplified UML class diagram describing the architecture of the MyXMLDesigner IDE. The *MyXMLDesigner* class is the main class of the application and creates the desktop with the *MainFrame* class. MyXML, XML and Java documents are opened in the desktop using the *Notepad* class. The device-independence components are stored and accessed using the *ComponentCollection* class. The IDE imports the MyXML compiler and Jakarta Ant packages. The MyXML compiler is directly accessed using its API in the *Xenon* class (see Section 5.2). The Jakarta Ant libraries are accessed using the

JakartaAnt class (for makefile generation) and the *JakartaAntBuilder* (for source code compilation).

5.5 Summary

This chapter presented the MyXML tool suite, an implementation of the Device-Independent Web Engineering (DIWE) framework discussed in the previous chapter. The tool suite consists of the MyXML compiler, three configurable run-time device-independence components and a visual Integrated Development Environment (IDE). The set of tools in the suite provide support for the design, implementation, deployment and maintenance of device-independent Web sites.

Chapter 6

Case Study: VIF e-Commerce Web service

The two previous chapters discussed the conceptual details of the Device-Independent Web Engineering (DIWE) framework and presented the technical details of its prototype implementation; the MyXML tool suite.

To evaluate the DIWE framework and its concepts of LCL separation, page splitting, process partitioning and XSL pre-processing, the MyXML tool suite was used to design, implement and extend a device-independent version of the *online ticket booking and ordering Web service* of the Vienna International Festival (VIF) Web site.

The Web service supports a default full-fledged HTML layout for traditional Web browsers on medium to large displays, a simpler HTML layout for PDA micro-browsers and small displays, and a WAP-layout for WAP-enabled mobile phones. Furthermore, after the user has completed an order, she has the possibility of downloading the receipt as a PDF file. The PDF information is generated dynamically and is treated as an additional device layout that the developer can add to an existing service.

The case-study Web service demonstrates that the devices a Web service will have to support in the near future might not only have varying display sizes and technical capabilities, but may also use different Web formats (e.g., WML for WAP, XSL:FOP for PDF, etc.). It shows that devices supported by a Web service *do not* necessarily have to be mobile or *computing* devices (e.g., PDF file generation).

The next sections give an overview of the Vienna International Festival (VIF) Web site, the functionality of the VIF e-commerce Web service and the device-independent implementation of the service with the MyXML tool suite.

6.1 The Vienna International Festival (VIF) Web site

The Vienna International Festival (*Wiener Festwochen*) is the major cultural event in Vienna. Visitors from around the globe come to Vienna during the festival. The festivities take place in various famous theater locations and concert halls. The annual festival, which usually lasts five weeks, presents operas, plays, lectures, concerts, musicals and exhibitions, featuring and hosting eminent international directors, performers and ensembles.

6.1.1 Service overview

The VIF Web site provides an extensive range of services to the visitors. Event locations, information on the current programme, an archive on former performances since 1995, on-line ticket service, as well as maps of major stages and venues are just some of the service offerings by the site. All of the information and interactive services are designed bilingually, in English and German, with the potential for extending the service to integrate other languages.

The number of services offered vary each year. The received user input and collected site statistics are analyzed annually, and the services offered, including the look-and-feel of the site, are tuned accordingly. These modifications can be anything from minor changes to significant transformations with major implications on the provided services.

6.1.2 Main VIF components

The festival programme, the archive system and the ticket reservation service are the main components of the VIF Web site. Additionally, services are offered that inform the user on stage highlights, press coverage, site news and some text translations of musicals and stage performances. The site visitor is able, anytime, to switch between German and English versions of the offered information.

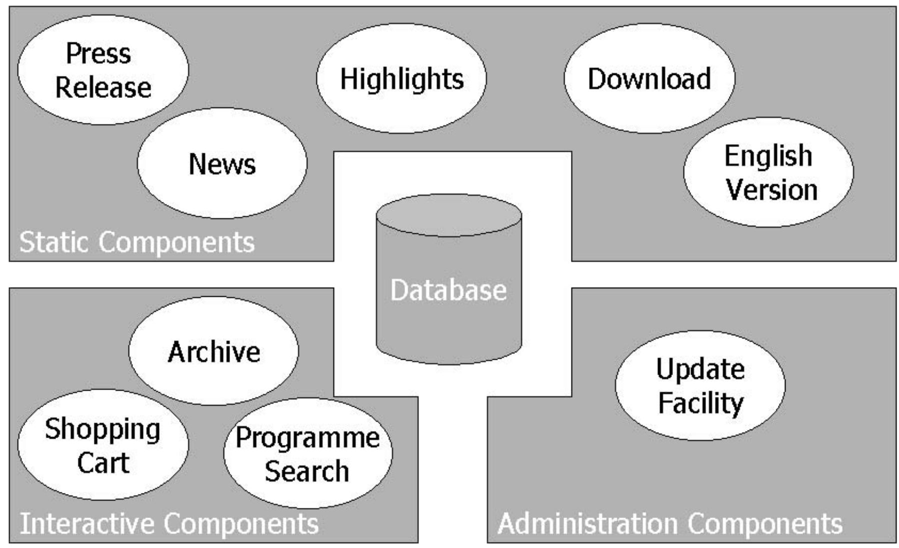


Figure 6.1: Main VIF Components in 2000

All of the site is indexed and coupled with a search engine. The user can search extensively in the archive and the current programme for specific locations, performances and events.

The programme information and the ticket management data are stored in an external

data source: a relational DBMS. The DBMS is used to manage all performance and event-related information.

The programme information is very dynamic and changes occur frequently. Online Web forms enable the content managers to modify the information in the DBMS. The information stored in the DBMS is retrieved every night and static HTML pages are compiled from it. Figure 6.1 depicts the various VIF components.

6.2 VIF e-commerce Web service

The VIF e-commerce Web service allows users to browse through cultural events such as operas and theater performances in the festival, retrieve detailed information about them and order tickets online.

The application is backed by a MySQL database (version 3.22.32). The layout and look-and-feel of the e-commerce Web service change every year. The general information structure and the way it is presented to the user is usually the same.

The graphical look of the site has also shown similarities in the last couple of years. There is a header on each page that contains logos and a navigation bar that allows the user to jump to different sections of the site. Sponsor logos are usually placed at the bottom and sides of pages.

This information is presented to the user over a number of pages: the programme (i.e., overview of events), detailed event information, ticket information, the shopping cart and the order form. The information in the database in the case study is from the 2001 festival.

6.2.1 The programme

The programme page gives an overview of the events in the festival for the specific season. There are about 30-40 events that are displayed in a clickable list. By clicking on an event, the user is taken to a page that provides more in-depth information about the event.

The typical HTML implementation of the festival programme displays all of the events in a single page. The user needs to vertically scroll to get an overview of all the events. This scrolling is acceptable as the number of events is low.

6.2.2 Detailed event information

In each detailed event page, the user can retrieve information about the event such as its language, short and long descriptions, dates and times, length, performers, authors and directors. Typically, some events also provide an introductory image.

After looking at the details of an event, the user can either go back to the programme overview, or can decide to book tickets for the event.

By clicking on a button (i.e., image) that is designated for ticket reservation, the user is taken to a page that displays ticket booking information for the event.

6.2.3 Ticket availability, date and price information

The ticket reservation page provides price and booking information about the available tickets for an event.

The page displays a list of dates, times and locations that show where and when the event is performed. By typing in the number of tickets into a corresponding input field, the user is able to specify the number of seats she would like to book for a specific performance. Once she has booked the tickets, her order is placed in a virtual shopping cart.

If the tickets for a specific performance are sold out, the input field for that date is replaced by an image that indicates that no more tickets are available.

The number of performances of a single event usually vary. There is often one performance per day and events may be performed for up to seven days. The entire information is displayed on a single page.

There are four different price categories for the tickets: A,B,C,D – A being the most expensive. The prices per category change from event to event and are listed with the performance dates, times and locations.

6.2.4 The shopping cart

Whenever the user books a ticket for an event, she is taken to a page that displays the contents of her virtual shopping cart.

The user is shown a list of tickets she has booked, the dates, times and locations of the performances, the prices of the tickets and the total sum she has to pay for the tickets if she decides to confirm and go ahead with the booking.

At the shopping cart page, the user can choose to go back to the programme page to browse information about other events and to book more tickets. She can also decide to complete the order (i.e., check out) by providing the necessary purchase information such as her name and credit card number.

6.2.5 Completing the order (checking out)

Once the user decides to go ahead with the purchase and buy the tickets she has booked, she is taken to a page that displays an order form.

The page presents a list of tickets she has booked with the corresponding dates, times and locations. The total sum that she has to pay for the tickets is displayed.

If there are any errors in the bookings, the user has a final chance to go back and make modifications. Otherwise, by entering the necessary purchasing information such as her name, address, credit card number and e-mail address, she confirms the bookings she has made and the order is sent to the festival organization.

The user is displayed a finishing page that thanks her for the purchase. It serves as a receipt for the purchase.

6.3 Implementation with the MyXML tool suite

Using the MyXML tool suite, a device-independent version of the VIF e-commerce Web service was designed and implemented. The constructed e-commerce Web service was extensible and able to support different devices with the same application logic.

The engineering of the case study covered four phases: Design, implementation, deployment and maintenance. In all engineering phases of the case study, the MyXMLDesigner visual IDE was used.

The target development and deployment environment for the case study consisted of the Java Development Kit (JDK) version 1.2 and the Apache Tomcat servlet engine version 4.0. MM_MYSQL version 1.1b was used as the JDBC driver for the MySQL relational database.

Some extra libraries were also needed for implementing the application logic. The following libraries were used: PerlTools version 1.2.0a, Apache FOP toolkit version 0.20.2, Apache Xerces XML parser version 1.4.0 and the Apache Xalan XSL processor version 2.2.D6.

6.3.1 Design

The design phase consisted of five stages: device identification, data organization planning, content definition and XSL stylesheet definition.

6.3.1.1 Device identification

The default device family for the VIF e-commerce Web service was identified as being the traditional HTML access that the VIF had been supporting since 1995. The default family was to provide full support to all the functionality.

It was also decided to provide full functionality and service support to PDA devices. The total provided information, however, would be less. The detailed event pages, for example, would not present long descriptions of events because of the smaller display sizes. The user would be able to access all the pages with a PDA and book and purchase tickets online with a custom-tailored layout. This layout would be a simpler HTML layout that would not contain as many images and tables as the default family HTML layout.

The objective was to initially provide support for the default and PDA device families and to add additional devices during the maintenance phase.

6.3.1.2 Data organization planning

MyXMLDesigner provided support for the data organization and planning of the case-study. A new project was created for the e-commerce Web service and a development directory structure was created automatically. Build files (i.e., makefiles) were also generated that enabled command line compilation and generation outside of MyXMLDesigner.

6.3.1.3 Content definition

The content definition identified which MyXML elements would be necessary to select the content from the relational database and how the XML content should be structured.

Page	MyXML Functionality
Programme	<myxml:sql> (<myxml:dbcommand>, <myxml:dbitem>)
Event details	<myxml:sql> (<myxml:dbcommand>, <myxml:dbitem>),<myxml:cgi>
Ticket details	<myxml:single>,<myxml:multiple>
Shopping cart	<myxml:single>,<myxml:multiple>
Order form	<myxml:single>,<myxml:multiple>
Receipt	<myxml:single>,<myxml:multiple>

Table 6.1: Identification of MyXML dynamic content functionality on each page

Six different types of pages had to be constructed: The programme, event details, ticket information, shopping cart, order form and a final receipt.

By analyzing the content provided in these pages, some commonalities were identified: The final page, for example, displayed the tickets the user had ordered and its contents overlapped with the contents of the shopping cart page. The order form also displayed the contents of the user's shopping cart and there was again a commonality with the shopping cart page.

In the pages that had to be constructed, the content often had to be retrieved from the database. <myxml:sql> elements were necessary to retrieve the contents from the database and in some of the pages, there was also a need for <myxml:single> and <myxml:multiple> elements for dynamic content.

Table 6.1 presents the list of pages in the case study and the MyXML elements that they use. The programme page uses a <myxml:sql> command to select all the event titles from the database. The detailed event pages are constructed by passing a CGI database ID parameter to the service (using <myxml:cgi>) with which the necessary event details are retrieved (using <myxml:sql> again). Pages such as the shopping cart, on the other hand,

receive dynamic content from the application logic directly. A <myxml:single> element, for example, is used that contains the total number of tickets the user has ordered. The total sum is calculated in the application logic.

The content is structured in four different MyXML documents: *Show events*, *Show event details*, *Shopping cart* and *Ticket information*. Two pages, the receipt and the order form, reuse existing content definitions. The description granularity was kept as high as possible.

Appendix A lists the content definition for the shopping cart and the order form.

6.3.1.4 XSL stylesheet definition

During the XSL stylesheet definition, the default HTML layout was analyzed and commonalities were identified such as header, footer and navigational constructs. The main aim in defining the XSL stylesheets was to keep the number of stylesheets needed to generate the pages as small as possible.

The XSL stylesheets for the pages were defined incrementally: First, stylesheets were written that generated the common layout elements and that were to be imported by the rest. Then, XSL stylesheets were built that displayed simple HTML pages (i.e., without icons, logos, pictures, etc.) that implemented the functionality and that were used for testing.

The XSL stylesheet infrastructure that had been defined was then extended and adapted to the graphical look of the default HTML layout: Headers, navigational constructs, icons, images and the correct fonts were added.

Appendix A lists the XSL default device family layout definition for the shopping cart.

6.3.2 Implementation

6.3.2.1 Construction of the pages

The application logic was created traditionally using servlet session management to keep track of the tickets the user had booked. The logic accessed the database to check for ticket availability and to build the dynamic content accordingly. Based on the discussion in Chapter 5, it used the *Output* component to pass the dynamic content to the layout/content code by using string variables and arrays. Appendix A lists the Java application logic for the shopping cart servlet.

MyXMLDesigner's page creation and management functionality was used to construct pages by choosing MyXML documents and XSL layout files.

Figure 6.2 shows a screenshot of the project pane in MyXMLDesigner for the VIF e-commerce Web service. Two groups have been defined to organize the pages: *Event information* and *Ticket booking*. The following pages have been defined: *ListOfEvents* (i.e., the programme page), *EventDetails* (i.e., the detailed event information), *TicketDetails* (i.e., the ticket information), *ShowCart* (i.e., the shopping cart), *Checkout* (i.e., the order form) and *Receipt* (the receipt page).

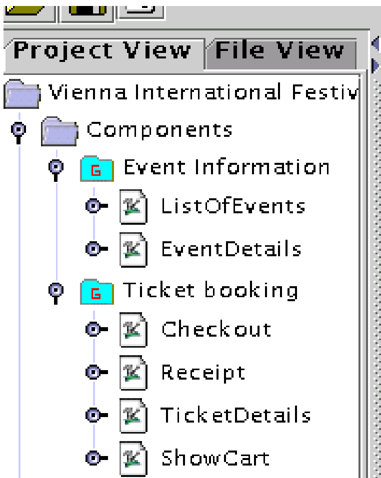


Figure 6.2: Screenshot of the project pane for the VIF project

6.3.2.2 Integration of PDA device family

After the default device family pages had been implemented, the aim now was to integrate a PDA layout that had been identified in the design phase.

Because of the smaller display sizes of PDAs, page splitting and process partitioning information was integrated into the existing stylesheets. The HTML pages for PDAs had less images and simpler tables and the selected content also varied. The detailed event page, for example, presented less information and omitted a long description of the event. PDA devices were added to the existing pages in MyXMLDesigner as discussed in the previous chapter.

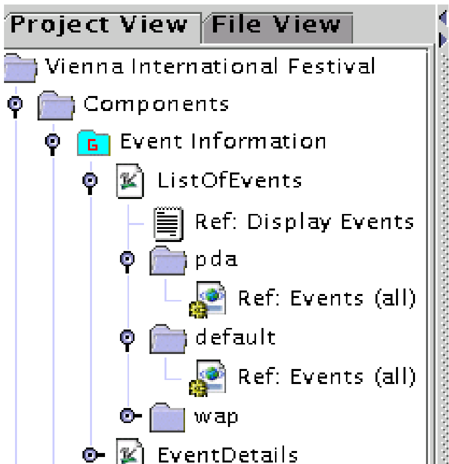


Figure 6.3: Adding the PDA layout to the Web service

Figure 6.3 presents a screenshot of the project pane for the case study that shows the PDA and default stylesheets the *ListOfEvents* page supports. This implementation uses the same stylesheets for the default and PDA layouts and makes use of XSL stylesheet pre-processing.

6.3.3 Deployment

Device name	Content type definition	Steps	Processor
default	text/html	none	/collector
pda	text/html	3	/collector
wap	text/vnd.wap.wml	2	/collector
pda	text	none	/fopprocessor

Table 6.2: Device configurations for the VIF case study

Four devices were configured for the service during deployment. Table 6.2 shows the device configurations for the VIF e-commerce Web service. A stepping value of 3 is used, for example, for the PDA device family.

The default family service was configured to be accessible via the URL */wf/displayevents*. The Dispatcher component detects the device the user is using (see discussion in the previous chapter) and dispatches the corresponding URL.

6.3.4 Maintenance

During the maintenance, it was decided that a WAP layout should be added to the e-commerce service. The WAP layout was to support full access to the service.

Page splitting and process partitioning had to be used again to provide WAP access support. In contrast to the default and PDA device families, no images were used for the WAP pages.

The existing service was extended by both adding new device stylesheets to the pages (where necessary) using MyXMLDesigner, and by extending the existing stylesheets using stylesheet pre-processing.

During the maintenance phase, it was also decided that the receipt page that the user sees at the end of a completed order should be downloadable as a PDF file. A PDF device family was added to the receipt page that generates XSL:FOP commands. The XSL:FOP information is sent to a FOP processor (i.e., via URL */fopprocessor*). The FOP processor then generates PDF information that is sent to the user's browser.

6.4 Usage scenarios

This section illustrates the usage of the device-independent VIF e-commerce Web service with three different devices. Screenshots from the Web service are presented.

6.4.1 Ordering a ticket using a traditional browser

Imagine Dr K is using a common PC browser, the Internet Explorer, and would like book and purchase a ticket. He is thinking about going to *Le Nozze di Figaro* when the festival starts a couple of months later.

He accesses the service and sees the complete list of events in the festival programme. Figure 6.4 shows the screenshot of the default device layout that Dr K sees.

He clicks on the link for *Le Nozze di Figaro* and is taken to a page that provides detailed information about the event. Figure 6.5 presents the screenshot of the default detailed event information page for *Le Nozze di Figaro*. Dr. K reads a description of the opera and decides that he would like to go. He clicks on an image for ticket reservation.

He sees a page that lists performance dates and locations for *Le Nozze di Figaro* (Figure 6.6). He decides to book and purchase one ticket for the 18th of June. He types in “1” in the input field for Category A (that may cost between 1800 and 2450 ATS depending on availability).

When Dr K submits the ticket booking form, he sees a page that shows the contents of his shopping cart. Figure 6.7 shows a screenshot of his shopping cart. He decides to go ahead with the booking and clicks an image to complete the order.

He is presented a page that displays the tickets he has reserved and a number of empty input fields prompting for information such as his name and credit card number (Figure 6.8). He fills in the information and confirms the order.

He sees a confirmation and receipt page (Figure 6.9). He clicks on a link at the bottom of the page and downloads his receipt as PDF.

6.4.2 Ordering a ticket using an iPAQ PDA

A few days later, Dr K is attending a meeting with his Compaq iPAQ Windows CE PDA. During a short break, he decides to book another ticket and accesses the VIF e-commerce application with his PDA.

He sees a page that fits his PDA display and that uses simple tables and small images for navigation. Figure 6.10(a) shows a screenshot of the programme page that Dr K sees. By pressing the *previous* and *next* buttons, he is able to see two event titles at a time (i.e., he is browsing through the page splits on the same page).

He clicks on *Intolleranza* and sees a new page that displays information about the event. Figure 6.10(b) shows a screenshot of the PDA *Intolleranza* information page. He sees that the event is in German and the music is by Luigi Nono. He has heard of him before and decides to buy a ticket.

Once Dr K clicks the ticket reservation button, he is presented a number of small pages with booking information (i.e., the ticket information page splits). He clicks through the pages by pressing the *next* button. Figures 6.10(c) and 6.10(d) present screenshots of the PDA page splits for the ticket availability and information page that Dr K is shown. He decides to book a ticket for the 15th of May. He enters the amount into the input field and submits the form.

Dr K is shown the contents of his shopping cart – again, over a number of smaller pages (i.e., Figure 6.11(a) depicts the first page split).

Dr. K clicks the link to complete the order and sees a final confirmation page that lists the tickets he is buying (i.e., Figure 6.11(b)). He continues by pressing the *next* button and is prompted for input over a number of smaller pages (i.e., page splits) where he enters information such as his name and address (i.e., Figures 6.11(b), 6.11(c) and 6.11(d)).

Finally, he sees a receipt page that confirms that his order has been successfully sent.

6.4.3 Ordering a ticket using a WAP phone

Dr K is waiting at an airport and is waiting for his flight to Chicago. He will be attending a conference there. He decides book another ticket for *Intolleranza* and invite somebody when he is back. He takes out his WAP phone and accesses the VIF service.

He is able to browse through the festival programme over a number of smaller WAP pages and sees two events per page (i.e., Figure 6.12(a) shows the first page split). He clicks on *Intolleranza* and is displayed a page that provides short information about the event such as its length and language (i.e., Figure 6.12(b)).

He clicks on ticket reservation and is presented a number of smaller pages that contain general ticket reservation information (i.e., ticket information page splits in Figures 6.12(c) and 6.12(d)).

He then chooses the 5th of May again and clicks a link to book a ticket for Category A (i.e., Figure 6.13(a)).

He is shown his shopping cart over a number of pages (i.e., page splits in Figures 6.13(b), 6.13(c) and 6.13(d)).

He clicks a link to complete the order and is taken to a final confirmation page. He enters information such as his name and address over a number of smaller pages and confirms the order (i.e., Figures 6.14(a), 6.14(b) and 6.14(c)).

Finally, he sees a page that confirms that his order has been sent successfully (i.e., Figure 6.14(d)).

6.5 Summary

This chapter presented the case study Vienna International Festival (VIF) Web site. It described the functionality of the VIF e-commerce Web service and the device-independent implementation of the service with the MyXML tool suite.

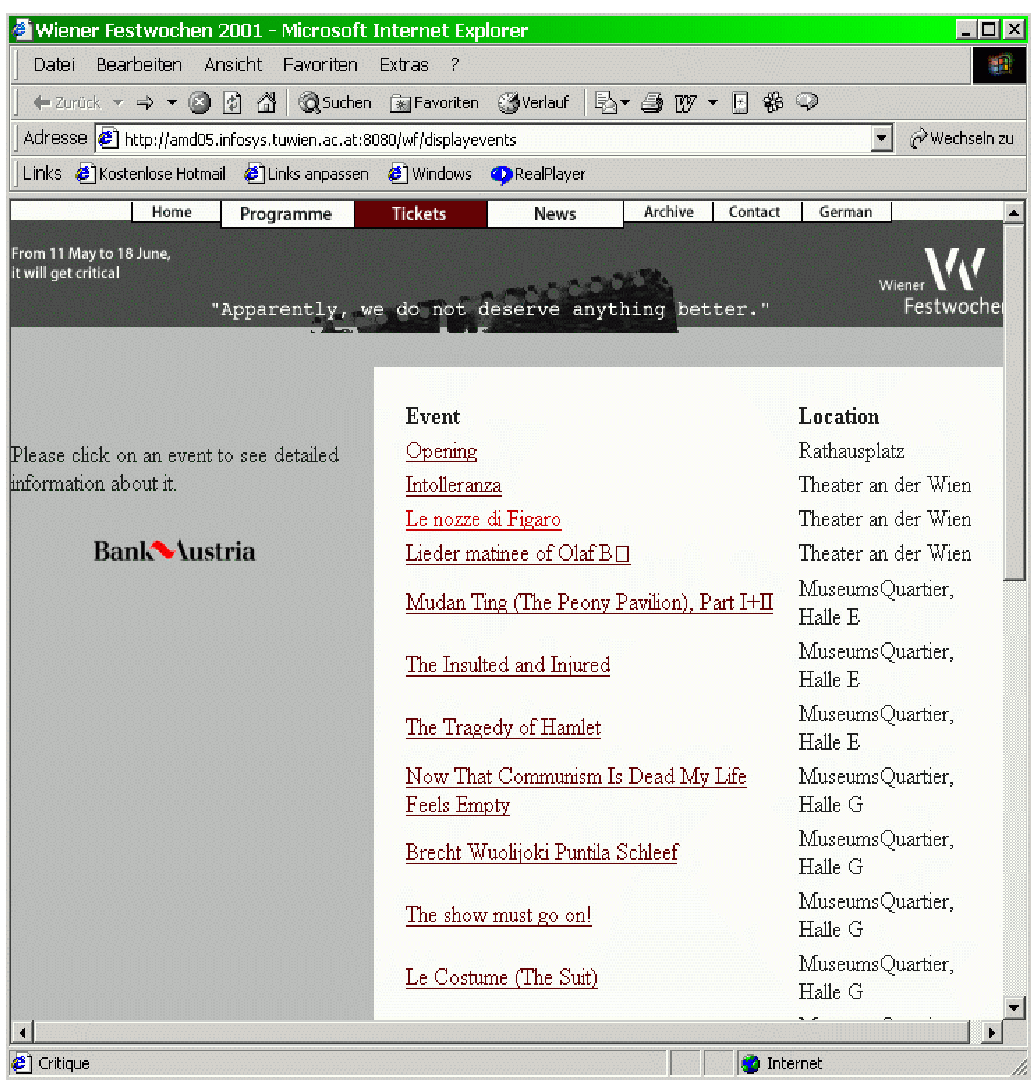


Figure 6.4: Default HTML programme page

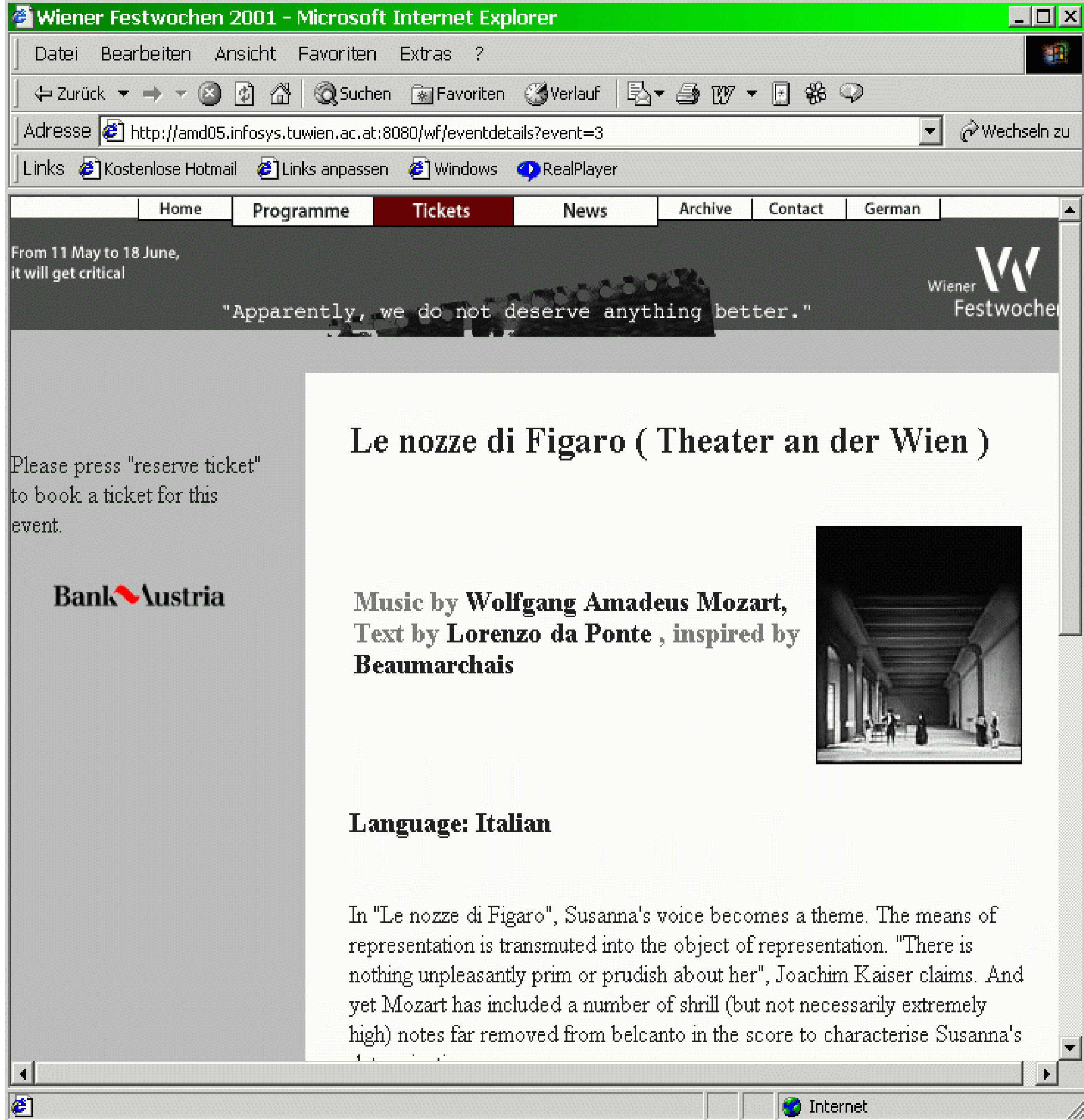


Figure 6.5: Default HTML detailed event information

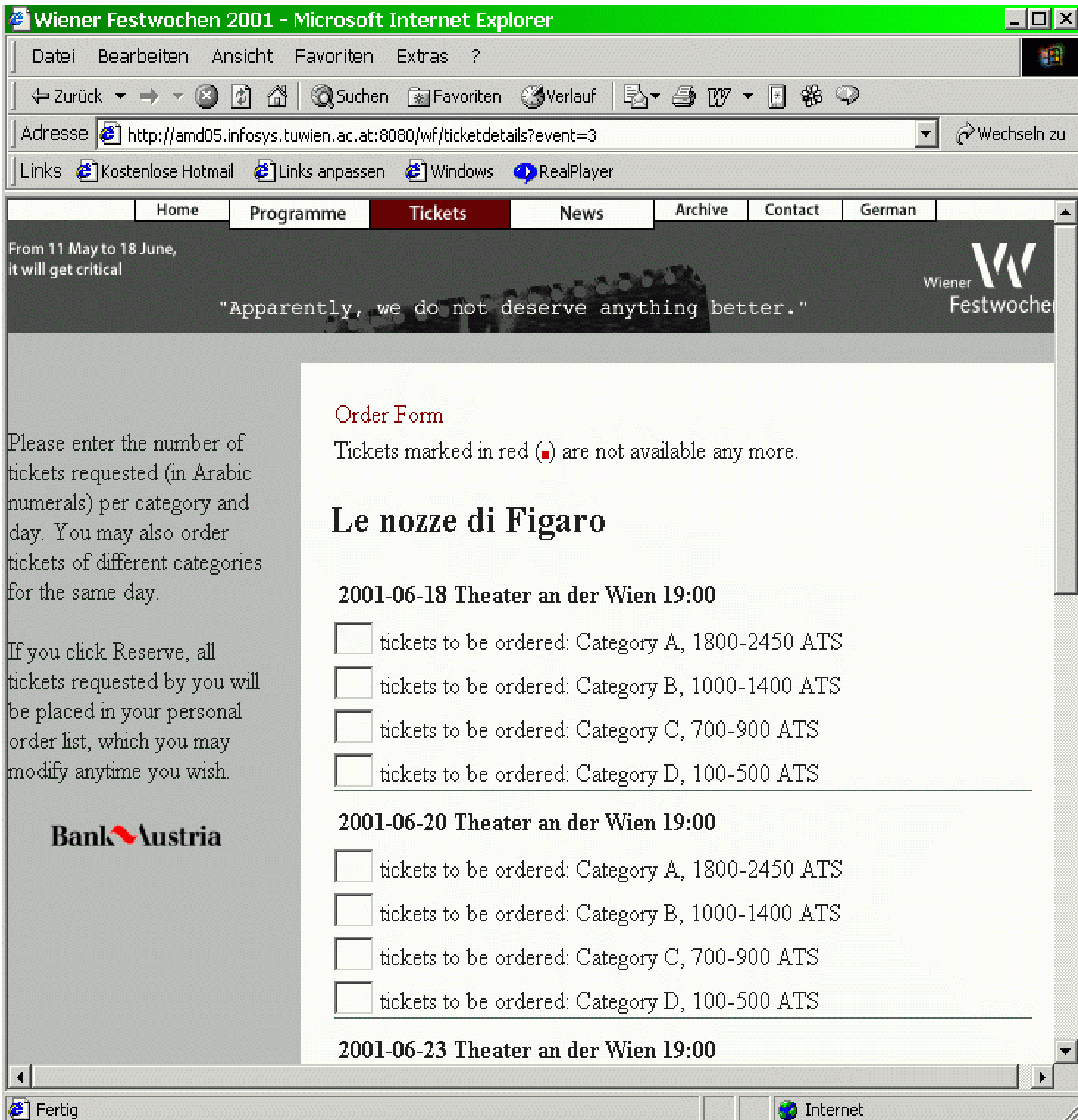


Figure 6.6: Default HTML ticket reservation page

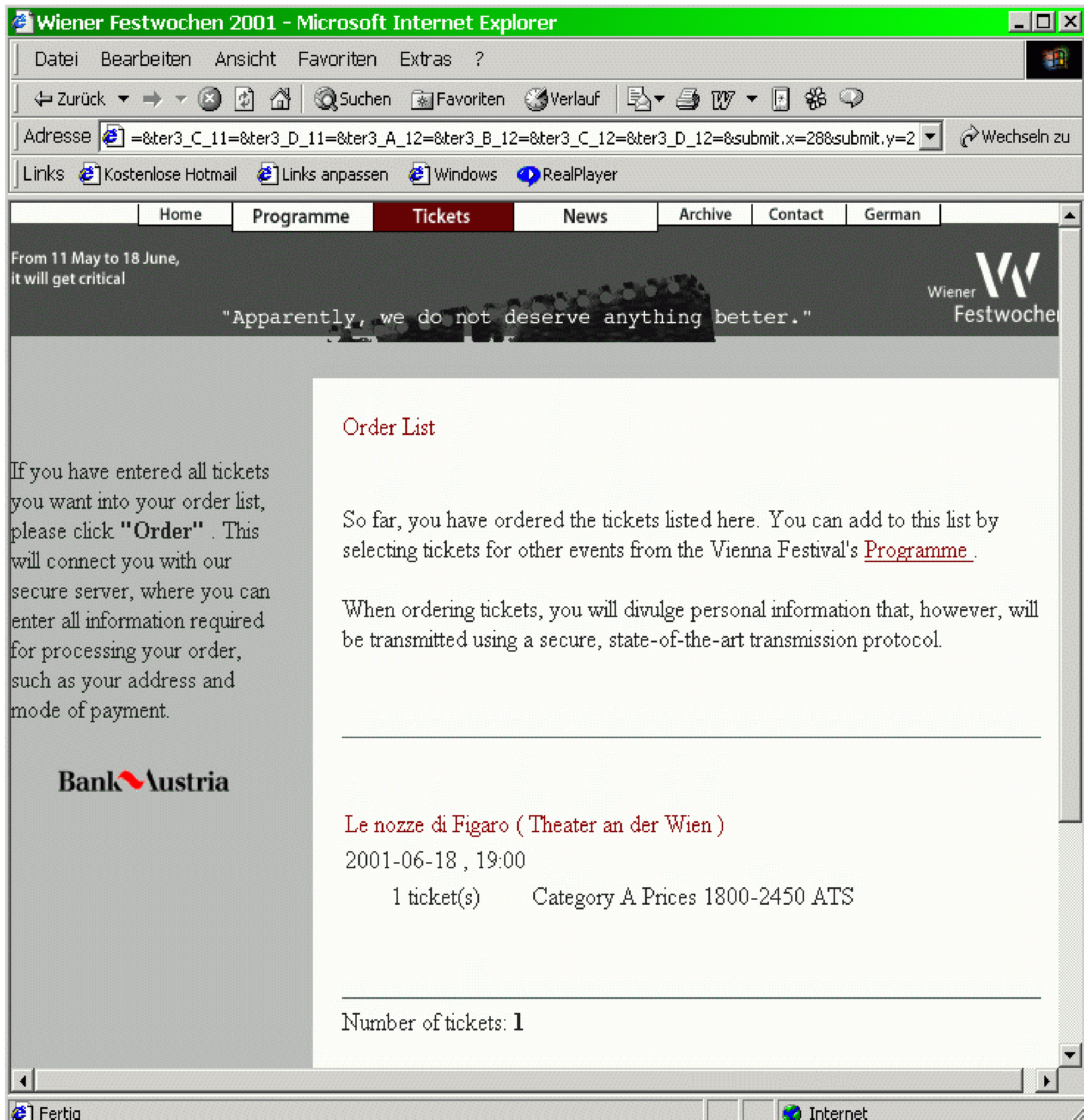


Figure 6.7: Default HTML shopping cart

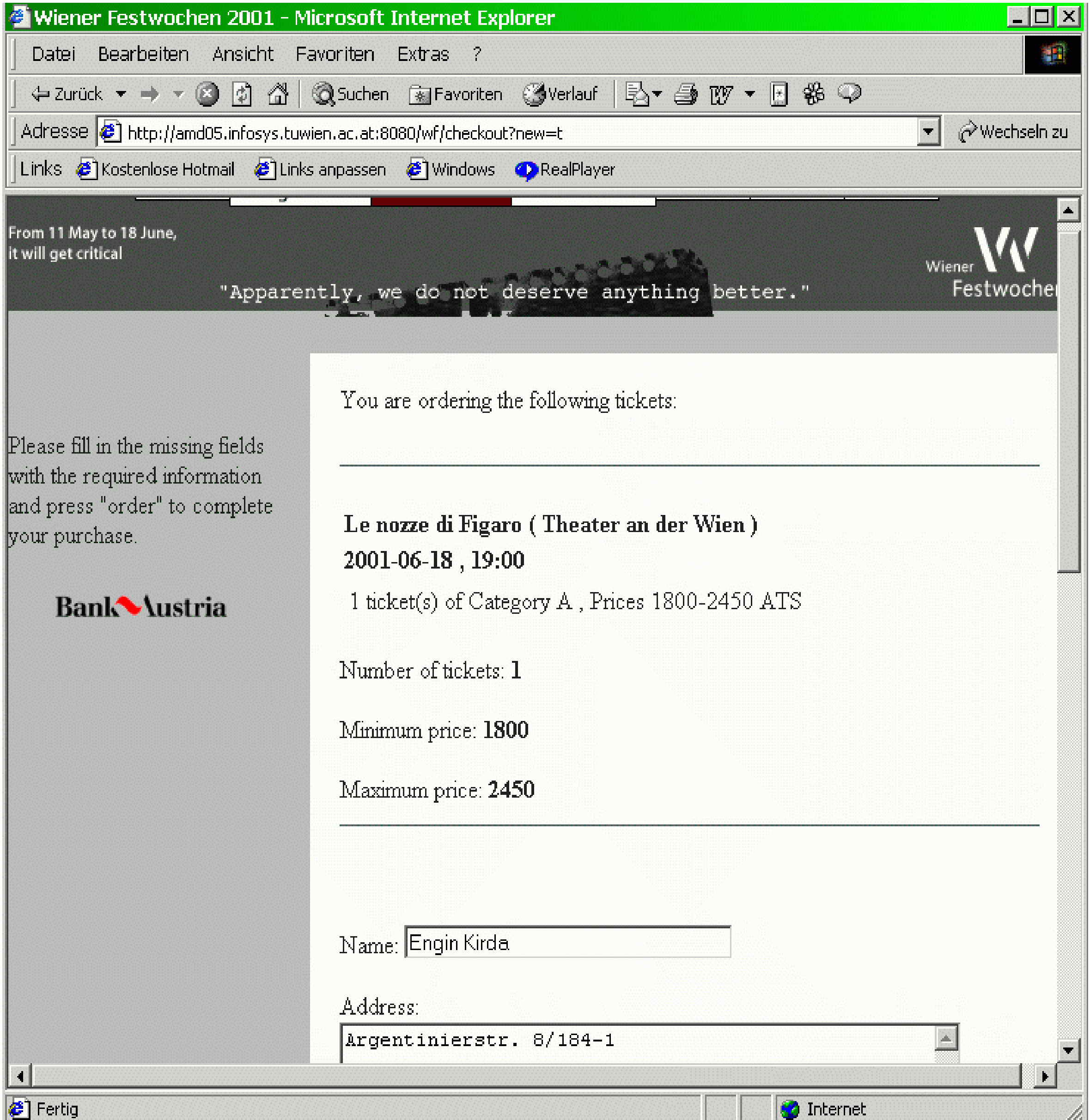


Figure 6.8: Completing the order (checking out) in the default HTML layout

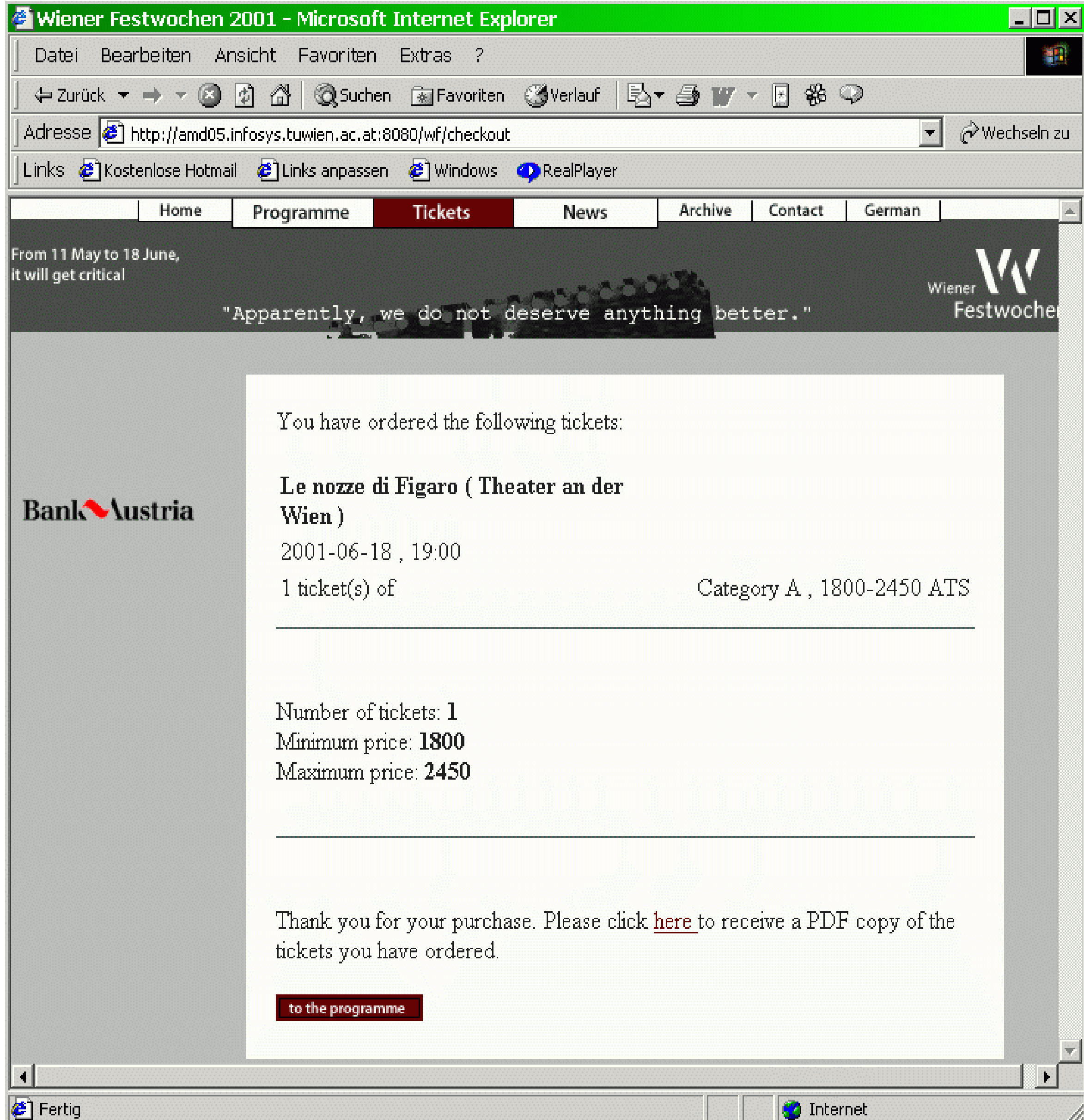
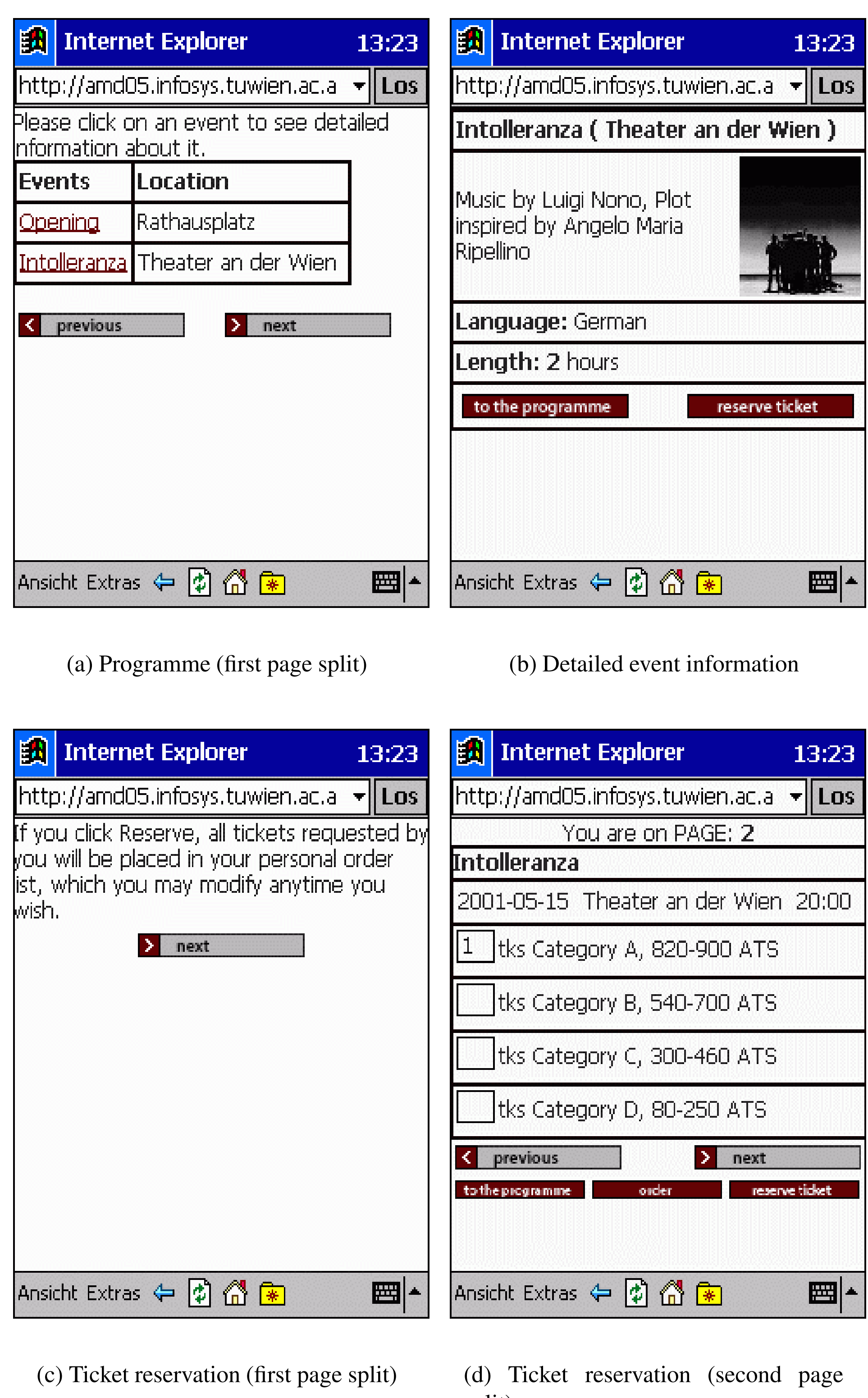


Figure 6.9: Default HTML order confirmation



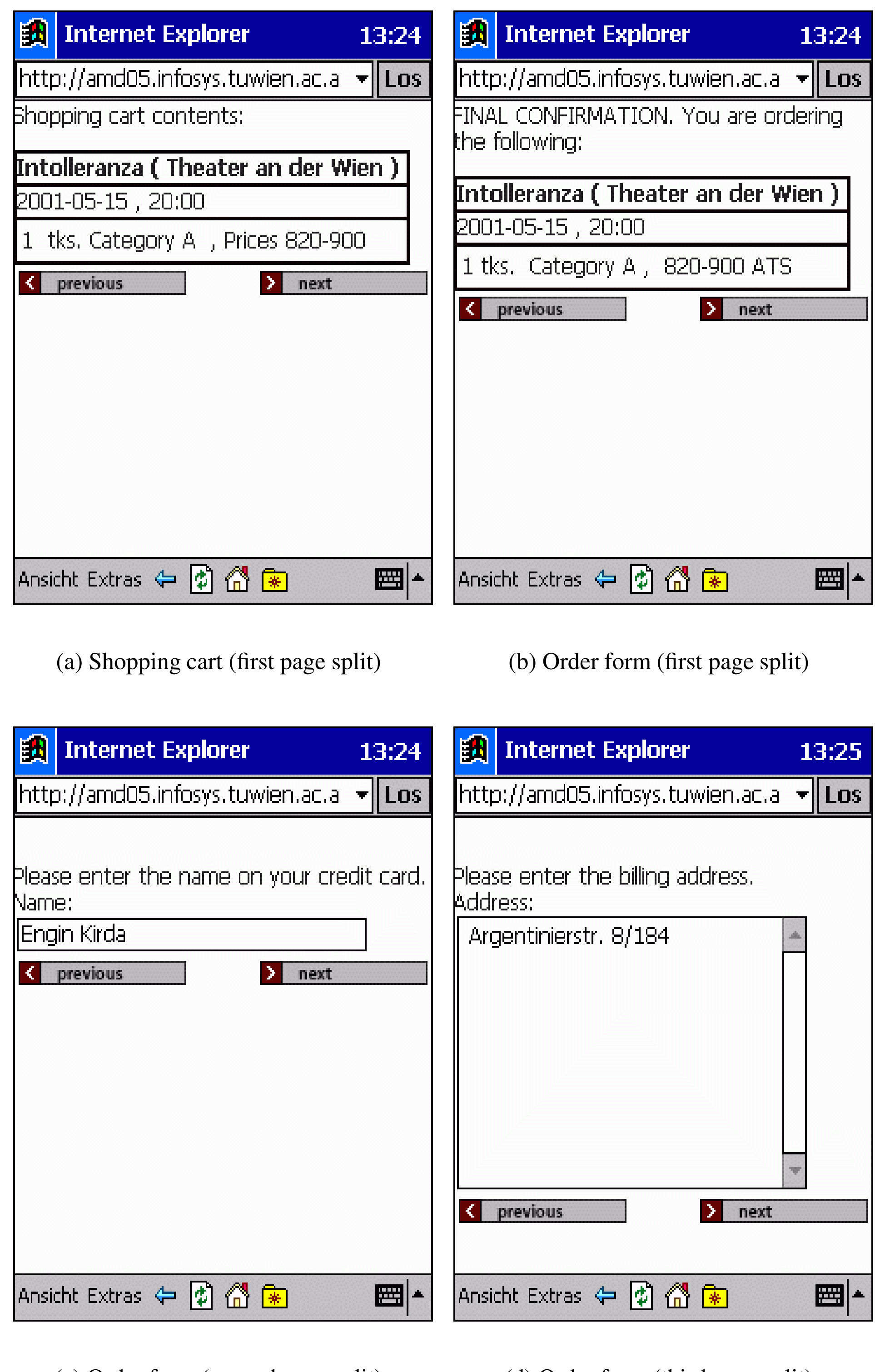
(a) Programme (first page split)

(b) Detailed event information

(c) Ticket reservation (first page split)

(d) Ticket reservation (second page split)

Figure 6.10: Programme, detailed event information and ticket reservation for the PDA device family (screenshots from an iPAQ running Windows CE)



(a) Shopping cart (first page split)

(b) Order form (first page split)

(c) Order form (second page split)

(d) Order form (third page split)

Figure 6.11: Shopping cart and order form for the PDA device family (screenshots from an iPAQ running Windows CE)

(a) Programme (first page split)

(b) Detailed event information

(c) Ticket reservation (first page split)

(d) Ticket reservation (second page split)

Figure 6.12: Programme, detailed event information and ticket reservation for the WAP device family (as seen on a WAP emulator)

(a) Ticket reservation (third page split)

(b) Shopping cart (second page split)

(c) Shopping cart (third page split)

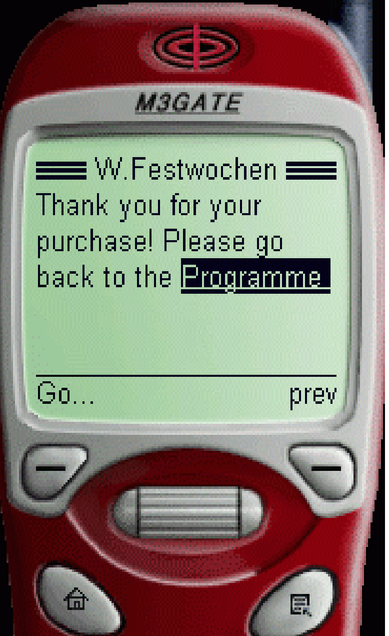
(d) Shopping cart (fourth page split)

Figure 6.13: Part of ticket reservation and shopping cart for the WAP device family (as seen on a WAP emulator)

(a) Order form (first page split)

(b) Order form (second page split)

(c) Order form (third page split)



(d) Final message

Figure 6.14: Order form for the WAP device family (as seen on a WAP emulator)

Chapter 7

Evaluation and Future Work

The device-independent implementation of the VIF case study with the MyXML tool suite provides access to four different Web devices: traditional HTML browsers, micro browsers on PDAs, WAP-enabled mobile phones and PDF readers. The case study backs my general thesis that Web services can effectively be made device-independent if device-independence support is integrated into the Web service design, implementation and maintenance phases and that adaptation is not only the key to *mobile* information access [Sat96b], but to multi-device access in general.

This chapter analyzes the DIWE framework and the concepts of page splitting, process partitioning and XSL stylesheet pre-processing in practice. It discusses the advantages and disadvantages of the concepts and compares the DIWE approach to existing solutions. The chapter also lays out future work.

7.1 Empirical proof of concepts

This section discusses the difficulty of providing useful empirical data to measure and compare the extensibility and maintainability of Web service engineering approaches. The problem may become even more complex if device-independence is involved.

7.1.1 Setting up an experiment

To set up controlled experiments to measure the flexibility, extensibility and maintainability provided by the tools and concepts presented in the dissertation, software metrics are necessary. Some software metrics and methods for metric definition have already been introduced that can be used to measure qualities such as complexity, productivity and maintainability (e.g., see [Fen96, vSB99]). The problem, however, is that Web services are not traditional software: They do not only consist of source code and libraries, but also content, layout files and a large number of resources such as images.

The following example illustrates the ineffectiveness of using traditional software engineering metrics for Web services.

7.1.2 Example: Measuring readability

If traditional metrics are used, one can show that the readability of the logic code improves using the DIWE framework: It has been reported that readability of code is an important factor in determining maintainability.

De Young and Kampen defined (in [YK79]) the readability R of programs as:

```
R=0.295a-0.499b+0.13c
```

The variable a is the average normalized length[1] of variables, b is the number of lines containing statements, and c is McCabe's cyclomatic number[2] (see [Fen96]). The authors derived this formula using regression analysis of data about subjective evaluation of readability. They discovered that readability worsens as the number of lines in a program increase no matter how complex it is and how long the variables are. Based on this finding, it can be deduced that the readability of the logic source code increases when the DIWE framework is used. This is because the layout is not encoded into the source code and as a result the logic has less number of lines.

This empirical evidence, however, is not really convincing. Readability might improve for the application logic source code, but there is no evidence about the readability of XML and XSL files and other resources that the Web service depends on.

Special metrics are needed to measure the flexibility, extensibility and maintainability of Web services. The field of *Web metrics* is young (e.g., [MMC01]) and much work is still needed. This chapter presents a qualitative analysis of the concepts introduced in this dissertation.

7.2 Analysis and discussion

This section analyzes the device-independent implementation of the VIF e-commerce Web service with the DIWE framework (i.e., the MyXML tool suite) and compares it to the traditional single-device implementations in the past.

7.2.1 Stylesheet complexity and numbers

Using traditional servlet writing techniques, the layout information is often encoded into the source code. This can be a tedious and error-prone task. The header information that contained a logo and a navigation bar in a typical servlet-based implementation, for example, need to be duplicated in all the servlets. Whenever there is a requirements change and the general layout needs to be adapted, all the duplicated code has to be analyzed and modified. Although this approach works, the code usually becomes difficult to maintain and reuse (e.g., for different devices), and may show the typical symptoms of *spaghetti code* (e.g., poor readability).

[1] Number of characters in a variable

[2] Defines the complexity of the code

The usage of stylesheets for defining and generating the layout is criticized sometimes. The argument is that the effort spent in separating the layout information by using stylesheets is not less (and sometimes even more) than integrating the layout into the code *directly*. This argument is justified to a certain degree. Not separating the layout, however, makes device-independence support difficult.

In the case study, the use of stylesheets eased the integration, separation and maintenance of layout information. Commonalities could be grouped together and imported.

Although using stylesheets has advantages, it has disadvantages as well. The following discussion lists two stylesheet-related problems and presents solutions.

7.2.1.1 Discussion

The DIWE framework allows developers to use a separate stylesheet for each supported device, but this feature may have a negative and significant effect on maintainability. When a separate stylesheet is used for each device in a project, the number of XSL stylesheets needed to implement the service increase proportionally to the number of supported devices. For a service that supports four devices, for example, each XSL stylesheet is duplicated four times. Hence, it may become difficult to maintain repeated complex XSL functionality such as cascading *<xsl:when>* statements.

XSL stylesheet pre-processing support in the DIWE framework eliminates the problem of increased number of stylesheets in projects. The stylesheets, however, become more complex. Each stylesheet usually supports more than one device and good documentation (i.e., comments in the stylesheets) became a critical factor in reducing the complexity and readability.

When adding new devices, it is often easier to copy and adapt an XSL stylesheet rather than integrate a new layout directly into existing stylesheets with XSL pre-processing. This is because the unnecessary layout code in the stylesheet can be completely deleted – hence, increasing readability – and the new layout can be incrementally built in.

7.2.1.2 Conclusion

Obviously, a tradeoff is necessary in deciding between using separate stylesheets or stylesheet pre-processing when adding new devices. The aim is to *combine* the advantages of both approaches.

An effective solution is to initially use separate stylesheets by copying and adapting existing ones. Once the layout has been debugged and is functioning correctly, the layout is extracted and integrated into the default family stylesheets by using XSL stylesheet pre-processing.

As a result, the total number of stylesheets does not increase and one can effectively deal with the increased complexity of using XSL pre-processing when adding new devices.

7.2.2 Complexity

The traditional, single-device implementation of the VIF e-commerce Web service with servlets took three days, but more than a week was needed to have a first running device-independent version. This section discusses the problem of increased design and implementation complexity of device-independent Web services.

7.2.2.1 Discussion

Obviously, the design and implementation of device-independent Web services is more complex than traditional Web engineering techniques and needs more time. The main reason is because more steps are involved (e.g., content definition with a sufficient description granularity) and the separation of layout, content and application logic not only needs more analysis, but is also more difficult to implement.

XSL requires the *programming* of the layout by use of templates and XSL commands. Hence, although the layout becomes more flexible, building and debugging the initial layout requires a significantly higher effort.

7.2.2.2 Conclusion

The advantages provided by the DIWE framework may not be apparent during the design and implementation phase, but the extra deployment effort pays off once new devices are added to the service.

In the case study, although it took longer to create a device-independent version of the VIF e-commerce Web service, adding new device layouts during maintenance was much easier than traditional approaches and technologies. For example, once the XSL infrastructure had been built, both the PDA and WAP layouts were built within one day without any modifications to the application logic.

The more devices that need to be supported by the Web service, the more the usage of the DIWE framework pays off. Setting up a service initially is more difficult, but it enables the construction of custom-tailored services that can meet evolving access requirements (e.g., VoiceXML-based speech access in the near future).

7.2.3 Layout adaptation

Figure 7.1 shows the screenshot of the full HTML layout of the VIF programme as seen on an iPAQ PDA and motivates the usage of the page splitting and process partitioning concepts in the case study. The user is only able to see a small proportion of the available information and needs to scroll a lot.

Although the idea of page splitting and process partitioning works, how much effort is necessary to deploy the techniques? The following discussion evaluates page splitting and process partitioning in practice.

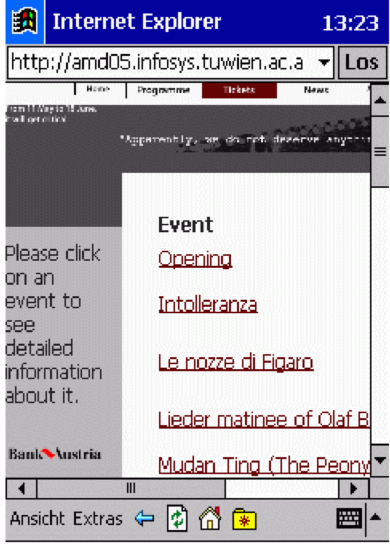


Figure 7.1: The full HTML interface of the VIF programme as seen on an iPAQ PDA

7.2.3.1 Discussion

When a PDA layout was being added to the VIF e-commerce Web service in the case study, groups and subgroups had to be defined in the stylesheets.

The same group and subgroup definitions were used in the stylesheets for supporting WAP access. Only minor adaptations were necessary. By using different step values (i.e., 3 for PDA and 2 for WAP),the grouping and subgrouping infrastructure could be reused for page splitting and process partitioning on two different device families.

Hence, the design of groups and subgroups for device families with similar restrictions and characteristics is only required once and the design can often be reused.

7.2.3.2 Conclusion

Clearly, splitting and process partitioning imposes an extra design effort on the Web developer. This effort, however, is acceptable because 1) it is not needed for every device family 2) in most cases, it can be reused (e.g., for mobile devices).

In the case study, for example, four devices are supported and page splitting and process partitioning is only needed for PDAs and WAP phones. For both devices, grouping and subgrouping was done once.

7.2.4 Graphical and navigational design

In the usage scenarios presented in the last chapter, Dr K. accesses the VIF service using different devices. Although the functionality is the same, the ways the interactions are supported are different. When viewing the shopping cart contents on a PDA, for example, Dr K is required to press the *next* button at the bottom of each page to continue, but no such button exists in the default layout.

The following discussion analyzes the graphical and navigational design issues involved in device-independent Web service engineering.

7.2.4.1 Discussion

One difficulty of device-independent Web engineering is that the main navigation and layout features may not work on some devices. In the case study, for example, the main navigational information in the default HTML layout was in the header of each page. Putting the navigational information in the header of the PDA interface, on the other hand, did not make any sense because of the small display size. Furthermore, using a header was also not possible on WAP devices.

Hence, the navigation and layout features may not always be portable to other devices. As a consequence, the graphical design process of device-independent Web services differ from traditional, single-device Web services and there is a need for a systematic approach.

7.2.4.2 Conclusion

The DIWE framework does not focus on navigation and layout design issues. Its focus is on the engineering of flexible and extensible Web services that can effectively support different layouts for different Web devices.

The layout and navigation features of a device-independent Web service often have to be redesigned for most devices and there is a considerable effort involved. It is important to consider this effort during the design stage. When more than one layout is involved, the interactions and the navigational model have to adapted to the device characteristics.

7.2.5 Layout/Content/Logic (LCL) separation

This section discusses LCL separation in device-independent Web engineering.

7.2.5.1 Discussion

Although a full LCL separation has many advantages such as multi-lingual [3] and multi-device support, a full separation is not always easy to achieve. Application logic separation can be quite easy, but the main problem is the separation of content and layout. The effort needed to achieve a full separation of layout and content may not be trivial and there may be a tendency by Web developers to make *quick fixes* by intermixing them.

Dealing with hyper-links, for example, often raises the question of where the links belong: are they content or layout? It is usually better to treat hyper-links as content because a link description (i.e., text such as "click here to continue") is described in a specific language. Encoding this link directly into a stylesheet eliminates the possibility of reusing the stylesheet for multiple languages.

[3] e.g., Separating the content enables the stylesheets to be reused for supporting content in different languages

On the other hand, it is often much easier and faster to encode links directly into a stylesheet without defining and selecting them as content.

7.2.5.2 Conclusion

Obviously, a tradeoff is necessary in separating layout and content. The aim should be to achieve a complete separation of layout and content whenever possible, but if there are time problems, content may be encoded into the stylesheet. It is important, however, to make corrections later and to continue supporting the separation for easing maintenance and future extensions.

The process is similar to writing source code and documenting it later. Unfortunately, the problems with this approach are also similar: Just as there may be a tendency not to document code although it is written with the intention of documenting later, there may also be a tendency to ignore the LCL separation goal during maintenance.

7.2.6 Comparison of the DIWE framework to other approaches

This section compares the DIWE framework to the related approaches. Tables 7.1 and 7.2 show the comparison and evaluation of the DIWE framework with the device-independent Web engineering taxonomy defined in Chapter 3.

Table 7.1 compares the general technical features, the life cycle support and the usability of each approach. Based on the discussion in Chapter 4, the *Deployment* phase has also been inserted into life cycle section in the table.

It can be seen in the table that OOH, IStudio and WebML are the only approaches besides DIWE that have full life cycle support. Although these approaches cover the Web service life cycle, only DIWE provides all the technical features that are important for constructing Web services. WebML, for example, does not have any dynamic content support and OOH does not support the integration of external databases.

In comparison, Cocoon and Total e-mobile are conceptually platform independent and provide all important technical features, but do not cover the full Web service life cycle.

When usability is evaluated, DIWE is not easy to learn and requires high developer skills when compared to the other approaches. A user interface, however, is provided to make its usage easier.

Table 7.2 compares the standard usage, flexibility and maintainability and device-independence support of each approach.

Only Cocoon, Total e-mobile and DIWE use layout and content definition standards. Most of the other approaches at least use one standard for content definition (e.g., XML in WebML), but the layout is defined in a system-specific, proprietary way.

When the flexibility and maintainability of each approach is evaluated, the table shows that Cocoon and DIWE are the only approaches that provide a maximum flexibility and maintainability because they support the complete layout, content and logic separation (LCL).

The importance of logic reuse has been identified by most of the approaches: SISL, UIML, iStudio, Cocoon and DIWE all support logic reuse.

	Approach Name	OOH	WebML	JML	SISL	UIML	iStudio	Cocoon	MS MDT	Total e-Mobile	DIWE
	Main Objective	To support all Web devices	To support all Web devices	To support all Web devices	To support speech interfaces	To support all User Interfaces	To support all Web devices	To support flexible services	To support mobile devices	To support mobile devices	To support all Web devices
General Technical Features	Conceptually Platform Independent	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
	External Database Integration	No	Yes	Yes	No	No	No	Yes	Yes	Yes	Yes
	Static Content Support	Yes	Yes	Yes	No	Yes	No	Yes	No	Yes	Yes
	Dynamic Content Support	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Life Cycle Support	Design Support	Yes	Yes	No	No	No	Yes	No	No	No	Yes
	Implementation Support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Deployment Support	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Maintenance Support	Yes	Yes	Yes	No	No	Yes	Yes	No	No	Yes
Usability	Ease of Learning	Low	Medium	High	Medium	Medium	Medium	Medium	High	Medium	Medium
	Required Developer Skills	Medium	Medium	High	Low	Medium	Medium	High	Low	Medium	High
	Service Complexity	Medium	Medium	Medium	Low	Low	Medium	Medium	Low (hidden)	Unknown	Medium
	Visual Interface	Yes	Yes	No	No	No	Yes	No	Yes	No	Yes

Table 7.1: Comparison of the DIWE framework with other approaches

	Standards		Flexibility and Maintainability						Device-Independence Support	
Approach Name	Standard Content Definition (e.g., XML)	Standard Layout Definition (e.g., XSL)	Overall Service Maintainability	Overall Service Flexibility	LC Separation	LL Separation	LCL Separation	Logic Reuse	XML Web Formats	Device Detection
OOH	No	No	Medium	Medium	No	Yes	No	Yes	Yes	No
WebML	Yes	No	Medium	Medium	Yes	No	No	No	Yes	No
JML	Yes	No	Low	Medium	Yes	No	No	No	Yes	No
SISL	No	No	Low	Low	No	Yes	No	Yes	No	No
UIML	Yes	No	Low	Low	No	Yes	No	Yes	Yes	No
iStudio	Yes	No	Medium	Medium	No	Yes	No	Yes	Yes	No
Cocoon	Yes	Yes	High	High	Yes	Yes	Yes	Yes	Yes	Yes
MS MDT	No	No	Low	Low	No	No	No	No	Yes	Yes
Total e-Mobile	Yes	Yes	Medium	Medium	Yes	Yes	No	Unknown	Yes	Yes
DIWE	Yes	Yes	High	High	Yes	Yes	Yes	Yes	Yes	Yes

Table 7.2: Comparison of the DIWE framework with other approaches

Cocoon, MS MDT, Total e-mobile and DIWE are the only approaches that have full device-independence support. Although WebML, OOH, JML, UIML and iStudio support different XML Web formats, they provide no support for device detection.

The tables show that although DIWE is not easier to learn or use than most of the other approaches, it provides maximum flexibility and maintainability, important technical device-independence features and full support for the Web service life cycle. Furthermore, it is one of the only approaches that uses standards for content and layout definition and hence can be used together with other industrial tools.

7.3 Laying out future work

The DIWE framework supports and enables the engineering of device-independent Web services, but there is room for improvement. This section discusses and lays out future work.

7.3.1 Higher level abstractions

One of the difficulties of Web projects is the lack of easy-to-use and easy-to-understand graphical notations for communicating with the customers. For example, UML has become a standard in software engineering projects for communicating system requirements and architecture, but how does one describe and communicate the structure of a Web site to the customers? Existing Web design methodologies (e.g., [ISB95, SR95]) are too low level for Web managers or customers without a technical background to appreciate and may cause confusion and misunderstanding (i.e., technical terms such as *node* and *entity* are often unknown to customers). Although these methodologies are useful for the developers in designing the site, they are not as useful during the requirements discussions with the involved parties.

No graphical notations have been proposed that support device-independent Web access. It would be useful, for example, to able to depict which pages provide which services on different devices.

There is a need for more work in this area for improving communication with non-technical users and customers.

7.3.2 UML for visual modeling

In [Con99] Conallen proposed an extension of UML for modeling Web applications. However, the use of UML in modeling Web applications has not universally been accepted by Web developers yet. These extensions of UML for the Web domain concentrate on the modeling of the *architectures* of Web applications and not the *information structure* for Web sites. Furthermore, it remains to be seen if UML will be easy to understand by Web managers and customers who may lack technical knowledge and experience in object-oriented domain modeling.

The UML model that Conallen proposes assumes that the Web service will be HTML-based. The model, hence, needs to be extended for device-independent Web services.

7.3.3 Re-engineering for device-independence

An important question that remains to be discussed is how to deal with existing Web applications. In many cases, it is not feasible to rewrite these applications to meet the new device-independence requirements.

Not much work exists on the *re-engineering* of Web applications to make them flexible and multi-device-aware. The developer in the field often has to deploy ad-hoc techniques and tools if she is faced with a need to re-engineer Web sites and applications for device-independent access.

Although some work has been done in re-engineering and analyzing Web sites (e.g., [RP00, RT01]), the adaptation of legacy Web applications to make them flexible and multi-device-enabled has received less attention. [HH01] presents a framework to recover the architecture of Web applications to gain a better understanding of the underlying system. It does not deal with the code-adaptation of Web applications, though.

Kienle's[KM01] states that Web application reverse- engineering is ad-hoc and traditional reverse-engineering tools are ill-equipped to meet the needs of Web developers.

There is a need for re-engineering approaches and tools that aim to adapt existing Web services to make them device-independent.

7.4 Summary

This chapter analyzed the DIWE framework and the concepts of page splitting, process partitioning and XSL stylesheet pre-processing in practice. It discussed the advantages and disadvantages of the concepts, compared the DIWE approach to existing solutions and briefly discussed future work.

Chapter 8

Conclusion

When the first laptop computers became commercially available, they were quite weak compared to desktop computers. Their displays were small and they had memory limitations. Many believed that software applications had to be adapted to cope with the technical restrictions. They were wrong. Laptops and notebooks have become so powerful in the last decade that many companies are only issuing notebooks to their employees and are not using desktop computers anymore. While notebook sales are constantly increasing, desktop sales are decreasing.

The popularity of computing devices such as PDAs (e.g., the new generation such as the Compaq iPAQ) and mobile phones (e.g., the Nokia Communicator) have been increasingly and these devices have been getting more powerful every day. Limitations such as memory and CPU power will probably become less important in the near future. Although the latest PDAs are even able to display frames, it is still important to adapt the content for these devices in order to provide a satisfactory surfing experience for users. Web services in the near future will not only have to support mobile access, but will also have to deal with other forms of Web access such as voice interfaces. Hence, Web services will often need to be *device-independent* and will have to support different XML Web formats.

My general thesis was that Web services can effectively be made device-independent if device-independence support is integrated into the Web service design, implementation and maintenance phases.

Much work has been done on providing mobile access to Web content, but the focus has mainly been the adaptation of HTML content to make it viewable on mobile devices that might have memory and screen-size limitations. Only a few attempts have been made to date to integrate device-independence into the design, implementation and maintenance phases of Web services.

The dissertation presented an extended model of the traditional Web service life cycle that takes device-independence support into account and presented the Device-Independent Web Engineering (DIWE) framework for engineering device-independent Web services. It introduced the novel concepts of page splitting, process partitioning and XSL stylesheet preprocessing. The MyXML tool suite is a prototype implementation of the DIWE framework and consists of the MyXML processor, three configurable run-time device-independence components and the MyXMLDesigner visual Integrated Development Environment (IDE). The MyXML tool suite was used in the device-independent implementation of the Vienna

International Festival e-commerce Web service. The service provides Web access to full-fledged HTML browsers, PDAs and WAP-enabled mobile phones with the same application logic.

Nielsen predicts in [Nie99] that the Web will eventually suffer a usability meltdown unless the vast majority of Web sites are improved considerably. He states that the emphasis has to be placed on quality content and software and not on “dazzle and coolness.” Not only these factors will determine the future of the Web, but also the development and usage of device-independent Web engineering techniques and tools.

Appendix A

Sample case study code listings

MyXML Document for shopping cart

```
<?xml version="1.0" ?>

<root xmlns:myxml="http://www.infosys.tuwien.ac.at/myxml/ns">

        <pageInformation>
                <explanation>
                        If you have entered all tickets you want into your order list, please click
                         <b>"Order"</b>. This will connect you with our secure server, where you can
                        enter all information required for processing your order, such as your address
                         and mode of payment.
                </explanation>
                <explanation2>
                        Please fill in the missing fields with the required information and press "order" to
                        complete your purchase.
                </explanation2>
        </pageInformation>

        <ticketinfo>
          <myxml:loop>
                <booking>
                <event_information>
                        <event_name>
                                <myxml:multiple> event_name </myxml:multiple>
                        </event_name>
                        <event_date>
                                <myxml:multiple> event_date </myxml:multiple>
                        </event_date>
                        <event_location>
                                <myxml:multiple> event_location </myxml:multiple>
                        </event_location>
                        <event_time>
                                <myxml:multiple> event_time </myxml:multiple>
                        </event_time>
                </event_information>
```

```
                <tickets>
                        <myxml:loop>
                                <loop>
                                        <category_info>
                                                <myxml:multiple> category_info </myxml:multiple>
                                        </category_info>
                                        <category_name>
                                                <myxml:multiple> category_name </myxml:multiple>
                                        </category_name>
                                        <number_of_tickets>
                                                <myxml:multiple> number_of_tickets</myxml:multiple>
                                        </number_of_tickets>
                                </loop>
                        </myxml:loop>
                </tickets>
                </booking>
          </myxml:loop>
        </ticketinfo>
        <summary>
                <totalNumberOfTickets>
                        <myxml:single> totalNumberOfTickets </myxml:single>
                </totalNumberOfTickets>
                <minimumPrice>
                        <myxml:single> minimumPrice </myxml:single>
                </minimumPrice>
                <maximumPrice>
                        <myxml:single> maximumPrice </myxml:single>
                </maximumPrice>
        </summary>
        <orderform>
                <errorMessage> <myxml:single> errorMessage </myxml:single> </errorMessage>
                <name> <myxml:single> name </myxml:single> </name>
                <address> <myxml:single> address </myxml:single> </address>
                <phonePrivate> <myxml:single> phonePrivate </myxml:single> </phonePrivate>
                <phoneWork> <myxml:single> phoneWork </myxml:single> </phoneWork>
                <email> <myxml:single> email </myxml:single> </email>
                <comments> <myxml:single> comments </myxml:single> </comments>
                <creditCard>
                        <myxml:loop>
                                <creditCardType>
                                        <myxml:multiple> creditCard </myxml:multiple>
                                </creditCardType>
                        </myxml:loop>
                </creditCard>
                <cardNumber> <myxml:single> cardNumber </myxml:single> </cardNumber>
                <validThru> <myxml:single> validThru </myxml:single> </validThru>
        </orderform>
</root>
```

XSL stylesheet for shopping cart

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:myxml="http://
www.infosys.tuwien.ac.at/myxml/ns" version="1.0">
<xsl:import href="/home/ek/eksstuff/xenon/resources/xenon.xsl"/>
<xsl:import href="/home/ek/eksstuff/xenon/resources/well-formed-html.xsl"/>
<xsl:output method="html" indent="yes"/>

<myxml:import name="/home/ek/eksstuff/xenon/case-study-devices/Styles/layout.xsl"/>

<xsl:template match="ticketinfo">
        @myxml:device:default{
        <tr><td class="hl1" align="left"> Order List </td></tr>
        <tr><td>
        So far, you have ordered the tickets listed here. You can add to this list by selecting tickets
         for other events from the Vienna Festival's
        <a class="linkb" href="/english/programme/programme.html" shape="rect" >Programme</a>.
                <br clear="none" /><br clear="none" />When ordering tickets,
                you will divulge personal information that, however, will be transmitted using a
                 secure, state-of-the-art transmission protocol.</td>
        </tr>
        <tr><td>
        <img src="/images/english/imgs/line475.gif"/><br/><br/>
        <table border="0" cellpadding="2" cellspacing="0">
        }@myxml:device
                <xsl:apply-templates/>
        @myxml:device:default{
        </table>
        </td></tr>
        }@myxml:device
</xsl:template>

<xsl:template match="booking">
        @myxml:device:pda{
        @myxml:group{
        Shopping cart contents:<br/><br/>
        <table border="1" cellspacing="0" cellpadding="0">
        }@myxml:device
        <xsl:apply-templates/>
        @myxml:device:pda{
        </table>
        <table width="400" border="0">
        <tr><td align="left">
                <a href="/wf/collector?ui=@myxml:previousGroup"><img border="0" alt="Show previous
page" src="/images/english/buttons/vorige.gif"/></a>
        </td><td align="right">
                <a href="/wf/collector?ui=@myxml:nextGroup"><img border="0" alt="Show next page" src="/
images/english/buttons/naechste.gif"/></a>
        </td></tr>
        </table>
        }@myxml:group
        }@myxml:device
</xsl:template>
```

```
<xsl:template match="event_information">
        <xsl:apply-templates select="event_name"/>
        <xsl:apply-templates select="event_date"/>
</xsl:template>

<xsl:template match="event_name">
        @myxml:device:default{
        <tr><td class="hl1" colspan="4">
        <xsl:apply-templates/>
        <xsl:apply-templates select="../event_location"/>
        </td></tr>
        }@myxml:device

        @myxml:device:pda{
        <tr><td>
        <b><xsl:apply-templates/>
        <xsl:apply-templates select="../event_location"/></b>
        </td></tr>
        }@myxml:device
</xsl:template>

<xsl:template match="event_location">
        (<xsl:apply-templates/>)
</xsl:template>

<xsl:template match="event_date">
        @myxml:device:default,pda{
        <tr><td>
        <xsl:apply-templates/>
        <xsl:apply-templates select="../event_time"/>
        </td></tr>
        }@myxml:device
</xsl:template>

<xsl:template match="event_time">
        ,<xsl:apply-templates/>
</xsl:template>

<xsl:template match="tickets">
        <xsl:apply-templates/>
</xsl:template>
```

```
<xsl:template match="loop">
        @myxml:device:default{
        <tr>
        <xsl:apply-templates select="number_of_tickets"/>
        <xsl:apply-templates select="category_name"/>
        <xsl:apply-templates select="category_info"/>
        </tr>
        }@myxml:device

        @myxml:device:pda{
        <tr><td>
        <table border="0">
        <tr>
        <xsl:apply-templates select="number_of_tickets"/>tks.
        Category <xsl:apply-templates select="category_name"/>, Prices
        <xsl:apply-templates select="category_info"/>
        </tr>
        </table>
        </td></tr>
        }@myxml:device
</xsl:template>

<xsl:template match="category_info">
        @myxml:device:default{
        <td align="center">
        Prices <xsl:apply-templates/> ATS
        </td>
        }@myxml:device

        @myxml:device:pda{
        <td>
                <xsl:apply-templates/>
        </td>
        }@myxml:device
</xsl:template>

<xsl:template match="category_name">
        @myxml:device:default{
        <td align="center">
        Category <xsl:apply-templates/>
        </td>
        }@myxml:device

        @myxml:device:pda{
        <td>
                <xsl:apply-templates/>
        </td>
        }@myxml:device
</xsl:template>
```

```
<xsl:template match="number_of_tickets">
        @myxml:device:default{
        <td align="center">
        <xsl:apply-templates/>ticket(s)
        </td>
        }@myxml:device

        @myxml:device:pda{
        <td>
                <xsl:apply-templates/>
        </td>
        }@myxml:device
</xsl:template>

<xsl:template match="minimumPrice">
        @myxml:device:default{
        <tr><td>
        }@myxml:device

        Minimum price: <b> <xsl:apply-templates/> </b>
        @myxml:device:pda{ <br/> }@myxml:device

        @myxml:device:default{
        </td></tr>
        }@myxml:device
</xsl:template>

<xsl:template match="maximumPrice">
        @myxml:device:default{
        <tr><td>
        }@myxml:device

        Maximum price: <b> <xsl:apply-templates/> </b>
        @myxml:device:pda{ <br/> }@myxml:device

        @myxml:device:default{
        <img src="/images/english/imgs/line475.gif"/>
        </td></tr>
        }@myxml:device
</xsl:template>

<xsl:template match="totalNumberOfTickets">
        @myxml:device:default{
        <tr><td>
        <img src="/images/english/imgs/line475.gif"/>
        }@myxml:device
        Number of tickets: <b> <xsl:apply-templates/> </b>
        @myxml:device:pda{ <br/> }@myxml:device

        @myxml:device:default{
        </td></tr>
        }@myxml:device
</xsl:template>
```

```
<xsl:template match="summary">
	@myxml:device:default{
	<xsl:apply-templates/>
	<tr><td>
	<table border="0" width="460">
	<tr><td align="left">
	<a href="/wf/displayevents"><img border="0" src="/images/english/buttons/programm.gif"/></a>
	</td>
	<td align="right">
	<a href="/wf/checkout?new=t"><img border="0" src="/images/english/buttons/bestellen.gif"/></a>
	</td></tr>
	</table>
	</td></tr>
	}@myxml:device

	<!-- %%%%%%%%%%%%%%%%%%%% PDA
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%-->

	@myxml:device:pda{
	@myxml:group{
	<xsl:apply-templates/>
	<table width="400" border="0">
	<tr><td align="left">
		<a href="/wf/collector?ui=@myxml:previousGroup"><img border="0" alt="Show previous
page" src="/images/english/buttons/vorige.gif"/></a>
	</td><td align="right">
		<a href="/wf/collector?ui=@myxml:nextGroup"><img border="0" alt="Show next page" src="/
images/english/buttons/naechste.gif"/></a>
	</td></tr>
	</table>
	}@myxml:group
	@myxml:group{
	<br/><br/> Please choose: <br/><br/>
	<table width="400" border="0">
	<tr><td align="left">
	<a href="/wf/displayevents?device=pda&amp;reset=t"><img border="0" src="/images/english/buttons/
programm.gif"/></a>
	</td><td align="right">
	<a href="/wf/checkout?device=pda&amp;reset=t&amp;new=t"><img border="0" src="/images/english/
buttons/abschliessen.gif"/></a>
	</td></tr>
	</table>
	}@myxml:group
	}@myxml:device
</xsl:template>

<xsl:template match="orderform">
<!-- Ignore -->
</xsl:template>

</xsl:stylesheet>
```

Application logic for shopping cart

```
public class ShoppingCart {

   private String totalNumberOfTickets = null;
   private String minimumPrice = null;
   private String maximumPrice = null;
   private String[] event_name = null;
   private String[] event_location = null;
   private String[] event_date = null;
   private String[] event_time = null;
   private String[] termin_id = null;
   private String[][] number_of_tickets = null;
   private String[][] category_name = null;
   private String[][] category_info = null;

   public void init(String totalNumberOfTickets, String minimumPrice, String maximumPrice,
                    String[] event_name, String[] event_location, String[] event_date, String[] event_time,
                    String[][] number_of_tickets, String[][] category_name, String[][] category_info) {

         this.totalNumberOfTickets = totalNumberOfTickets;
         this.minimumPrice = minimumPrice;
         this.maximumPrice = maximumPrice;
         this.event_name = event_name;
         this.event_location = event_location;
         this.event_date = event_date;
         this.event_time = event_time;
         this.number_of_tickets = number_of_tickets;
         this.category_name = category_name;
         this.category_info = category_info;
   }

   public void addEvent(String terminID, String eventName, String eventLocation, String eventDate, String
eventTime) {

         if (event_name==null) {
            event_name = new String[1];
            event_location = new String[1];
            event_date = new String[1];
            event_time = new String[1];
            termin_id = new String[1];

            event_name[0] = eventName;
            event_location[0] = eventLocation;
            event_date[0] = eventDate;
            event_time[0] = eventTime;
            termin_id[0] = terminID;

            number_of_tickets = new String[1][];
            category_name = new String[1][];
            category_info = new String[1][];
            return;
         }
```

```
        else {
          int dimension = event_name.length+1;
          String[][] number_of_tickets2;
          String[][] category_name2;
          String[][] category_info2;
          number_of_tickets2 = new String[dimension][];
          category_name2 = new String[dimension][];
          category_info2 = new String[dimension][];

          for (int i=0;i<event_name.length;i++) {
                number_of_tickets2[i] = number_of_tickets[i];
                category_name2[i] = category_name[i];
                category_info2[i] = category_info[i];
          }

          number_of_tickets = number_of_tickets2;
          category_name = category_name2;
          category_info = category_info2;
        }

        String[] name = new String[event_name.length+1];
        String[] location = new String[event_location.length+1];
        String[] date = new String[event_date.length+1];
        String[] time = new String[event_time.length+1];
        String[] termin = new String[termin_id.length+1];

        for (int i=0;i<event_name.length;i++) {
          name[i] = event_name[i];
          location[i] = event_location[i];
          date[i] = event_date[i];
          time[i] = event_time[i];
          termin[i] = termin_id[i];
        }

        name[event_name.length] = eventName;
        location[event_name.length] = eventLocation;
        date[event_name.length] = eventDate;
        time[event_name.length] = eventTime;
        termin[termin_id.length] = terminID;

        event_name = name;
        event_location = location;
        event_date = date;
        event_time = time;
        termin_id = termin;
}

public void addOrder(String numberoftickets, String categoryname, String categoryinfo) {

        int eventIndex = event_name.length-1;

        String tickets[];
        String categories[];
        String categoryinfos[];
```

```
        if (number_of_tickets[eventIndex]!=null) {
          tickets = new String[number_of_tickets[eventIndex].length+1];
          categories = new String[number_of_tickets[eventIndex].length+1];
          categoryinfos = new String[number_of_tickets[eventIndex].length+1];

          for (int i=0;i<number_of_tickets[eventIndex].length;i++) {
                tickets[i] = number_of_tickets[eventIndex][i];
                categories[i] = category_name[eventIndex][i];
                categoryinfos[i] = category_info[eventIndex][i];
          }
        }
        else {
          tickets = new String[1];
          categories = new String[1];
          categoryinfos = new String[1];

          tickets[0] = numberoftickets;
          categories[0] = categoryname;
          categoryinfos[0] = categoryinfo;
          number_of_tickets[eventIndex] = tickets;
          category_name[eventIndex] = categories;
          category_info[eventIndex] = categoryinfos;
          return;
        }

        tickets[number_of_tickets[eventIndex].length] = numberoftickets;
        categories[number_of_tickets[eventIndex].length] = categoryname;
        categoryinfos[number_of_tickets[eventIndex].length] = categoryinfo;

        number_of_tickets[eventIndex] = tickets;
        category_name[eventIndex] = categories;
        category_info[eventIndex] = categoryinfos;
}

public String getTotalNumberOfTickets() {

        int total = 0;

        for (int i=0;i<event_name.length;i++) {
          for (int t=0;t<number_of_tickets[i].length;t++) {
                Integer totalInt = new Integer(number_of_tickets[i][t]);
                total = total + totalInt.intValue();
          }
        }
        totalNumberOfTickets = new Integer(total).toString();

        return totalNumberOfTickets;
}
```

```
public String getMinimumPrice() {

        int total = 0;

        for (int i=0;i<event_name.length;i++) {
          for (int t=0;t<category_info[i].length;t++) {
                String str = category_info[i][t];
                if (str.indexOf("-")==-1) {
                  Integer totalInt = new Integer(category_info[i][t]);
                  total = total + totalInt.intValue()*new Integer(number_of_tickets[i][t]).intValue();
                }
                else {
                  Integer totalInt = new Integer(category_info[i][t].substring(0,category_info[i][t].indexOf("-")));
                  total = total + totalInt.intValue()*new Integer(number_of_tickets[i][t]).intValue();
                }
          }
        }

        minimumPrice = new Integer(total).toString();

        return minimumPrice;
}

public String getMaximumPrice() {

        int total = 0;

        for (int i=0;i<event_name.length;i++) {
          for (int t=0;t<category_info[i].length;t++) {
                String str = category_info[i][t];
                if (str.indexOf("-")==-1) {
                  Integer totalInt = new Integer(category_info[i][t]);
                  total = total + totalInt.intValue()*new Integer(number_of_tickets[i][t]).intValue();
                }
                else {
                  Integer totalInt = new Integer(category_info[i][t].substring(category_info[i][t].indexOf("-
")+1,category_info[i][t].length()));
                  total = total + totalInt.intValue()*new Integer(number_of_tickets[i][t]).intValue();
                }
          }
        }

        maximumPrice = new Integer(total).toString();

        return maximumPrice;
}
```

```
    public String[] getEventName() {
            return event_name;
    }

    public String[] getEventLocation() {
            return event_location;
    }

    public String[] getEventDate() {
            return event_date;
    }

    public String[] getEventTime() {
            return event_time;
    }

    public String[][] getNumberOfTickets() {
            return number_of_tickets;
    }

    public String[][] getCategoryName() {
            return category_name;
    }

    public String[][] getCategoryInfo() {
            return category_info;
    }
}
```

Bibliography

[Abr00] Marc Abrams. Device-Independent Authoring with UIML. In *W3C Workshop on Web Device Independent Authoring, Bristol, Englandm, http://www.harmonia.com/resources/papers/*, October 2000.

[AF99] Prathima Agrawal and David Famolari. Mobile computing in next generation wireless networks. In *3rd international workshop on Discrete algorithms and methods for mobile computing and communications (DIAL 99), Seattle, WA, USA*, August 1999.

[Alp] Alphaworks. Web Services - http://www.alphaworks.ibm.com/webservices.

[AMM+98a] P. Atzeni, G. Mecca, G. Merialdo, P. Masci, and G. Sindoni. The Araneus Web-Based Management System. In L.M. Haas and A. Tiwary, editors, *Proceedings of the International Conference Sigmod98, Exhibits Program, Seattle, WA, USA*, page 544 546, June 1998.

[AMM98b] P. Atzeni, G. Mecca, and P. Merialdo. Design and Maintenance of Data-Intensive Web Sites. In I. Ramos H. J. Schek, F. Saltor and G. Alanso, editors, *Proceedings of the International Conference on Extending Database Technology, EDBT98, Valencia, Spain*, page 436 450, March 1998.

[Ani01] Scott Anian. JCup: CUP Parser Generator for Java -
http://www.cs.princeton.edu/ appel/modern/java/CUP/ , 2001.

[ant02] Apache Jakarta ANT -
http://jakarta.apache.org/ant. Technical report, 2002.

[AP99] Marc Abrams and Constantinos Phanouriou. UIML: An XML Language for Building Device-Independent User Interfaces. In *XML '99 Conference, Philadelphia, PA, USA, http://www.harmonia.com/resources/papers/*, December 1999.

[Apa01a] Apache. Xalan XSL Processor - http://xml.apache.org/xalan-j , 2001.

[Apa01b] Apache. Xerces XML Parser - http://xml.apache.org/xerces-j , 2001.

[APBW99] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, and Stephen M. Williams. UIML: an appliance-independent XML user interface

language. In *Proceedings of the 8th International World Wide Web Conference, Toronto, Canada*, volume 31 of *Computer Networks*, page 1695 1708. Elsevier Science, 1999.

[Arc01] Tom Archer. *Inside C#*. Microsoft, 2001.

[BCD+00] Thomas Ball, Christopher Colby, Peter Danielsen, Lalita Jategaonkar Jagadeesan, Radha Jagadeesan, Konstantin Laeufer, Peter Mataga, and Kenneth Rehor. Sisl: Several interfaces, single logic. *International Journal of Speech Technology*, 3:93 108, 2000.

[BCL+94] T. Berners-Lee, R. Cailliau, A. Loutonen, H. F. Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM*, 37(8), August 1994.

[Ber01] Eliot Berk. JLex: A Lexical Analyser Generator for Java- http://www.cs.princeton.edu/ appel/modern/java/JLex/, 2001.

[BFJT01] George Buchanan, Sarah Farrant, Matt Jones, and Harold Thimbleby. Improving Mobile Internet Usability. In *Proceedings of the 10th International World Wide Web Conference, Hong Kong, China*, May 2001.

[BGP00] Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. Focused Web searching with PDAs. In *Proceedings of the 9th International World Wide Web Conference, Amsterdam, Netherlands*, May 2000.

[BGP01] Orkut Buyukkokten, Hektor Garcia-Molina, and Andreas Paepcke. Seeing the Whole in Parts: Text Summarization for Web Browsing on Handheld Devices. In *Proceedings of the 10th International World Wide Web Conference, Hong Kong, China*, May 2001.

[blu02] Hp bluestone home page, http://www.bluestone.com, 2002.

[BMY95] V. Balasubramanian, Bang Min Ma, and Joonhee Yoo. A Systematic Approach to Designing a WWW Application. *Communications of the ACM*, 38(8):47–8, August 1995.

[BN96] Martin Bichler and Stefan Nusser. Modular Design of Complex Web-Applications with W3DT. In *Proceedings of the 5th Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '96)*, page 328 333. IEEE Comput. Soc. Press., Los Alamitos, CA, USA, 1996.

[BS97] Timothy W. Bickmore and Bill N. Schilit. Digestor: Device-Independent Access To The World Wide Web. In *Proceedings of the 6th World Wide Web Conference, Santa Clara, CA, USA*, 1997.

[BS98] Robert Barta and Markus W. Schranz. JESSICA – An Object-Oriented Hypermedia Publishing Processor. *Computer Networks and ISDN Systems*, 30(1–7):281, Apr. 1998.

[BS00a] Robert Barta and Markus Schranz. Syndication with JML. In *Proceedings of the ACM Symposium on Applied Computing, Como. Italy*, pages 962–70, March 2000.

[BS00b] C. Bauer and A. Scharl. Tool-supported Web Development: Rethinking Traditional Modeling Principles. In *Proceedings of the 8th European Conference on Information Systems, Vienna, Austria*, volume 1, pages 282–289. Vienna University of Econ. and Bus. Adm., 2000.

[CE99] S. Chandra and C.S. Ellis. JPEG Compression metric as a quality aware image transcoding. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, page 81 92. USENIX Assoc., Berkeley, CA, USA, 1999.

[CEV99] Surendar Chandra, Carla SChlatter Ellis, and Amin Vahdat. Multimedia Web Services for Mobile Clients Using Quality Aware Transcoding. In *2nd ACM International Workshop on Wireless Mobile Multimedia (WoWMoM 99), Seattle, WA, USA*, August 1999.

[CEV00] Surendar Chandra, Carla Schlatter Ellis, and Amin Vahdat. Application-Level Differentiated Multimedia Web Services Using Quality Aware Transcoding. *IEEE Journal on selected areas in communications*, 18(12):2544 2565, December 2000.

[CFB00] Stefano Ceri, Piero Fraternali, and Aldo Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. In *Proceedings of the 9th World Wide Web Conference, Amsterdam, Netherlands*, volume 33 of *Computer Networks*, page 137 157. Elsevier Science B.V, May 2000.

[CFP99] Stefano Ceri, Piero Fraternali, and Stefano Paraboschi. Data-Driven, One-To-One Web Site Generation for Data-Intensive Applications. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 615–626. Morgan Kaufmann, 1999.

[Coc96] Alistair Cockburn. The Interaction of Social Issues and Software Architecture. *Communications of the ACM*, 39(10):40–6, October 1996.

[col] Coldfusion home page, http://www.coldfusion.com.

[Con99] Jim Conallen. Modeling Web Application Architectures with UML. *Communications of the ACM*, October 1999.

[cvs] CVS,
http://cellworks.washington.edu/pub/docs/cvs.

[dev] Essential .NET :Component Development with C#. Technical report, Developmentor.

[DIMG95] Alicia Diaz, Tomas Isakowitz, Vanesa Maiorana, and Gabriel Gilabert. RMC: A Tool To Design WWW Applications. December 1995.

[DMCS95] D.B.Ingham, M.C.Little, S.J. Caughey, and S.K. Shrivastava. W3Objects: bringing object-oriented technology to the Web. In *Proceedings of the 4th International World Wide Web Conference, Boston, MA, USA*, 1995.

[Eng95] Douglas C. Engelbart. Toward Augmenting the Human Intellect and Boosting our Collective IQ. *Communications of the ACM*, 38(8):30–3, August 1995.

[FC96] Mohamed Fayad and Marshall P. Cline. Aspects of Software Adaptability. *Communications of the ACM*, 39(10):58–9, October 1996.

[Fen96] Norman E. Fenton. *Softare Metrics*. Thomson Computer Press, 1996.

[FFKL98] Mary Fernandez, Daniela Florescu, Jaewoo Kang, and Alon Levy. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In *Proceedings of Sigmod '98, Seattle, Washington, USA*, page 414 425, June 1998.

[FKST00] Thomas Feyer, Odej Kao, Klaus-Dieter Schwebe, and Bernhard Thalheim. Design of Data-Intensive Web-Based Information Services. In *Proceedings of the First International Conference on Web Information Systems Engineering*, volume 1, page 462 467. IEEE Computer Society, Los Alamitos, CA, USA, 2000.

[FP00] Piero Fraternali and Paolo Paolini. Model-Driven Development of Web Applications: The Autoweb System. *ACM Transactions on Information Systems*, 18(4):323 382, 2000.

[Fra97] Larry Francis. Mobile computing - a fact in your future. In *15th Annual International Conference on Computer Documentation (SIGDOC 97), Snowbird, UT, USA*, October 1997.

[Fra99] Piero Fraternali. Tools and approaches for developing data-intensive applications: A survey. *ACM Computing Surveys*, 31(3):227 263, 1999.

[GCP01] Jaime Gomez, Christina Cachero, and Oscar Pastor. Conceptual Modeling of Device-Independent Web Applications. *IEEE Multimedia*, 8(2):26–39, April-June 2001.

[GGS+99] Martin Gaedke, Hans-W. Gellersen, Albrecht Schmidt, Ulf Stegemueller, and Wolfgang Kurr. Object-oriented web engineering for large-scale web service management. In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*. IEEE Computer Society, Los Alamitos, CA, USA, January 1999.

[GJJL00] Patrice Godefroid, Lalita Jagadeesan, Radha Jagadeesan, and Konstantin Laeufer. Automated systematic testing for constraint-based interactive services. pages 40–50. ACM Press, November 2000.

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Gla01] Steve Glasgow. Enterprise Applications, Electronic Commerce and XML. In *Proceedings of OMG Days, Vienna, Austria*. OMG, February 2001.

[GM01] Athula Ginige and San Murugesan. Web Engineering: An Introduction. *IEEE Multimedia, Special Issue on Web Engineering*, 8(1):14–18, March 2001.

[Goe98] Karl M. Goeschka. *Architectures of Web applications*. PhD thesis, 1998.

[GWG97a] Hans Werner Gellerson, Robert Wicke, and Martin Gaedke. Web composition: An object oriented support system for the web engineering life cycle. *Computer Networks and ISDN Systems*, pages 1429–38, April 1997.

[GWG97b] Hans Werner Gellerson, Robert Wicke, and Martin Gaedke. Web composition: An object oriented support system for the web engineering life cycle. *Computer Netowrks and ISDN Systems*, pages 1429–38, April 1997.

[Har99] Elliotte Rusty Harold. *XML Bible*. IDG Books, 1999.

[HH01] Ahmed Hassan and Richard C. Holt. Towards a better understanding of Web applications. In Scot Tilley, editor, *Proceedings of the 3rd Web Evolution Workshop, International Conference on Software Maintenance 2001, Florence, Italy*, page 112 116. IEEE Computer Society Press, November 2001.

[HKO$^+$00] Masahiro Hori, Goh Kondoh, Kouichi Ono, Shin ichi Hirose, and Sandeep Singhal. Annotation-based Web content transcoding. In *Proceedings of the 9th International World Wide Web Conference, Amsterdam, Netherlands*, May 2000.

[HM00] Udo Hahn and Inderjeet Mani. The challenges of automatic summarization. *IEEE Computer*, 33(11):29 35, November 2000.

[HS94] Frank Halasz and Mayer Schwartz. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2):30–39, February 1994.

[ICL96] D. B. Ingham, S. J. Caughey, and M.C. Little. Fixing the ”broken link”problem: the W3Objects approach. In *Proceedings of the 5th International World Wide Web Conference, Paris, France*, volume 28 of *Computer Networks and ISDN Systems*, page 1255 1268. Elsevier Science, 1996.

[ICL97] D. B. Ingham, S. J. Caughey, and M.C. Little. Supporting highly manageable Web services. In *Proceedings of the 6th International World Wide Web Conference, Santa Clara, California*, number 29 in Computer Networks and ISDN Systems, page 1405 1416. Elsevier Science, 1997.

[ISB95] Tomas Isakowitz, Edward A. Stohr, and P. Balasubramanian. Rmm: A methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34–43, August 1995.

[Jaw98] J. Jaworski. *Java 1.2 UNLEASHED*. Sams Publ., 1998.

[KAK+00] Eija Kaasinen, Matti Aaltonen, Juha Kolari, Suvi Melakoski, and Timo Laakko. Two approaches to bringing internet services to wap devices. In *9th International World Wide Web Conference, Amsterdam, Netherlands*, May 2000.

[KBGP01] Oliver Kaljuvee, Orkut Buyukkokten, Hector Garcia-Molina, and Andreas Paepcke. Efficient Web Form Entry on PDAs. In *Proceedings of the 10th International World Wide Web Conference, Hong Kong, China*, May 2001.

[KJKS01] Engin Kirda, Mehdi Jazayeri, Clemens Kerer, and Markus Schranz. Experiences in Engineering Flexible Web Services. *IEEE Multimedia*, 8(1):58–65, April-June January - March 2001.

[KK00] Engin Kirda and Clemens Kerer. MyXML: An XML based template engine for the generation of flexible Web content. In *Proceedings of WEBNET 2000, San Antonio, Texas, USA*, November 2000.

[KK01] Clemens Kerer and Engin Kirda. Layout, Content and Logic Separation in Web Engineering. In *Proceedings of the 9th International World Wide Web Conference, 3rd Web Engineering Workshop, Amsterdam, Netherlands, May 2000*, number 2016 in Lecture Notes in Computer Science, page 135 147. Springer Verlag, 2001.

[KKJK01] Clemens Kerer, Engin Kirda, Mehdi Jazayeri, and Roman Kurmanowytsch. Building XML/XSL-Powered Web Sites: An Experience Report. In *Proceedings of the 25th International Computer Software and Applications Conference (COMPSAC), Chicago, IL, USA*. IEEE Computer Society Press, October 2001.

[KM01] Holger M. Kienle and Hausi A. Mueller. Leveraging Program Analysis for Web Site Reverse Engineering. In Scot Tilley, editor, *Proceedings of the 3rd Web Evolution Workshop, International Conference on Software Maintenance 2001, Florence, Italy*, page 117 125. IEEE Computer Society Press, November 2001.

[LB96] Songwu Lu and Vaduvur Barghavan. Adaptive resource management algorithms for indoor mobile computing envoironments. In *ACM SIGCOMM 96, Stanford, CA, USA*, August 1996.

[Lin01] Sumanth Lingham. UIML for Voice Interfaces. In *UIML Europe 2001 Conference, http://www.harmonia.com/resources/papers/*, March 2001.

[LS99] Hakon Wium Lie and Janne Saarela. Multipurpose Web Publishing: Using HTML, XML, and CSS. *Communications of the ACM*, 42(10), October 1999.

[Luc00] Bruce Lucas. Voicexml for web-based distributed conversational applications. *Communications of the ACM*, 43(9):53 57, September 2000.

[Mau96] Hermann Maurer. *Hyper-G now Hyperwave, the next generation Web solution*. Addison-Wesley England, 1996.

[MES95] Lily B. Mummert, Maria R. Ebling, and M. Satyanarayanan. Expoliting weak connectivity for mobile file access. In *15th ACM Symposium on Operating Systems Principles, Copper Mountain, CO, USA*, December 1995.

[MMC01] Emilia Mendes, Nile Mosley, and Steve Counsell. Web Metrics – Estimating Design and Authoring Effort. *IEEE Multimedia*, 8(1):50–67, April-June January - March 2001.

[Nel95] Theodor Holm Nelson. The Heart of Connection: Hypermedia Unified by Transaction. *Communications of the ACM*, 38(8):31–3, August 1995.

[Nie99] Jacob Nielsen. User interface directions for the web. *Communications of the ACM*, 42, January 1999.

[NKR+02] C. Narayanaswami, N. Kamijoh, M. Raghunath, Inoue T, T. Cipolla, J. Sanford, E. Schlig, S. Venkiteswaran, D. Guniguntala, V. Kulkarni, and K. Yamazaki. IBM's Linux watch, the challenge of miniaturization. *IEEE Computer*, 35(1):33–41, January 2002.

[NN95] Jocelyne Nanard and Marc Nanard. Hypertext design environments and the hypertext design process. *Communications of the ACM*, 38(8):49–56, August 1995.

[Pag] Perl Home Page. http://www.perl.com.

[Qui94] Christine A. Quinn. From Grass Roots to Corporate Image - The Maturation of the Web. In *Proceedings of the 2nd International World Wide Web Conference, Chicago, Illinois, USA, 17-20 October 1994*, October 1994.

[RAS00] Rob Howard Richard Anderson, Alex Homer and Dave Sussman. *A Preview of Active Server Pages+*. Wrox Press, 2000.

[RM98] Louis Rosenfeld and Peter Morville. *Information Architecture for the World Wide Web*. O'Reilly & Associates, February 1998.

[RP00] F. Ricca and P.Tonella. Web site analysis: Structure and evolution. In *Proceedings of the International Conference on Software Maintenance 2000*, page 76 86. IEEE Computer Society Press, 2000.

[RS01] D. Ralph and C. G. Shephard. Services via mobility portals. *BT Technology Journal*, 19(1):88–99, January 2001.

[RSL99] Gustavo Rossi, Daniel Schwabe, and Fernando Lyardet. *Web Application Models are more than Conceptual Models*, volume 1727 of *Lecture Notes in Computer Science*, chapter Proceedings of the World Wide Web and Conceptual Modeling '99 Workshop, ER '99 Conference, page 239 252. Springer, Paris, 1999.

[RSS+99] Harish Rawat, Sascha Schumann, Chris Scollo, Jesus M. Castagnetto, and Deepak T. Valiath. *Professional PHP Programming*. Wrox Press. Incorporated ISBN: 1861002963, 1999.

[RT01] F. Ricca and P. Tonella. Understanding and restructuring Web sites with ReWeb. *IEEE Multimedia*, 8(2), April-June 2001.

[Sat89] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. In *Proceedings of the Second IEEE Workshop on Workstation Operating Systems, Pacific Grove, California, USA*, September 1989.

[Sat96a] M. Satyanarayanan. Fundamental challenges in mobile computing. In *15th Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, USA*, May 1996.

[Sat96b] Mahadev Satyanarayanan. Accessing information on demand at any location: Mobile information access. *IEEE Personal Communications*, pages 26–30, February 1996.

[Sch97] M. W. Schranz. Management process of WWW services: An Experience Report. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering (SEKE '97),Madrid, Spain*, pages 16–23. Knowledge Systems Institute, June 1997.

[Sch98a] Arno Scharl. Reference Modeling of Commercial Web Information Systems Using the Extended World Wide Web Design Technique (eW3DT). In *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31), Hawaii, USA*. IEEE Computer Society Press, 1998.

[Sch98b] Markus W. Schranz. *World Wide Web Service Engineering – Object Oriented Hypermedia Publishing*. PhD thesis, Distributed Systems Group, Technical University of Vienna, September 1998.

[Sd98] Daniel Schwabe and Rita de Almeida Pontes. OOHDM-WEB: Rapid Prototyping of Hypermedia Applications in the WWW. Technical Report MCC 08/98, Department of Informatics, PUI-Rio, Brasil, 1998.

[Sen00] James A. Senn. The emergence of m-commerce. *IEEE Computer*, 33(12):148–51, December 2000.

[She95] Deri Sheppard. *An Introduction to Formal Specification with Z and VDM*. The McGraw-Hill International Series in Software Engineering, 1995.

[SHKE01] Andrea H. Skarra, Karrie J. Hanson, Gerald M. Karam, and Jeff Elliott. The iStudio Environment: An Experience Report. In *Proceedings of the XML in Software Engineering Workshop (XSE 2001), 23rd International Conference on Software Engineering (ICSE 2001)*, May 2001.

[SR95] Daniel Schwabe and Gustavo Rossi. The Object-Oriented Hypermedia Design Model. *Communications of the ACM*, 38(8):45–6, August 1995.

[SRB96] Daniel Schwabe, Gustavo Rossi, and Simone D.J. Barbosa. Systematic Hypermedia Application Design with OOHDM. In *Proceedings of the Seventh ACM Conference on Hypertext, New York, NY, USA*, page 116 128, 1996.

[Str95] Norbert A. Streitz. Designing hypermedia: A collaborative activity. *Communications of the ACM*, 38(8):70–1, August 1995.

[Sun] Sun. Implementing Services on Demand with the SUN Open Net Environment – Sun ONE. Technical report, Sun Microsystems.

[TL97] Kenji Takahashi and Eugene Liang. Analysis and Design of web-based Information Systems. In *Proceedings of the 6th International World Wide Web Conference, Santa Clara, CA, USA*, 1997.

[tot01] Hp bluestone mobile and wireless computing description, http://www.bluestone.com, March 2001.

[TYF86] TJ. Teorey, D. Yang, and J. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2):197–222, 1986.

[Var00] Ken Varnum. Information @ your fingertips: porting library services to the PDA. *Online*, 24(5):14 17, September - October 2000.

[vSB99] Rini van Solingen and Egon Berghout. *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. McGraw Hill, 1999.

[W3C] W3C.
Cascading Style Sheets,
http://www.w3.org/Style/CSS/ . Technical report.

[W3C98a] W3C. Extensible Markup Language (XML) 1.0 -
http://www.w3.org/TR/1998/REC-xml-19980210. Technical report, Feb. 1998.

[W3C98b] W3C.
XML Specification DTD
http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm. Technical report, Sep. 1998.

[W3C00] W3C. eXtensible Stylesheet Language 1.0 -
http://www.w3.org/TR/xsl/. Technical report, Jan. 2000.

[web01] The webml tool site, http://webml.org, 2001.

[YK79] G. E. De Young and G. R. Kampen. Program factors as predictors of program readability. In *Proceedings of the Computer Software and Applications Conference (COMSAC)*, pages 668–673. IEEE Computer Society Press, 1979.