DISSERTATION

XGuide - Concurrent Web Development with Contracts

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

unter der Leitung von

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

eingereicht an der

Technischen Universität Wien
Fakultät für Technische Naturwissenschaften und Informatik

von

Univ.-Ass. Dipl.-Ing. Clemens Kerer
clemens@infosys.tuwien.ac.at

Matrikelnummer: 9220676
Hetzendorferstrasse 93/6/8
A-1120 Wien, Österreich

Wien, im Mai 2003

Kurzfassung

Diese Dissertation stellt die XGuide Web Entwicklungsmethode vor. XGuide betont die termingerechte Entwicklung von Webapplikationen und garantiert qualitativ hochwertige Designdokumente und wiederverwendbare Implementierungen. Es unterstützt den vollen Lebenszyklus von Webapplikationen und deckt die Phasen Analyse, Design, Implementierung und Wartung ab.

Die zentrale Idee in XGuide ist die Einführung der bewährten Software Engineering Konzepte "Interface" und "Vertrag" in die Domäne des Web Engineering. Verträge legen die Anforderungen und internen Abhängigkeiten von Webseiten fest und fungieren als Spezifikationen für die folgende Implementierung. Um die parallele Durchführung von Implementierungstätigkeiten durch unterschiedliche Personen zu fördern, führt XGuide sogenannte mehrdimensionale Verträge ein, die die zeitgleiche Entwicklung von Teilen der Implementierung wie dem Inhalt, der grafischen Repräsentation und der Applikationslogik ermöglichen.

Zusätzlich zur gleichzeitigen Implementierung unterstützen Verträge auch die Definition von Web-Komponenten: wiederverwendbaren Fragmenten, die zu Webseiten zusammengesetzt werden. Eine Web-Komponente ist durch ihren Vertrag vollkommen spezifiziert und das Zusammensetzen von Verträgen gibt die Regeln für die Integration von Komponenten in Seiten vor.

Aufgrund der kurzen Innovationszyklen des World-Wide Web werden Web Entwicklungsmethoden ständig mit neuen Anforderungen konfrontiert. Eine formale Definition von Verträgen und deren Komposition bildet die Grundlage für ein offenes und erweiterbares Vertragsmodell in XGuide, das neue Anforderungen wie Zugriffskontrolle, Meta-Daten oder Geräteunabhängigkeit als getrennte Module realisieren kann.

Der XGuide Prozess verwendet einen modellbasierten Ansatz, der anfängliche Anforderungen schrittweise in Designdiagramme, Verträge und Implementierungskomponenten umwandelt. In der Wartungsphase werden alle Änderungen als Aktualisierung der Diagramme und Verträge formuliert, die sich letztendlich in der Implementierung wiederfinden. Diese iterative Vorgehensweise stellt sicher, dass alle Designmodelle mit der Implementierung konsistent sind und gewährleistet gut strukturierte Projekte und nachvollziehbare Änderungen.

Um den Einsatz von XGuide in echten Web Projekten zu unterstützen, entwickelten wir das XSuite Entwicklungswerkzeug. Das Ziel der XSuite IDE ist es, die Entwickler in allen Phasen des XGuide Prozessmodelles zu unterstützen. Grafische Designmodelle werden automatisch in Verträge umgewandelt und Assistenten bieten Hilfe bei der Erstellung von neuen Seiten, beim Zusammensetzen von Verträgen und bei der Installation der Webapplikation. XSuite setzt auf das generische Entwicklungsmodell von Eclipse auf und integriert eine Java IDE, ein System zur Versionskontrolle und einen Web Server in die eigentliche Web Entwicklungsumgebung.

Die Umsetzbarkeit des XGuide Prozesses und die Anwendbarkeit des XSuite Softwarepaketes wird anhand der Implementierung der Webapplikation für die Wiener Festwochen 2003 demonstriert.

Abstract

In this dissertation we propose the XGuide Web development method. XGuide focuses on the timely development of Web applications while guaranteeing high-quality designs and reusable implementation artifacts. It supports the whole life-cycle of a Web application and covers the analysis, design, implementation and maintenance phases.

The central idea in XGuide is to bring the well-established software engineering concepts of interfaces and contracts to the Web engineering domain. Contracts clearly state the requirements and internal dependencies of Web pages and act as specifications for a subsequent implementation. To support multiple activities being carried out in parallel by different people, XGuide introduces multi-dimensional contracts that enable the concurrent development of implementation concerns such as the content, the graphical appearance and the application logic.

In addition to the parallel implementation phase, contracts also enable the definition of Web components—reusable page fragments that get assembled to form the final Web page. A Web component is fully specified by its contract and contract composition defines the rules for embedding components into pages.

The short innovation cycles on the Web further require a Web development methodology to constantly cope with new requirements. In XGuide, a formal definition of contracts and their composition is the foundation for an open contract model that can integrate new concerns such as access control, meta-data or device independence as separate modules.

The XGuide process applies a model-driven approach to Web development that iteratively refines initial requirements into design diagrams, contracts and implementation components. After the initial deployment, XGuide directly maps maintenance and evolution tasks to updates of the design models (i.e., diagrams and contracts). Model updates trigger a new iteration of the XGuide process, i.e., follow the same contract-based, parallel implementation paradigm. This round-trip engineering ensures that models remain consistent with the implementation and guarantees well-structured and easy to trace projects.

In order to support the application of XGuide in real-world Web projects, we implemented the XSuite Web development environment. The purpose of the XSuite IDE is to support the developer in all phases of the XGuide process. Visual design models are automatically transcoded into contracts and wizards assist in creating pages, composing contracts and deploying the application. XSuite is built on top of the generic Eclipse framework and integrates a Java IDE, a version control system and a Web server into the actual Web development environment.

The practicality of the XGuide method and the XSuite tool suite is demonstrated in the implementation of the Vienna International Festival 2003 (VIF) case study.

Acknowledgements

It is my pleasure to thank the many people who made this thesis possible.

First and foremost I am greatly indebted to my beloved girl-friend Isabelle Schlager-Weidinger who accompanied me on the long way leading to this dissertation, accepted me working (too) long hours, and kept supporting me wherever possible. Many thanks also to my parents for encouraging me to strive for a dissertation and providing an environment I could resort to in order to relax and regain my strength.

I further offer my sincerest gratitude to my supervisor, Prof. Dr. Mehdi Jazayeri, who introduced me to research, gave me the possibility to pursue my research ideas, and kept me on the right track with his advice. The discussions with Prof. Dr. Gerti Kappel and the feedback I received from her and her group revealed new views on my work and helped me improve it. I deeply appreciate your collaboration.

Special thanks also go to Engin Kirda and Roman Kurmanowytsch for the many fruitful discussions, the visions we created, and the innumerable help in identifying research problems and shaping my ideas. Together we went through challenging times and solved many problems. Your company and friendship are most valuable to me.

I would like to thank Werner Wohlfahrter for his positive and encouraging comments and for proofreading this dissertation. Your close scrutiny revealed many inconsistencies and kept pointing out areas in need of additional explanation.

I owe a great deal to David Hunt, Mike Barnett, Clemens Szyperski and Arthur Watson at Microsoft XAF/Microsoft Research for sharing their ideas on contracts in software engineering with me. I benefitted so much from your experience and extraordinary knowledge.

Thanks also to the members of the Distributed Systems Group who formed a most enjoyable working environment. I am grateful for all I learned from you and will keep many memories of our extra-curricular activities.

Clemens Kerer
Vienna, Austria, May 2003

CONTENTS

1 Introduction **1**

1.1 Motivation 1

1.1.1 A Typical Web Development Scenario 2

1.1.2 The Problem Domain 3

1.2 Contract-Based Web Development with XGuide 4

1.3 Contributions 6

1.4 Structure of This Thesis 7

2 Web Engineering Review **9**

2.1 Terminology 9

2.2 The Architecture of the World Wide Web 11

2.3 The Communication Model of the World Wide Web 13

2.4 A Short History of Web Evolution 13

2.4.1 A First Server and Browser - The Web Infancy 14

2.4.2 The Web Gets Dynamic - The Childhood 15

2.4.3 Various Web Technologies Flourish - The Youth 17

2.4.4 Web Engineering - The Adolescence 18

2.5 Scratching the Surface of XML and Some Related Technologies 21

2.5.1 The eXtensible Markup Language - XML 22

2.5.2 Document Type Definitions - DTDs 23

2.5.3 XML 2nd Edition 24

2.5.4 The Extensible Stylesheet Language - XSL 27

2.5.5 XML Schema 29

2.5.6 Other X Technologies 31

3 Related Work **33**
3.1 A Taxonomy of Web Engineering Methodologies 34
3.2 A Discussion of Existing Approaches Towards Web Engineering 38
3.2.1 The Relationship Management Method - RMM 38
3.2.2 Analysis and Design of Web-based Information Systems 39
3.2.3 Object-Oriented Hypertext Design Method - OOHDM 41
3.2.4 A Schema-Based Approach to Web Engineering 42
3.2.5 SWM - A Simple Web Method 43
3.2.6 The Object-Oriented-Hypermedia Method (OO-H) 44
3.2.7 The Five-Module Framework for Internet Application Development . . . 45
3.2.8 The WWW Design Technique - W3DT 46
3.2.9 LifeWeb: An Object-Oriented Model for the Web 46
3.2.10 Conceptual Modeling and Web Site Generation using Graph Technology 47
3.2.11 The Web Modeling Language (WebML) 48
3.2.12 WebComposition: An Object-Oriented Support System for the Web Engineering Lifecycle 50
3.2.13 Synthesis of Web Sites from High Level Descriptions 51
3.2.14 The Extensible Web Modeling Framework (XWMF) 52
3.2.15 Strudel 53
3.2.16 WOOM - The Web Object Oriented Model 54
3.2.17 Comparison of the Presented Web Engineering Methodologies 55

4 XGuide - A novel Approach towards XML-based Web Engineering **59**
4.1 An Overview of the XGuide Methodology 60
4.2 Requirements Analysis 62
4.3 The Feasibility Decision 66
4.4 Conceptual Modeling and Design 70
4.4.1 Design In-The-Large 71
4.4.2 Design In-The-Small 80
4.5 Implementation Phase 81
4.6 Testing Phase 85
4.6.1 The Abstract State Machine Language - AsmL 87
4.7 Deployment Phase 89
4.8 Maintenance and Evolution 89
4.9 Conclusion 94

5 Contracts and Contract Composition 97
5.1 A Meta-Model for Web Applications 97
5.1.1 The Dexter Hypertext Reference Model 98
5.1.2 Modeling Web Application Architectures with UML 99
5.1.3 The XGuide Meta-Model for Web Applications 100
5.2 A Formal Model for Web Component Contracts 103
5.2.1 Contract Composition 106
5.2.1.1 Composition of Orthogonal Contracts 107
5.2.1.2 Composition of Dependent Contracts 107
5.3 XGuide Contracts - XContracts 110
5.3.1 The Structure Contract Concern 112
5.3.2 The Interface Contract Concern 112
5.4 Contract Composition 114
5.4.1 Composition of Structure Contract Concerns 117
5.4.2 Composition of Interface Contract Concerns 117
5.4.2.1 Output Interfaces 119
5.4.2.2 Input Interfaces 119
5.4.3 Composition of Dependent Contract Concerns 127

6 XSuite - An Integrated Development Environment for XGuide 131
6.1 The Eclipse Project 132
6.1.1 The Eclipse Extensibility Mechanism 134
6.2 The MyXML Web Publishing Framework 138
6.3 XSuite Conceptual Modeling 141
6.4 XSuite Eclipse IDE 143
6.4.1 The Concern Extension Point 144
6.4.2 The Technology Extension Point 146
6.5 Separation of Concerns with MyXML 147
6.6 JAXB - Java XML Data Binding 150
6.7 The Contract Cache 150
6.8 Generating Canonical XML from an XML Schema 151

7 The Vienna International Festival (VIF) Case Study 153
7.1 Analysis of VIF Requirements . . . 154
7.1.1 Discussion . . . 157
7.2 The Feasibility Decision . . . 158
7.3 Designing the VIF Web application . . . 160
7.3.1 Design In-the-Large . . . 160
7.3.2 Design In-the-small . . . 163
7.3.3 Discussion . . . 167
7.4 Concurrent Implementation based on Contracts . . . 169
7.4.1 Discussion . . . 171
7.5 Testing and Deploying the VIF case study . . . 173
7.5.1 Discussion . . . 176
7.6 Maintenance and Evolution of the VIF 2003 Web application . . . 176
7.6.1 Discussion . . . 178

8 Evaluating the XGuide Web Development Method 181
8.1 Experiences from the VIF Case Study . . . 182
8.2 XGuide for Development . . . 183
8.3 XGuide for Maintenance and Evolution . . . 185
8.3.1 Content Updates . . . 186
8.3.2 Layout Updates . . . 186
8.3.3 Application Logic Updates . . . 187
8.3.4 Page-Related Updates . . . 187
8.3.5 Navigation Updates . . . 187
8.3.6 New Output Formats . . . 188

9 Conclusion and Future Work 191
9.1 Analysis of this Dissertation . . . 192
9.2 Ongoing and Future Research . . . 194

Bibliography 197

Appendix 209

LIST OF FIGURES

2.1 The extended client-server architecture of the World Wide Web. 12
2.2 A sample CGI script implemented in the Perl scripting language. 16
2.3 A sample processing pipeline for the *chained processing* approach. 19
2.4 A sample XML document representing an order. 23
2.5 The document type definition for the sample order document. 23
2.6 Associating a DTD to an XML document using the `DOCTYPE` declaration. 24
2.7 Two XML documents with conflicting definitions for the `<address>` elements. 25
2.8 The sample XML document using namespace abbreviations. 26
2.9 The sample XML document overriding the default namespace. 26
2.10 The sample XML document using and overriding several namespace abbreviations. 26
2.11 The sample XSLT stylesheet to transform an order document into an XHTML page. 28
2.12 The result of applying the sample XSLT stylesheet to the order document. 29
2.13 A sample schema for the order document. 30

4.1 The XGuide Development Process 61
4.2 A simple page named 'Homepage' that has a navigational dependency to the simple page 'Search' that has additional requirements associated with it. 64
4.3 A legacy Web application for customer feedback is modeled as external page named 'Customer Feedback'. The 'Product Details' multi page could be used as a template for a product catalogue. 65
4.4 The initial XGuide requirements diagram for the Orange Juice, Inc. Web site. . . 65
4.5 A Web component for the header region of the Orange Juices, Inc. Web site. . . . 71
4.6 The updated icons for the XGuide elements including the *References* section. . . 72
4.7 The component web for the Orange Juice, Inc. Web site. 72
4.8 The 'application logic process' diagram element used by a search page and producing the search result. 73
4.9 The graphical representation of the proxy diagram artifact. 74

4.10 The interface dialog for the product details multi page indicating its input requirements. 75
4.11 A simple example demonstrating the definition of an output interface. 76
4.12 The application logic process matches the input/output requirements of the connecting pages. 76
4.13 The abbreviated notation for the conceptual model of the search example. 77
4.14 An example demonstrating the XGuide consistency checking algorithm. 78
4.15 Structure of an XGuide XML sitemap. 79
4.16 The basic structure of an XContract. 81
4.17 A simple XContract for a Web page. 82
4.18 The content concern for the contract in Figure 4.17. 82
4.19 The layout concern for the contract in Figure 4.17. 83
4.20 The application logic concern for the contract in Figure 4.17. 83
4.21 The XPage implementing the contract in Figure 4.17. 85

5.1 The structure of an XContract. 110
5.2 An XContract with concerns from different namespaces. 111
5.3 The structure concern of a sample XContract for a Web page. 113
5.4 The interface concern of a sample XContract for a shopping cart Web page. . . . 114
5.5 The structure of an extended XContract with concern composition operators. . . 116
5.6 The structure concern composition operator of a sample Web page that references a component contract. 118
5.7 The structure concern after the composition with the contract of the header component. 118
5.8 The component web for the sample scenario. 120
5.9 Sample scenario for the composition-by-addition approach. 120
5.10 A partial XContract demonstrating the syntax of the composition-by-addition composition operator. 121
5.11 Sample scenario for the composition-by-unification approach. 122
5.12 A partial XContract demonstrating the syntax of the composition-by-unification composition operator. 123
5.13 Sample scenario for the composition-by-adaptation approach. 124
5.14 A partial XContract demonstrating the syntax of the composition-by-adaptation composition operator. 124
5.15 Sample scenario for the composition-by-omission approach. 126
5.16 A snippet from a typical Web page offering a page identifier. 126
5.17 A partial XContract demonstrating the syntax of the composition-by-omission composition operator. 127

5.18 A contract for a simple navigation bar with access control information. 128
5.19 A page contract embedding the navigation bar contract. 129

6.1 A generic instance of the Eclipse platform. 133
6.2 The Eclipse Java IDE with the XSuite sources. 135
6.3 The plug-in architecture of the Eclipse platform. 136
6.4 A snippet of the *plugin.xml* manifest file for the XSuite application plug-in. . . . 137
6.5 The MyXML process. 139
6.6 A sample MyXML content page querying a database to display an event with a given identifier. 140
6.7 The extended Visio workspace for XGuide development. 142
6.8 The dependencies of the plug-ins constituting the XSuite IDE. 143
6.9 The Java interface for contract concerns. 144
6.10 The plug-in interface for new implementation technologies. 146
6.11 The modified UML package diagram for the XSuite IDE. 148
6.12 The MyXML generated interface for a sample page with input and output interfaces. 149

7.1 The requirements diagram for the programme and ticket ordering sections of the VIF 2003 application. 156
7.2 The snippet of the design diagram responsible for the programme overview, search and details pages. 161
7.3 The page from the design diagram that specifies the ticket ordering process. . . . 162
7.4 A fragment from the XML representation of the VIF sitemap. 164
7.5 The first page of the contract creation wizard of the XSuite IDE. 165
7.6 The contract template generated for the programme search component with specifications for the interface and the structure concerns. 166
7.7 A page of the contract composition wizard requesting composition information for the interface concerns. 167
7.8 The updated composition reference section in the contract that contains all composition operators. 168
7.9 The XPage creation wizard provides separate tabs for all concerns and the options to reuse existing implementation files. 170
7.10 The generated XPage acting as a container for references to the actual concern implementations. 171
7.11 The generated interface and factory of the programme search component. 172
7.12 The property page of the integrated Tomcat servlet engine. 174
7.13 The final programme overview page of the VIF case study. 175

LIST OF TABLES

3.1 Comparison of Web Engineering Methodologies. 56

4.1 The development cycles of the Apache Xerces XML parser and the Jakarta Tomcat servlet container. 91

4.2 A classification to determine the state of a Web application with regard to updates and extensions. 94

5.1 The artifacts of the XGuide Meta-Model for Web Applications. 102

6.1 XSuite extensions contributed to the Eclipse platform. 145

CHAPTER 1

INTRODUCTION

> The killer app will not be a shrink-wrapped program that sells millions.
> The killer app will be a Web site that touches millions of people and
> helps them to do what they want to do.
>
> Lou Gerstner

1.1 MOTIVATION

Did you ever think about how your life would be without the Internet? How much of your daily routine would change without electronic mail? How much harder it would be to access information without the World Wide Web?

Students, business people, parents, and children all over the world have email accounts to keep in touch with friends, relatives or business partners. Companies have Web sites to communicate with their customers or advertise and sell their products. Educational institutions provide teaching material on the Web, offer online training, and use email as the primary communication media with their students. But there is much more: the Internet effectively spreads into almost all domains of our daily life. The creation of terms such as eCommerce, eLearning, eSupport, eProcurement or eGovernment clearly emphasizes this fact.

We keep talking about the Internet but what we actually refer to is usually either electronic mail (email) or the World Wide Web (Web, WWW). These two services by far outnumber any other application on the Internet. This thesis focuses on the World Wide Web.

On the Web, we distinguish *Web sites* and *Web applications* [5, 37]. A Web site's main intent is information dissemination. It presents structured information in a mostly static way, i.e., does not support user interaction. News portals or product catalogues are examples of Web sites. Their extensive information is mainly for viewing and users cannot interact with the system other than following pre-defined hyperlinks.

A Web application, on the other hand, uses the World Wide Web as user interface for a back-end software application. In Web applications, user interaction and business processes outrank the information dissemination aspect. The back-end software of a Web application, i.e., its functional behavior, is also called the *application logic*. An online shopping cart application that lets the user select items, stores the contents of the cart, validates payment information and processes the order is an example of a Web application; its functionality is implemented by a custom application logic.

1.1.1 A Typical Web Development Scenario

Imagine you own the company *Orange Juice, Inc.* and decide to create a Web application to represent your company on the Internet (the World Wide Web to be more precise).

First you make a list of all the information and functionality that should be accessible via your Web site. Then the graphics designer proposes various graphical designs, e.g., based on your corporate identity policy. The next question is how to implement the graphical design templates, i.e., what technology to use and how to integrate the content with the layout templates. This depends heavily on the programmers and their know-how. Often easy-to-use scripting and template languages are used for this purpose (e.g., Microsoft's Active Server Pages (ASP), Java Server Pages (JSP), Perl, PHP, etc.). Eventually, the content managers can start to generate the content for the Web application in the appropriate form. When all the content is available, it can be integrated with the application logic and layout templates and the site can go online.

Two months after your site went online, the analysis of the Web server log files indicate that a large number of visitors is interested in the site. As a consequence you decide to upgrade your service to contain an e-commerce component to directly sell your products over the Internet. Thus you iterate again through the above steps, i.e., talk to the graphics designer, have the programmers implement the application logic based on the design templates and, finally, ask the content managers to provide the content.

In the process of extending your Web site, you realize that it would be nice if you could reuse the existing product catalogue with the shopping system, the contact information on the order pages, or the customer database with the feedback facility. Unfortunately, many of today's implementation choices do not support this kind of separation and reuse of components. This is because the layout definitions, the content information and the application logic are all scattered across various page and fragment definition files, and are possibly even stored together in the same (content) database.

This scenario still does not cover the so-called maintenance task, i.e., continuous content updates, addition and removal of special offers, correction of bugs in the application logic or integration of new functionality.

In the end, what started as a small Web site development project grew to a complex, hard to maintain Web site with lots of dependencies and a never ending amount of undocumented updates and changes. And it keeps growing. It is not unlikely that after a year or two your Web site has become so unmanageable that you decide to start from scratch and build a new Web site to avoid the maintenance and update nightmare.

1.1.2 THE PROBLEM DOMAIN

Right from the beginning, Tim Berners-Lee pointed out that it is important that information on the Web can be edited as easily as it can be viewed [17, 32]. With visual Web editors and site management tools this vision became reality to a large extent for moderate-sized, HTML-based Web sites with only little application logic.

The above scenario, however, already demonstrated that today's Web applications do not have much in common with the simple, pure HTML pages of the early days of the Web. Scripting languages extend the functionality of HTML on the client side, dynamic pages are generated by server-side programming, and personalized Web sites remember user preferences or behavior. But the dynamic generation of Web pages is not the only reason for their tremendously increased complexity.

In addition to the original idea of information dissemination via the Web, other aspects such as an attractive and intuitive look-and-feel, a clear navigation concept, up-to-date and correct information, security and access control, transactional behavior, database connectivity, and back-end business processes (e.g., integration of legacy applications) play important roles in state-of-the-art Web applications.

The creation of such Web applications is not a trivial task and is often compared with a full-fledged software project. It requires careful planning and a wide range of expertise [127] to successfully deploy a new Web application. Additionally, a considerable effort continuously goes into maintenance and evolution activities once the application is deployed.

The varying goals of the different stakeholders involved in a Web project, further contribute to its complexity. From the customer's point of view the budget of the project is usually the main concern; the project manager focuses on the project duration and the available resources; graphics designers are interested in the visual appearance of the application; programmers usually are concerned with the functional aspects and the integration of content, layout and application logic; other stakeholders such as testers or marketing staff have yet another agenda.

Also in analogy to software projects, time-to-market is an important aspect in many Web projects. In the above scenario, the development process is virtually serialized: the layout designers design the templates, then the programmers take over, and eventually the content managers provide the actual content. Then it is again the programmer's task to integrate the content with the layout templates and make it work together with the application logic before deploying the Web site. If we could parallelize this process we could significantly reduce the project duration which often also means a decrease in the total development costs.

With the explosion of possibilities and technologies on the Web over the last years, however, Web development did not get easier or faster. Instead the development of a Web application soon became a somewhat chaotic and often ad-hoc process lacking systematic techniques and methodologies. As a consequence, Web applications became increasingly difficult to maintain or evolve, changes to the structure or the layout of a site were not possible without a great amount of work, and performance decreased. Ginige et al. [68] use the term *Web crisis* to describe this situation.

Two important developments stem from this unfortunate situation: (i) the *Web Engineering* discipline was founded to overcome the problem of ad-hoc development. (ii) with XML and

its related technologies, a whole set of new technologies and languages were standardized to overcome the limitations of HTML.

1.2 Contract-Based Web Development with XGuide

Given the wealth of existing Web engineering technologies and tools, the challenge is not so much how to develop a new Web site or how to use XML in doing so—though such an undertaking is still far from being trivial. The real challenge in Web engineering is how to develop Web sites and applications in a way that quality factors such as maintainability, performance, extensibility, device independence, development time, or flexibility are taken into consideration.

Before we can achieve these goals, we need to master the complexity in Web engineering projects, be able to define and measure the above mentioned quality criteria, and have to understand the direct and indirect dependencies among the involved artifacts. To this end, this thesis introduces the notion of a *contract* to the Web engineering domain.

The idea of contracts originated in the domain of software engineering. There a contract defines the public interface of a component and states the requirements that need to be satisfied before using the component as well as guarantees on the result of any operation on the component. Unlike contracts in software engineering [8,9,111,112], the notion of contracts in this work does not operate on the type or method level but deals with the characteristics and constraints of the components involved in Web development. This includes, among others, specifications for the structure and the data model of the content, the required interfaces to the applications logic, security properties, navigation modeling, etc.

From a high-level point of view, Web contracts can be seen as agreements among and interfaces between all stakeholders involved in the development of the artifacts that eventually make up the Web application (e.g., content managers, graphic designers, programmers, etc.). More concretely, a contract provides specifications for all these artifacts and makes dependencies explicit. As opposed to the software engineering domain where contracts exclusively deal with software artifacts, Web contracts have to deal with multiple dimensions (e.g., the content dimension, the layout dimension and the application logic dimension of a Web page).

Web contracts not only give us a specification technique for Web artifacts, they also provide the basis for four important concepts in order to meet the challenges presented at the start of this section: strict separation of concerns, composability, parallel development, and flexibility:

- **Separation-of-concerns**. The concept of separation of concerns enables the better understanding of complex systems. Contracts define the concerns involved in the development of a Web page. They separate the concerns and provide a specification of each concern including its dependencies. Typical concerns of a Web page are the content, the graphical appearance (i.e., the layout) or the functionality (i.e., the application logic) of a page. Since concerns are separated, they can be easily reused on other pages further decreasing development effort and avoiding potential inconsistencies.

- **Composability.** Web pages are the prevailing units of perception on the Web. Though they are presented as atomic entities to the user, they are frequently structured internally into multiple areas. A navigation area, a header area and a content area are typical examples. Contracts support the definition of such page fragments and their composition into Web pages. Thus not only can the concerns be reused but whole page fragments (e.g., a header fragment) can be reused on many pages. Unlike other object- or component-based Web systems, a contract composition language explicitly states the composition relationship among components and pages.

- **Parallel Development.** Existing Web development methods are inherently sequential. The sample scenario above demonstrates this fact. The graphical design, the application logic development, and the content creation and integration are performed one after the other. Creating these (separate) concerns in parallel can significantly reduce the project duration and for this reason the development costs. Contracts provide a clear specification for each concern and its interface to other concerns. As a result, all concerns can be developed in parallel. This is similar to interface-based programming where an interface specifies all external behavior of a component and decouples any concrete implementation.

- **Flexibility.** A Web application is a continuously changing system. Maintenance and evolution of Web applications are important aspects of Web engineering. Contracts support maintenance and evolution in that they clearly identify the affected concerns and isolate the potential impact on other pages or components. Also new concerns such as security, access control or navigational design can be added to a contract with no or minimal interference with existing concerns. In some cases, even the implementation technology of a concern can be changed transparently.

Our vision for the future of contract-based Web engineering is that there are two phases: first, the *contract phase*—an analysis and design effort which results in a set of contracts describing the Web site. Only then the *realization phase* starts in which the contracts get implemented. When all aspects of all contracts are fully implemented, the development is finished and the Web site is deployed.

If supported by appropriate development tools, we argue that contract-based Web development can significantly improve Web engineering in terms of a shorter development time, easy reuse of existing components, enhanced flexibility and maintainability (by clearly separating the different aspects of the Web application), and automatic consistency checking with respect to the specified contracts.

1.3 CONTRIBUTIONS

Management by objectives works if you first think through your objectives.
Ninety percent of the time you haven't.

Peter F. Drucker

The objective of this thesis is to introduce the notion of *contracts* into the Web engineering domain. Contracts provide specifications of all artifacts in the development process, enforce strict separation-of-concerns, enable parallel development, support composition of page fragments into Web pages, and facilitate seamless evolution scenarios.

To fully exploit the concept of Web contracts, we present *XGuide*, a contract-based development methodology for Web applications. XGuide provides full life-cycle support for the development, maintenance and evolution of Web applications. It conceptually models a Web application as a set of contracts (i.e., specifications) that themselves stay independent of any concrete implementation technology. Subsequently the conceptual model is refined and implemented in the technology of choice. The high reuse potential of many artifacts, the model-driven, fully parallel implementation phase and the structured maintenance and evolution scenario are direct benefits of applying the contract concept. An additional advantage of using contracts is that implementations can be validated against the contracts, i.e., we can check whether our Web site conforms to the specifications given in the contracts.

To substantiate the concepts used in the XGuide development method, we devise a formal model for Web contracts and concerns. The model can express independent as well as dependent concerns. Dependent concerns cannot exist alone but always depend on another concern. Further, the aggregation of concerns into contracts and the composition of contracts into larger contracts is presented.

This thesis further contributes the *XSuite* tool suite, an open, extensible development environment supporting all phases of the XGuide methodology. A visual modeling environment based on the Eclipse [142] framework supports the creation of the conceptual models that are in consecutive steps transformed into fine-grained contracts and concerns. These specifications are translated into a concrete implementation that can be directly deployed on the target platform.

A case study illustrates the use of the XGuide development methodology and the XSuite tool suite.

1.4 Structure of This Thesis

The remainder of this thesis is structured as follows.

Chapter 2 points out the fast evolution from the first Web servers to today's world-wide information network. It discusses the problems at the various stages of this development that resulted in the creation of the Web engineering discipline. Further it gives a brief overview of XML and its related technologies that are subsequently used in this work.

Chapter 3 presents a classification of Web engineering methodologies based on seven characterization properties. It then discusses related Web engineering work and evaluates each approach with respect to the given classification scheme. The chapter concludes with a comparison of existing methods and demarcates XGuide from the approaches taken in existing work.

Chapter 4 discusses in detail the seven phases of the XGuide Web engineering methodology and the diagrams, notations and terminology used. The Orange Juices, Inc. Web application introduced above is used as running example to demonstrate the approach.

Chapter 5 focuses on the concept of contracts. It defines what the semantics of a contract is and how it is represented. Separation of concerns on the contract level and contract composition operations rely on a formal model of contracts and contract concerns introduced in this chapter.

Chapter 6 describes the architecture and design of XSuite. The integrated development environment supports all steps of the XGuide development process and exploits Eclipse's potential to offer usability features such as automatic deployment, wizard dialogs and creation of contract and implementation templates.

Chapter 7 presents the Vienna International Festival case study. It demonstrates how we used XGuide and XSuite for the analysis, conceptual design and implementation of this Web application.

Chapter 8 evaluates the XGuide methodology and XSuite tools with respect to other approaches and the lessons we learned from the case study.

Chapter 9 concludes the thesis and summarizes the major contributions of this work. It also gives an outlook on potential future extensions and the integration of additional concerns into XGuide.

CHAPTER 2

WEB ENGINEERING REVIEW

We've all heard that a million monkeys
banging on a million typewriters
will eventually reproduce the entire works of Shakespeare.
Now, thanks to the Internet, we know this is not true.

Robert Wilensky

This chapter first defines the meaning of the vocabulary used throughout this thesis and illustrates the different architectures used on the Web. It then gives an overview on how the Web evolved since its birth at CERN in 1989 and presents the technologies that were introduced as the Web grew and are still commonly used to implement Web sites today.

As we move through the various stages of evolution of the Web, we point out problems in today's Web development practice. This discussion results in the introduction of the research field of *Web Engineering*, its mission statement and its goals. The chapter concludes with a walk-through of some fundamentals of XML and its related technologies and standards. They build the foundation not only for the XSuite approach but are the successors of HTML on the Web and keep spreading into other domains such as (cross-platform) data exchange or information storage.

2.1 TERMINOLOGY

Understanding the meaning of the terms used in any given context is a crucial requirement for effective and unambiguous communication. Unfortunately quite some confusion exists when it comes to frequently used Web engineering terms such as Web site, Web application or Web service. This section presents what we understand by these terms in the context of this thesis.

- **Web Page.** A Web page is a set of information items which are perceived as an indivisible entity by the client application or browser. In the case of HTML, an HTML page would be a Web page; in the case of the Wireless Markup Language (WML) [148] a card in a WML deck is referred to as a page. This term is sometimes (mis)used for 'Web site', i.e., meaning all the pages available through a given base URL.

- **Web Site.** A Web site is a collection of static and/or dynamically generated Web pages that form a unit in terms of the content they provide, often share a common look-and-feel, and are available through the same base URL.

- **Web Application.** A Web application is similar to a Web site in that it also presents related information in a uniform graphical layout. The focus of Web applications, however, lies in the application logic (functionality) offered via the Web. A Web application can be seen as a software application or business process leveraging the Web as a new type of user interface. In some definitions, a Web application is even characterized as a software application that is downloaded via the Web and executed on the client not taking any server-side processes into account. Web sites, in contrast, focus on being an information system and not on the application logic on the back-end or the client device. Since the distinction between Web sites and Web applications is not always clear to make, there exist hybrid forms where the information system and the back-end processes are equally important. A similar definition is found in [37]:

  > "In this article, a Web application will be loosely defined as a Web system (Web server, network, HTTP, browser) in which user input (navigation and data input) effects the state of the business. This definition attempts to establish that a Web application is a software system with business state, and that its front end is in large part delivered via a Web system."

- **Web Service.** The term *Web service* is definitely one of the most over-used terms in the Web arena. In the early days of the World Wide Web, many researchers and practitioners used it as a synonym for either a Web site or a Web application. As such a Web service was a very generic term. More recently, the term was redefined in the context of machine-to-machine services. These services exchange machine readable information utilizing Web technology; the most prominent representative to date is the *Simple Object Access Protocol* (SOAP) that communicates via XML messages which are (usually) transmitted over HTTP. For the remainder of this work, we restrict the term 'Web service' to the latter meaning, i.e., a machine-to-machine communication on the basis of XML messages. This is sometimes also called *browser-less access* to a service, indicating that there is no visual client interface and the information is further processed by the requesting machine.

- **Web Development.** In the scope of this thesis, we understand 'Web development' as the actual implementation process of a Web site. This does not include requirements gathering, domain analysis, or other phases such as design, maintenance or evolution frequently found in software and Web engineering methodologies.

- **Web Engineering.** We use the term 'Web Engineering' as defined in [56]:

  > "Web Engineering is the application of systematic, disciplined and quantifiable approaches to the cost-effective development and evolution of high-quality applications in the World Wide Web"

  This includes all the activities involved in the planning, design, implementation, deployment, maintenance and evolution of a Web site or application. As a consequence, Web engineering effectively is a superset of the activities associated with Web development.

2.2 THE ARCHITECTURE OF THE WORLD WIDE WEB

You affect the world by what you browse.

Tim Berners-Lee

Right from the beginning, the World Wide Web was designed as a client-server system. Clients (usually Web browsers) access the content of a Web server using Uniform Resource Locators (URLs) that uniquely identify any resource on the Web. Such a URL contains (among other information) the Web server's name, the protocol to contact it, and the name of the requested resource (see [21] for further details). The server deals with incoming requests by delivering the requested resource to the client. Over time this simple architecture was slightly extended to keep up with the increasing demand; client-side caching, proxy caches and Web server clusters are the most prominent extensions of this kind. Nevertheless the basic client-server principle remained intact.

Modern Web sites are also described as three tier or multi tier architectures. This categorization has its origins in the server-side separation of many Web sites in a Web server and a data repository/database layer. This again eases the development and maintenance of large or complex Web sites, but leaves the underlying client-server architecture untouched.

Figure 2.1 depicts the extended client-server architecture of the World Wide Web. Web browsers form the client-side of the system. These clients use client-side caching to improve performance and can use shared proxy caches to broaden the effectiveness of the cache from a single client to a cluster of clients. Web servers represent the server-side. A Web server frequently serves files from its local filesystem and collaborates with dedicated database servers that host the content repositories. To improve the response time, Web servers can be grouped in Web server clusters. A load balancer distributes the incoming requests on the separate servers improving performance and availability.

In general, Web sites and Web applications are perfect examples of distributed applications as outlined in [138]. Tanenbaum et al. classify applications according to where their processing

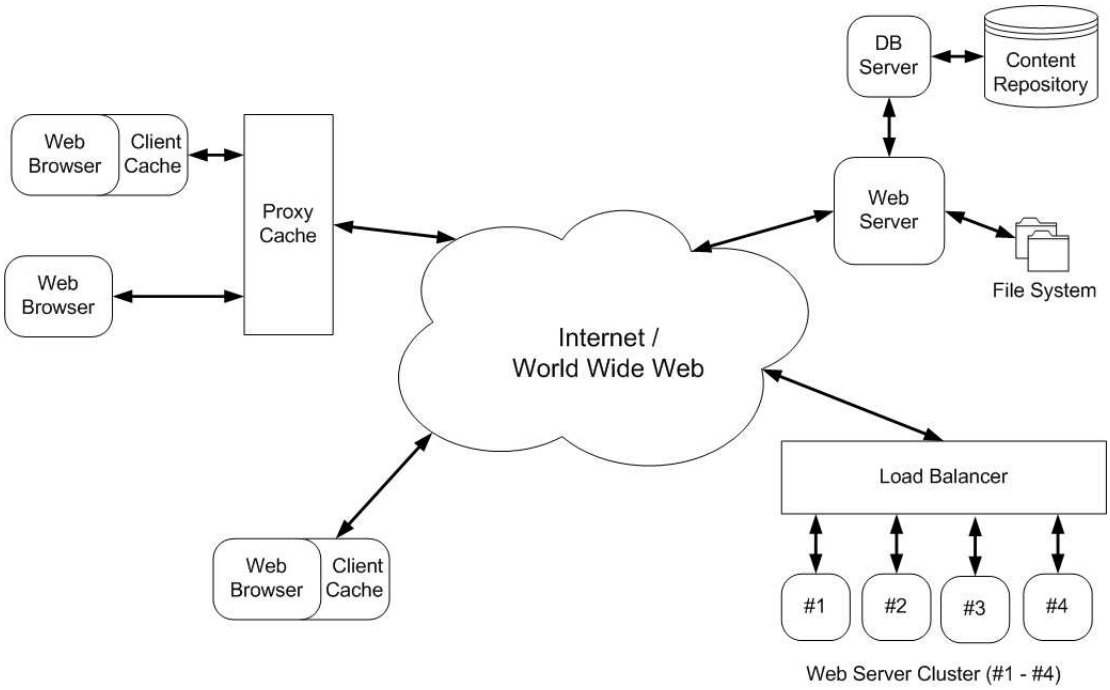


Figure 2.1: The extended client-server architecture of the World Wide Web.

happens. Some Web applications require the client to perform a large share of the workload; especially if the application requires a rich user interface, solutions based exclusively on HTML are often insufficient. A technical solution to this problem was introduced with Java applets that allow Java code to be executed on the client. They assume, however, a Java virtual machine being present at the client device and the client's ability to download executable code at runtime. Another example is Macromedia's Flash technology that facilitates full-fledged multimedia animations but require the client to run the corresponding Flash player.

While we can require capable clients to be responsible for many tasks, there are also many reasons to support thin clients: you do not have to upgrade all the client devices to new versions of the application, more devices can be supported easily, and today's mobile devices with limited battery and processing capabilities benefit in terms of a longer runtime. Though these devices are likely to become more and more powerful, their relative thinness compared to other (stationary) devices will remain. For the purpose of this thesis, we focus on the basic working of the Web as a client-server architecture and on keeping the client-side as thin as possible for the reasons given above. To fully exploit more capable clients, XSuite extensions can be introduced that take client profiles into account to provide a richer user experience, increase the performance or better distribute the processing load.

2.3 The Communication Model of the World Wide Web

The communication model on the Web is equally simple: client and server communicate via a TCP/IP connection and as soon as a client request is completed, the connection is terminated. The details of this synchronous communication protocol are specified in the HyperText Transfer Protocol (HTTP) [20, 66]. HTTP is an ASCII-based protocol on top of TCP that transmits requests and responses enriched by a message header that carries additional status and meta information. Over time, HTTP evolved only marginally to support new requirements such as virtual hosts (i.e., running multiple servers with varying names on the same physical machine). As mentioned before, HTTP is a state-less protocol, i.e., the server does not maintain state on behalf of the client. Thus a server could never identify subsequent requests of the same client as related. While this keeps the server simple and increases its performance, it is clearly insufficient in the context of, for instance, e-commerce applications where clients can create shopping carts and, at some later time, order all articles in the shopping cart. This means that the server must be able to relate a request to a previously created shopping cart and the items in it. As a consequence, clients are required to store session information themselves and retransmit it to the server with each request. Two approaches are widely used: URL re-writing and cookies. Using URL re-writing, the URLs in the response to a client request are dynamically modified to also include the client state. Cookies, on the other hand, are stored by the client and transmitted as header information of subsequent requests to the same server. There were many discussions whether a stateful protocol would have been superior to HTTP; the success of HTTP and its global deployment make these discussions moot. Since today's Web servers support URL re-writing and cookies transparently, we take them for granted in our further discussions.

Summing up we see that the Web is based on a simple client-server architecture with a text-based communication protocol. The simplicity and extensibility of the original design were key criteria for the success of the Web. In the next sections, we give an overview of the evolution of the Web and the used technologies that results in the discussion of some major problems traditional Web development suffers from.

2.4 A Short History of Web Evolution

It's [the Internet] like the flu -
it just spreads like crazy.

Jack Welch

The origins of the idea of hypertext can be traced back to the 1940s (see [18]). The World Wide Web itself was 'born' in 1989 at the CERN laboratories. Tim Berners-Lee then circulated

the paper "Information Management: A Proposal" for comments and wrote "HyperText and CERN" in which he proposed the establishment of a global hypertext space. The following historical developments are presented here based on information available from the World Wide Web Consortium (W3C) [2, 32] and Tim Berners-Lee [17, 19].

2.4.1 A First Server and Browser - The Web Infancy

In 1990, Tim Berners-Lee got the go-ahead for pursuing his idea and implemented the *WorlDwidEweb* program—a 'What You See Is What You Get' Web browser and editor. It is remarkable to note that since then, Web clients were intended not only for viewing but also for editing Web documents. In fact, it should be easy for everybody to edit documents on the Web.

The first Web server at CERN initially contained mainly material about the Web itself (e.g., the specifications for HTML, HTTP, URLs, etc.) to help spreading the knowledge of how to run or implement a Web server and browser. More browsers for other platforms eventually appeared. In the first three years the load of the first Web server increased steadily by a factor of 10. When academia and industry were taking notice, Tim Berners-Lee decided to found the World Wide Web Consortium (W3C) to coordinate the efforts. According to him,

> "The Consortium is a neutral open forum where companies and organizations to whom the future of the Web is important come to discuss and to agree on new common computer protocols. It has been a center for issue raising, design, and decision by consensus, and also a fascinating vantage point from which to view that evolution." [19]

Here are some of the major development steps in catchwords: in December 1991, the first Web server outside Europe was installed (by Paul Kunz at the Stanford Linear Accelerator Center (SLAC)); in November 1992, a list of 26 reasonably reliable servers was published; in March 1993, the Web traffic (HTTP on port 80) on the NSF backbone was measured to be 0.1 percent; in September 1993, the Web traffic increased to 1 percent of the NSF backbone traffic; in October 1993, about 200 Web servers exist; in May 1994, the first World Wide Web conference was held at CERN and is referred to as the 'Woodstock of the Web'; in June 1994, 1500 Web servers exist; in October, the World Wide Web Consortium is founded.

A more complete overview of the history of the Web can be found in the references cited above and Gromov's article "History of Internet and WWW: The Roads and Crossroads of Internet History" [73].

With the increasing popularity of Web sites and HTML, developers soon started to demand language extensions to deal with rendering related information such as fonts, colors, margins, etc. It was already then that the abuse of HTML for layout-specific tasks started. A prominent example is a button with round corners. To achieve this in HTML, developers used a 3 x 3 table where the four corner cells contained a little picture simulating a rounded edge. If the background color of the table cells and the color used in the images is the same, the desired impression is

achieved—at the cost of polluting the HTML code with tables and images solely contributing to the layout.

Cascading Stylesheets (CSS) became a W3C recommendation in 1996 and support the specification of layout properties such as fonts, font sizes, colors, etc. externally to the actual HTML code. Support for CSS version 1 is built into all popular browsers today. A more powerful second version of CSS (CSS 2) extends the original specification supporting adding of text, a selection mechanism for elements and multiple classes of devices. Unfortunately, CSS 2 is still not fully implemented by existing browsers; even worse, some features are implemented differently across browsers hindering the successful, large-scale deployment of CSS 2.

2.4.2 THE WEB GETS DYNAMIC - THE CHILDHOOD

Soon predefined, static Web pages alone were not sufficient. User input processing and dynamic page creation were needed to implement features such as search engines or feedback forms. The Common Gateway Interface (CGI) was the solution. Up to that time, a Web server mainly located the requested file on the harddisk and returned its content to the client. With the implementation of CGI-capable Web servers, requests could be dispatched to arbitrary other processes running on the Web server and client parameters could be passed to such processes.

Now any software program could be made responsible for processing a Web request and responding with an appropriate HTML document. With the integration of databases as content repository for dynamic Web page creation, a whole new class of Web sites appeared that could deliver up-to-date, user-tailored information; and this number continues to grow since then. In the year 2000, a report based on real Internet traffic estimated about 40 percent of all page requests to go to dynamically generated pages.

To avoid the performance drawback of the original CGI proposal which required a new process to be created for every request, similar approaches were introduced which reused processes or implemented multi-threaded solutions. Furthermore, special Web server modules exist for many Web servers and programming languages that allow the handling of dynamic requests within the Web server process. Perl scripts were the dominant language for implementing CGI interaction for quite some time. Today many other languages are also widely used with Java in the lead.

An important characteristics of CGI solutions is that the CGI script or program is responsible for creating the HTML response page sent to the client. A typical CGI script in Perl is shown in Figure 2.2.

The script generates a simple HTML page and uses the input parameter *keyword* to query a database and search for news items with the given keyword in their titles. The response contains a table listing all items found and provides links to each item based on the item's identifier.[1]

The major drawback of solutions purely based on CGI is that the program handling a request is responsible for generating an HTML page or fragment. Thus the HTML definitions are mixed

[1] More sophisticated error handling and extended functionality (e.g., create a different response page if no news items matching the query were found) were omitted for space reasons.

```
#!/usr/bin/perl
use CGI; use DBI;

$query = new CGI;
$keyword = $query->param("keyword");

$sql_command = "SELECT id, title FROM News where title like '%$keyword%'";
$dbh = DBI->connect("DBI:mysql:WWW:dbhost","login","password");
$sth = $dbh->prepare($sql_command) or die("Something went wrong!\n");
$rv = $sth->execute or die("Can't execute statement...");

print "Content-Type: text/html\n\n";

print "<HTML> <BODY> <CENTER><H2>Search Results</H2></CENTER>
  <TABLE>
   <TR> <TD> <B>Title</B> </TD> </TR>
";

while (@row= $sth->fetchrow_array) {
 $id = $row[0]; $title = $row[1];
 print "<TR><TD><A HREF=\"/cgi-bin/news.pl?id=$id">",$title,"</TD></TR>";
}

print "</TABLE> </BODY> </HTML>";
```

Figure 2.2: A sample CGI script implemented in the Perl scripting language.

with code fragments and often scattered throughout multiple programs. Changes to the layout, integration of new features and site maintenance became extremely difficult and error-prone.

2.4.3 VARIOUS WEB TECHNOLOGIES FLOURISH - THE YOUTH

Based on the experiences with CGI-based Web development, people around the world started to implement tools that helped them to overcome the problems with CGI-based development. In a first step, modules in various languages for better and easier handling of HTTP requests appeared on the scene; then people started to refactor the page creation process and extract values that appear on many pages (e.g., store the value of the background color in a globally accessible variable).

More promising approaches were based on so-called templates or includes. An early example for such an approach are *Server Side Includes (SSI)*. An SSI-aware Web server scans the HTML response it sends to the client for a predefined label and replaces the label with the content of the corresponding server side include fragment. By this means, a common header or footer HTML fragment can be inserted into all pages. The limitation of this approach is that the inserted page fragments can only be static HTML text. Server side includes were targeted at Web developers who build mostly static HTML Web sites; when doing dynamic page generation, such a functionality can be easily mimicked.

Another widely used approach uses the notion of *templates*. Templates in this context mean HTML pages that contain additional information such as hooks for some processing tool or application logic to be executed before delivery to the client. The *Java Server Pages* or Microsoft's *Active Server Pages* technologies are widely used instances of this idea. Many more template-based technologies exist but a profound discussion and comparison is beyond the scope of this work.

More sophisticated versions of this idea are implemented in tools such as WebMacro [51] or HTML++ [11] and support composition of reusable page fragments which are not required to be static any more.

Another group of tools focuses on making the creation of Web pages as intuitive and easy as possible for the end user. They typically support WYSIWYG editing of HTML pages, have property dialogs for colors and fonts, and sometimes even have site maintenance functionality such as maintaining link consistency when moving pages built into them. Frontpage, Dreamweaver or GoLive are some popular tools in this category.

A completely orthogonal approach is pursued by the Hyper-G/Hyperwave [106] project. After an initial evaluation of the Web (as it existed at that time), a better structured information base, a solution to the 'dangling link' problem, access control and support for multiple languages formed the project objectives. A Hyper-G server thus classifies the documents it hosts according to so-called collections—sets of documents that are grouped by topic. Collections form a hierarchy and documents can belong to more than one collection. The resulting information spaces guarantee to only provide related documents. Since a Hyper-G server takes care of storing documents, consistent appearance, access control and link consistency can be guaranteed. Link information in Hyper-G is not stored within the documents but in a separate link base; as a result

not only text documents but arbitrary (e.g., multimedia) documents can act as links or be targets for hyperlinks.

A final group of tools increasingly used in Web development is formed by so-called *application servers*. Application servers provide development frameworks that offer services such as persistence management, transaction handling, security mechanisms or transparent database connectivity. While application servers can be used for general purpose software development, they are often bundled with a Web server and some template technology. As such they provide a tightly integrated development environment for Web applications and are used as middleware layer in between the Web server and the back-end processes.

The benefits of an application server mainly pay off in large projects taking into account the additional overhead of learning the capabilities of the specific application server. An additional problem of this approach is that commercial application servers tend to be expensive and might even lock customers in since migrating from one application server to another (even if based on the same technology) from a different vendor is rarely fully supported. The three major application server technologies today are SUN Microsystem's *Enterprise Java Beans (EJB)* specification, Microsoft's counterpart based on COM+ and the Microsoft transaction server (MTS), and the only slowly advancing CORBA Component Model (CCM) based systems. The CCM and EJB specifications share many basic concepts, thus the CCM can be seen as extension of the EJB model and implementations of these technologies are expected to integrate easily.

Though the above technologies are widely deployed today, they all suffer from a shared architectural flaw: the content is directly generated in the target language. This is usually achieved by *nested processing*, i.e., the page generation process is based on calls and sub-calls to software modules. The problem here is that the output is generated by appending the result of each sub-call to the final page. Neither the various parts nor the whole page can later be re-processed to target other output formats or customize content depending on user or device profiles. All that changed when XML and XSL entered the arena of Web development.

2.4.4 Web Engineering - The Adolescence

With XML and XSL finally a strict separation of content and layout can be achieved. XML focuses only on the structure of the content and supports arbitrary XML vocabularies. XSL consists of XSLT and XSL-FO. XSLT is a powerful transformation language to transform an XML input document into an output document. The output document can again be an XML document or contain plain text, HTML, LaTeX, etc. XSL-FO is a page-oriented formatting language that supports publishing-oriented concepts such as page masters, page sequences, headers, page numbers, etc. Soon several XML-based development tools including Cocoon [107] and MyXML [84,85,89,93] became available. These tools support a strict separation of content, layout and application logic and use *chained processing* as opposed to nested processing. Chained processing means that a document is passed through a series of processing steps before it is delivered to the client (see also [28]). Thus the generated content can be processed multiple times and in different ways depending on the client's capabilities or user profiles. Figure 2.3 shows a sample pipeline for chained processing. A request is first extracted and its parameter decoded.

Next the application logic processes the input parameter and decides what content to include in the response. The content is subsequently generated. The content generation can trigger further application logic and content generation tasks. A sequence of transformation steps transforms the final content into the appropriate output format or markup language. Eventually, the server's reply is serialized and encoded and returned to the client. There the response is deserialized and rendered. Subsequent user interactions cause further requests to the server and the process starts anew.

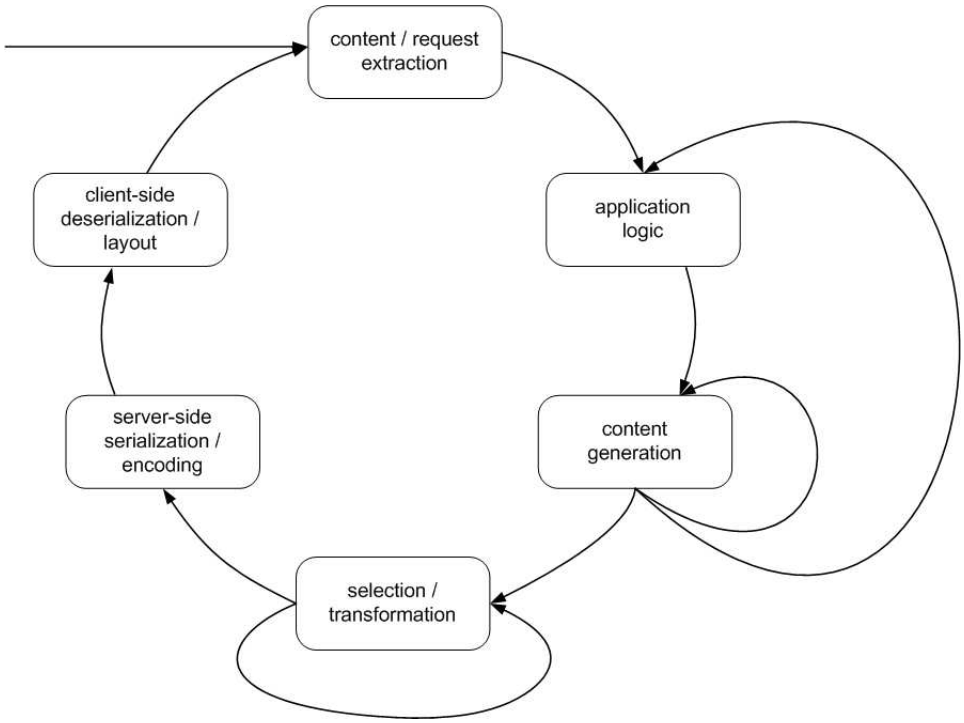


Figure 2.3: A sample processing pipeline for the *chained processing* approach.

Our experience with XML-based Web engineering as presented in [86, 94] revealed both positive and negative effects of deploying XML technology on the Web. One drawback was that learning of XML and XSL concepts was not easy for the developers. The strict separation of content and layout was misunderstood and misused by the developers. For example, there was a general tendency to mark content using XML tags such as <left>, <right>, <middle>, etc. that actually reflected the layout design requirements and did not describe the content. Another frequent misconception was that content management could directly benefit from XML. Content managers are used to a rich text-editor environment to manage content and available XML editors, though comfortable for XML editing, could not provide the same user experience. Thus we built custom content management interfaces on top of XML. Finally, graphics companies were used to creating design templates using WYSIWYG HTML editors rather than XML and XSLT. As a result, the translation of the HTML templates into appropriate XSLT stylesheets increased the overall development effort. In the long run, these negative experiences can be expected to

diminish as XML technology spreads and appropriate editing tools become available.

Even taking above problems into account, the advantages of using XML technology easily outweigh the disadvantages. Foremost, the strict separation of content and layout achieves a high layout flexibility that makes it easy to change the layout specifications and to support multiple output formats. Furthermore, consistent navigational structures (navigation bars, hierarchical menus, etc.) throughout a Web site can be easily implemented. Also support for multi-lingual sites enhances by reusing style elements for all languages. In combination with XML-based navigational structures, switching between languages while preserving the user context (i.e., the currently viewed page, the contents of a shopping cart, a search result, etc.) becomes possible. More details on how we could exploit the power of XML technology on the Web are given in [86, 91, 94, 95].

Having built and deployed many large-scale and complex Web sites using above technologies, people came to realize several important issues. First, the maintenance and evolution of such Web sites is an extremely time-consuming and tedious task. In some cases it was even easier (and cheaper) to re-build the whole Web presence from scratch instead of evolving it. This was mainly a result of having the layout information of the site (i.e., the actual HTML tags) scattered across different places: static HTML files, template files, in the application logic or even the content repository.

Second, the process of creating a Web site was often an ad-hoc and somewhat chaotic procedure since the various roles and responsabilities involved in such an undertaking were not properly understood. Especially the unreflected use of existing methods from software engineering did not work out to the desired extent. The significant differences between Web development and software development such as the integration of new disciplines (e.g., arts and graphic design) and content management were not considered.

Third, many Web projects suffered from continuously changing requirements. It is well known in software engineering that changes to the requirements later in the development process are costly [26]. The same is true—maybe even to a greater extent—for Web development; nevertheless proper requirements engineering is still rarely found in Web development. Other topics commonly disregarded are ease of navigation, accessibility, compatibility, cultural and legal aspects, and reliability.

In our Web projects, we also saw that XML per se does not help for arbitrary changes in the requirements for a Web site or Web application. This requires a more flexible and extensible framework for separately evolving concerns of Web applications.

The keynote held by Randy Katz at the Networking 2002 conference followed the same lines. It pointed out that the Web develops so fast, you do not know what the next killer application or technology or device will be. Nevertheless, you have to be prepared to develop applications for this yet to be appearing infrastructure in a flexible and maintainable way quickly. What is needed, thus, is a framework that enables application development in such an environment. The same is true for the development of Web sites and Web applications themselves.

A report by the Cutter consortium on Web projects [38] lists alarming facts: the developed systems did not meet requirements in 84 percent of the cases, 79 percent of the projects were delayed, 63 percent of the projects exceeded their budget.

For all these reasons, the previously cited term *Web Crisis* was coined as a parallel to the *software crisis* [117] when too many incompatible programming languages, exceeded budgets and schedules, a gap between users and programmers, and hacking type of programming methods were some of the problems. A cite from the Wikipedia free encyclopedia hits the spot: "Try `http://www.google.com` to search for 'software crisis' and read through the half a million web pages that match this simple query. Then try to summarize this knowledge within your budget and deadline. Now you might start to understand that the root of the software crisis is complexity."

In response to the Web crisis and the above challenges, a group of people at the University of Western Sydney in Austrialia coined a name for the emerging research disciplines: *Web Engineering*. A definition of Web engineering was already given in 2.1 above; a second one is given here:

> "Web Engineering is the establishment and use of sound scientific, engineering and management principles and disciplined and systematic approaches to the successful development, deployment and maintenance of high quality Web-based systems and applications." [52]

The main focus in Web engineering, thus, is not so much on how to develop new Web sites but on how to develop them in a systematic and quantifiable way that reduces development costs and supports quality attributes such as changing requirements, future extension and evolution.

The various existing Web engineering approaches are presented and evaluated in the context of our problem definition in Chapter 3. The remainder of this chapter gives a brief, non-exhaustive overview of the eXtensible Markup Language (XML) and related technologies which are the foundation on which XGuide and most other new developments in Web engineering build.

2.5 Scratching the Surface of XML and Some Related Technologies

Getting information off the Internet
is like taking a drink from a fire hydrant.

Mitchell Kapor

The original intention of the HyperText Markup Language (HTML) was not only to structure Web content but also reflect some semantic notion, e.g., using `<h1>` to indicate that the enclosed content forms a heading of level one. Web developers, however, soon started to request rendering related tags such as `<font>` to specify the font in which some content should be displayed. Even browser-specific tags such as the infamous `<blink>` and `<marquee>` elements further

polluted the language. The Cascading Stylesheet Specification (CSS) is the attempt to separate layout information from the actual HTML content. Unfortunately, CSS support—which is now available in an extended, powerful second version—is not consistently implemented across the major browsers (not even the newest versions).

Parallel to this development, the limited set of HTML tags, their nesting rules and fixed semantics were perceived as limitation. Other problems with HTML include the missing ability to specify semantic information (except for `<meta>` tags), the fact that support for reuse in HTML is hardly possible, and the inflexible model of embedding links into a document. A restriction gaining in importance with the increasing number of Web-enabled devices is that HTML was never designed with device independence in mind and thus does a poor job on several small devices and for different output formats. A detailed discussion of device independence issues in Web engineering can be found in Engin Kirda's dissertation [90].

The *eXtensible Markup Language (XML)* and accompanying specifications line up to tackle these problems and can be used in combination with existing technologies or on their own. This section briefly discusses the most important concepts and applications of these technologies and gives pointers where to find further information. Note that this discussion can only cover the most basic concepts needed in the scope of this thesis. Many details had to be omitted for space reasons; the reader is referred to the official W3C Web site for more information, new developments and the full specification documents of the technologies presented here [3, 24, 31, 39, 143].

2.5.1 The eXtensible Markup Language - XML

XML is a means to define new markup languages. Such a markup language structures documents into several sections. The sections of a document are called elements and contain text content and further nested document sections (elements). The hierarchy of all elements in a document forms a single-rooted tree structure. All elements in this tree have a start and an end tag. A tag consists of an opening angle bracket followed by the elements name and a closing angle bracket, e.g., `<order>`. The end tag further uses a slash as distinction criteria, i.e., `</order>`. Elements can contain other elements or plain text. In the start tag, attributes can be defined as key/value pairs. To add an attribute *id* to the order element, we just write `<order id="294">`.

Thus when using XML, there is no fixed set of tags but tags can be chosen as seen appropriate for the intended purpose. Technically, XML is a restricted version of the *Standardized General Markup Language (SGML)* [1] while being much more flexible than HTML. As such HTML can be completely formulated as one special instance of an XML language (and is then called *XHTML*). Figure 2.4 shows a simple XML document for an order using custom elements and attributes.

Also note that XML is, unlike HTML, *case sensitive*. Thus `<order>`, `<ORDER>` and `<Order>` are three different elements.

Two different ways to process XML documents exist today: the *Document Object Model (DOM)* and the *Simple API for XLM (SAX)*. The DOM approach strongly builds on the tree structure of an XML document and provides an interface to an in-memory representation of this

```
<order id="294">
  <customer>
    <name>Clemens Kerer</name>
  </customer>
  <product id="21">
    <name>Orange Juice</name>
  </product>
</order>
```

Figure 2.4: A sample XML document representing an order.

tree. SAX, on the other hand, iterates through the document in a depth-first manner and fires events whenever a new element, a new attribute, some text or the end of an element or attribute are detected. Unlike using DOM, SAX provides no means to navigate the XML document (e.g., find the parent node, list all child nodes, etc.). The downside of DOM is the low performance compared to SAX and the memory requirements that increase with the size of the document.

Besides Web development, many other application areas for XML exist: platform and application independent data exchange, XML messaging, XML content repositories, etc. In this work we concentrate on exploiting XML for Web development and do not discuss other application areas of XML. We will later see how some of them (e.g., XML databases as content repositories or XML Web Services as content providers) can be used and integrated in XGuide.

2.5.2 Document Type Definitions - DTDs

To describe what elements and attributes may be used in an XML document, a corresponding *Document Type Definition (DTD)* can be defined and associated with the document. Figure 2.5 shows the DTD for the order document from Figure 2.4.

```
<!ELEMENT order (customer, product+)>
<!ATTLIST order id CDATA #REQUIRED>

<!ELEMENT customer (name)>

<!ELEMENT product (name)>
<!ATTLIST product id CDATA #IMPLIED>

<!ELEMENT name (#PCDATA)>
```

Figure 2.5: The document type definition for the sample order document.

It defines element `<order>` to contain a `<customer>` and one or more `<product>` elements (quantifiers are specified using expressions known from regular expressions such as `+`, `*` or `?`). Both elements further contain `<name>` elements which in turn consist of text (indicated by the special `#PCDATA` keyword).

Attributes are defined in `<!ATTLIST>` items specifying the name of the element they belong to, their datatype (`CDATA` in the example, i.e., text), and optional modifiers specifying if an

attribute is required or optional (i.e., implied) for an element. In the example, attribute `id` is defined for elements `<order>` and `<product>`, mandatory for the former, optional for the latter.

To associate the DTD stored in `order.dtd` with our order document, we add a special line at the beginning of the XML document as shown in Figure 2.6.

```
<!DOCTYPE order SYSTEM "order.dtd">
<order id="294">
  ...
</order>
```

Figure 2.6: Associating a DTD to an XML document using the `DOCTYPE` declaration.

When an XML parser processes a document with an associated DTD; it automatically checks if all rules given in the DTD are followed. If not, an error is reported. To distinguish XML documents with and without DTDs, an XML document that only obeys the basic rules of XML (e.g., every opening tag must have a closing tag, attribute values must be enclosed in quotes, etc.) but does *not* have an associated DTD is said to be *well-formed*. If a document—in addition to being well-formed—obeys the rules in an associated DTD, it is said to be *valid*. Thus our sample document in Figure 2.4 is valid with respect to the DTD given in Figure 2.5.

2.5.3 XML 2nd EDITION

When people started composing XML documents, they realized that elements with the same name but different DTD definitions cause problems. Assume you want to compose the two documents shown in Figure 2.7; what definition could you give for element `<address>` in the resulting document?

One solution to this problem would be to rename one of the address tags to ensure uniqueness of element names. Since the DTD of that document might not be under your control and a modification of elements is undesireable in general, a more powerful solution was needed: *XML Namespaces*. The concept of namespaces was already successfully applied to solve a similar problem in programming languages. If several types with the same name need to work together, namespaces are used distinguish the different types (e.g., C++ or .NET namespaces, Java packages, etc.).

You can think of a namespace as a unique id for a DTD that is added to all elements of the DTD to be able to trace elements back to its original DTD. In fact, the namespace is added as prefix to elements resulting in the following form for element names: {`namespace-identifier`}:{`element-name`}, e.g., for a namespace *order* and element *customer*, this would result in `order:customer`.

The remaining question is how to ensure uniqueness of namespace names (i.e., make sure nobody else also uses namespace *order*). By convention, namespaces are based on URLs—and the unique association of URLs to their owning principals (e.g., companies, organizations, persons, etc.) is implicitly ensured because IANA (Internet Assigned Numbers Authority)

```
<customer>
  <name>Clemens Kerer</name>
  <address>
    <street>Hetzendorferstrasse</street>
    <number>93/6/8</number>
  </address>
</customer>
```

(a)

```
<producer>
  <name>Orange Juices Inc.</name>
  <address number="23a" street="Orange Street">
  </address>
</customer>
```

(b)

Figure 2.7: Two XML documents with conflicting definitions for the `<address>` elements.

globally manages URL names. Thus to define a new namespace, you take your URL (e.g., `http://www.orangejuices.com/`) as prefix and append the desired namespace name to it. You are locally (i.e., within your organization) responsible for the uniqueness of all namespaces using your URL as prefix. Principally, however, namespace names can be arbitrary strings and technically nothing keeps you from using any other string for your namespace. Not obeying the above convention quickly renders the idea of namespaces unusable since name clashes are likely to appear again. Following this convention, we would reformulate the above example as `http://www.orangejuices.com/order:customer`.

The element name including the namespace prefix is also called *fully-qualified element name* as opposed to the element name without namespace prefix (i.e., 'customer') which is called the element's *local name*. If you would now write a namespace-qualified XML document, it would be a tedious task to keep typing the URL-based namespaces all over again and again. As a consequence, abbreviation for namespace names can be introduced using the special `xmlns` syntax shown in Figure 2.8.

If you do not specify a namespace prefix for an element, it is said to belong to the *default namespace*. This namespace is implicitly always defined but can also be overridden as shown in line 2 of Figure 2.9. The default namespace is changed to `http://www.orangejuices.com/order` by not specifying an abbreviation for the associated namespace. All elements without an explicit namespace prefix are then assumed to lie in the specified namespace.

A final remark on namespaces: namespace abbreviations are valid always withing the scope of the element in which they are defined and can be overridden in nested elements. In Figure 2.10 this mechanism is used extensively.

```
<o:order id="294"
       xmlns:o="http://www.orangejuices.com/order">
  <o:customer>
    <o:name>Clemens Kerer</o:name>
  </o:customer>
  <o:product id="21">
    <o:name>Orange Juice</o:name>
  </o:product>
</o:order>
```

Figure 2.8: The sample XML document using namespace abbreviations.

```
<order id="294"
       xmlns="http://www.orangejuices.com/order">
  <customer>
    <name>Clemens Kerer</name>
  </customer>
  <product id="21">
    <name>Orange Juice</name>
  </product>
</order>
```

Figure 2.9: The sample XML document overriding the default namespace.

```
<o:order id="294"
       xmlns:o="http://www.orangejuices.com/order"
       xmlns="http://www.orangejuices.com/order2">
  <customer>
    <name xmlns="http://www.orangejuices.com/name">
      Clemens Kerer
    </name>
  </customer>
  <o:product id="21"
    xmlns:o="http://www.orangejuices.com/product">
    <n:name xmlns:n="http://www.orangejuices.com/name">
      Orange Juice
    </n:name>
  </o:product>
</o:order>
```

Figure 2.10: The sample XML document using and overriding several namespace abbreviations.

In the `<order>` element, the `order` namespace is bound to `o` and the default namespace is overridden with namespace `order2`. Consequently, the `<o:order>` element is in the `order` namespace while the customer element lies in the `order2` namespace. The customer name element again overrides the default namespace with a `name` namespace. Similarly, the `<o:product>` element binds the `product` namespace to prefix `o`—resulting in a different meaning of the `o` prefix of the `product` and `order` elements.

2.5.4 THE EXTENSIBLE STYLESHEET LANGUAGE - XSL

The Extensible Stylesheet Language specification is further structured into three parts:

- **XPath.** XPath is a selector language to select arbitrary parts of an XML document. This could be a single element, a set of elements, text values or any other part of an XML structure. XPath takes advantage of the tree structure of an XML document and uses an addressing scheme based on the branches of the tree (e.g., select all nodes with a given name, select all successors of the current node with a given name, etc.). Predicates further enrich the semantics of such expressions by adding conditions such as only selecting elements with a given attribute, with a given successor element, or with a given text content.

- **XSL Transformations (XSLT).** XSLT is a transformation language that provides rules to transform an input XML document into a different output representation such as another XML format or plain text. XSLT consists of so-called *templates* which transform a set of nodes selected by an XPath expression into some other representation. XSLT processing always starts at the root element of an XML document. In each template, recursive calls to other templates (again based on XPath expressions or names) can be embedded which results in an inherently recursive processing of XML documents.

- **XSL Formatting Objects (XSL-FO).** XSL-FO is an XML vocabulary for specifying page-oriented formatting information such as page dimensions, margins, headers, footers, page numbers and much more. The basic idea is to first specify the properties of a set of master pages, then arrange a set of references to the master pages to form the sequence of pages in the document, and finally, fill the pages with the actual content. A common application of XSL-FO is to generate RTF or PDF documents from XML content. Though currently not implemented in any browser, XSL-FO capable Web browsers can also be envisioned in the future.

In XSuite, we mainly use XPath and XSLT to transform input XML documents into other markup languages such as XHTML. Both parts are relatively complex on their own rendering a more detailed tutorial here impractical. In Figure 2.11 we provide a sample stylesheet to convert the order document from Figure 2.4 into an XHTML page.

The stylesheet first defines a template for the `<order>` element and creates a skeleton HTML document in its body. It then recursively invokes the templates for the `<customer>` and `product` elements. In the case of a customer, we output the name of the customer enclosed

```
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <!-- start with 'order' element -->
  <xsl:template match="order">
    <html>
    <head>
      <title>Order Document</title>
    </head>
    <body>

    <!-- continue with customer -->
    <xsl:apply-templates select="customer" />

    <!-- continue with product -->
    <xsl:apply-templates select="product" />

    </body>
    </html>
  </xsl:template>

  <!-- transform 'customer' element -->
  <xsl:template match="customer">
    <h1><xsl:value-of select="name" /></h1>
  </xsl:template>

  <!-- transform 'product' element -->
  <xsl:template match="product">
    <h2><xsl:value-of select="name" /></h2>
    <p>Product id = <xsl:value-of select="@id" /></p>
  </xsl:template>

</xsl:stylesheet>
```

Figure 2.11: The sample XSLT stylesheet to transform an order document into an XHTML page.

by `<h1>` tags. In the case of a product, we do the same using `<h2>` tags for the product name. We also add a paragraph with some text indicating the value of the product identifier (i.e., the attribute `id` of the product element).

The result of processing the sample order document with this stylesheet is illustrated in Figure 2.12.

```
<html>
  <head>
    <title>Order Document</title>
  </head>
  <body>
    <h1>Clemens Kerer</h1>
    <h2>Orange Juice</h2>
    <p>Product id = 21</p>
  </body>
</html>
```

Figure 2.12: The result of applying the sample XSLT stylesheet to the order document.

2.5.5 XML SCHEMA

XML Schemas are the successors of document type definitions to overcome several problems with DTDs: DTDs are not XML document themselves and thus cannot be processed with the same tools; DTDs support only a small number of datatypes; nesting of elements can be constrained in a rudamentary way and namespaces are not supported. The XML schema recommendation is based on many other schema languages such as XML Data [102], Document Content Description (DCD) [30] and Schema for object-oriented XML (SOX) [42] that tried to solve parts of the problems with DTDs.

The specification consists of a part on datatypes and a part on structures. The datatype part defines a complete type system for XML documents including simple types (i.e., string-based types) and complex types. The latter support element nesting, attributes, and mixed content. Further inheritance rules are defined to support extension and restriction of already existing types. The part on structures deals with how elements can be nested and how this nesting can be constrained (e.g., how often an element may appear as child, whether it is required or optional, whether it may be substituted by another element, whether the sequence of child elements matters, etc.).

XML schemas are regular XML documents living in a fixed schema namespaces and fully support namespaces by so-called *target namespaces*. A target namespace indicates to which namespace an element specified in this schema belongs, effectively solving the name clash problem mentioned before. An excellent introduction can be found in the XML schema primer [53]. A sample schema for the order document of Figure 2.4 is shown in Figure 2.13.

XSuite uses schemas as part of the contract to specify the structures and data types of the components used and introduces several operations on schemas to compose several components or generate documents from the schemas.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.orangejuices.com/fruit"
     xmlns:f="http://www.orangejuices.com/fruit">

  <xsd:element name="order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="f:customer"
                     minOccurs="1" maxOccurs="1" />
        <xsd:element ref="f:product"
                     minOccurs="1" maxOccurs="unbounded" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:integer"
                     use="required" />
    </xsd:complexType>
  </xsd:element>

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

  <xsd:element name="product">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="f:name" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:integer"
                     use="required" />
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="name" type="xsd:string" />

</xsd:schema>
```

Figure 2.13: A sample schema for the order document.

2.5.6 OTHER X TECHNOLOGIES

Apart from XML, namespaces, DTDs, schemas and XSL, other interesting XML related technologies exist in the context of Web engineering. *XForms* are an extended form specification that is much more powerful than HTML forms. *XInclude* is an emerging technology supporting composition of XML documents. *XLink* is a new linking specification that supports the definition of links outside the actual XML document (again a separation of concerns issue), defines multi-directional links and one-to-many links. *RDF*, *DAML*, *OIL* are specifications related to semantic information on the Web. *XML Query* is the counterpart to the Structured Query Language (SQL) in the XML world. *XML Signature* and *XML Encryption* add security to XML documents. *VoiceXML* supports voice input and output.

Though these technologies are interesting and important to Web engineering in general, they do not directly affect the core XGuide principles. For this reason, we provide no further details but refer the reader to their official hompages available via `http://www.w3.org`.

CHAPTER 3

RELATED WORK

Reviewing has one advantage over suicide:
in suicide you take it out on yourself;
in reviewing you take it out on other people.

George Bernard Shaw

As the Web evolved over the years, so did the methodologies for Web engineering. This process is very similar to the evolution of software engineering methodologies. The need for a structured process model for the development of software became obvious during the software crisis [117]: ad-hoc development, missing requirements engineering and a plenitude of programming languages and environments led to unmanageable software and projects constantly exceeding their budgets. Many of these problems also hold for early Web development. Web projects kept missing deadlines (and still frequently do), ad-hoc development resulted in overly complex systems and maintenance was exceedingly difficult since HTML fragments were distributed over templates files, program code and even content repositories.

In the field of software engineering, new methodologies appeared with the victory of object-oriented programming over modular techniques. Once object orientation was sufficiently understood, component-oriented and aspect-oriented approaches introduced even higher levels of abstraction and separation of concerns. The corresponding developments in Web development are object-oriented approaches that view a Web page as an object composed from several smaller objects and support inheritance and reuse. More recently and driven by the rapid acceptance of XML, separation of concerns (namely content, layout and application logic) are the dominant features of Web engineering methodologies paralleling component and aspect-centered approaches in software engineering.

In this chapter, we discuss Web engineering approaches and compare them with the scenario outlined in this work. We first introduce a taxonomy for Web engineering methodologies

capturing the key characteristics. A detailed discussion of existing methodologies follows. A comparison and classification of the presented approaches shows their applicability for different kinds of Web projects and points out how the approach taken in this thesis differs from them.

3.1 A Taxonomy of Web Engineering Methodologies

This section introduces the characteristics used to classify Web engineering methodologies presented in subsequent sections. They are independent of any particular domain or implementation technology. But as always, some methods and technologies lend themselves more easily towards integration with a technology or domain than others. The selected characteristics cover a broad range of requirements a modern Web engineering method has to deal with. It is important to emphasize, however, that an evaluation on such an abstract level necessarily ignores the specialties of a methodology and can only be used as a high-level indicator rather than the only source for a decision for a real-world Web project.

The first and one of the most important characteristics found in all Web engineering methods is *Separation of Concerns*. A strict separation of concerns is desirable both during development and even more during the maintenance phase to decouple tasks, avoid redundancies, and support reuse of design and implementation artifacts. Web applications integrate a growing number of disjoint concerns; most commonly the actual content, the layout information and the application logic. For the purpose of this evaluation we take into consideration the following five concerns commonly found in today's Web applications: content (data), site structure, graphical appearance (layout), application logic, navigation.

- The **content** is the actual information to be published. The data is kept in files, relational databases, object stores or repositories for semi-structured data.
- The **site structure** defines all the Web pages in the application and how the content is distributed among them. For large projects, it can define subsections (e.g., for internal vs. external, public or restricted access, etc.).
- The **graphical appearance** adds the formatting instructions to the actual content on the pages. This often means to transform the content into HTML pages. It is desirable, however, to support other output formats such as WML, text or PDF documents.
- Integration of and interaction with the **application logic** is crucial as many Web application get more and more dynamic. The application logic deals with the ability to dynamically generate pages and process user interaction in business processes on the back-end.
- The **navigation** concern describes how pages are related and what navigation paths are offered to users. Typical navigation structures include non-contextual structures such as sequential or hierarchical menu bars for quick access to different parts of a Web site and contextual structures such as indexers, guided tours and indexed guided tours based on the user's current context (e.g., search result, selected page, etc.).

In addition to these five concerns, there are several others that are likely to gain in importance in the future. These include full device-independence, security and meta-data. Device-independence is not only about supporting multiple output formats but also how to support the same business processes on multiple devices with varying capabilities. The security concern talks about the necessary credentials a user must have in order to access a page or trigger a user interaction. Meta-data currently attracts many researchers in the context of the W3C's semantic Web [145] initiative. Though the usefulness of meta-data on the Web is not completely clear to date, it is obvious that meta information can improve the searching and information retrieval process. As such, a separate concern might talk about the available or required meta-data for Web applications, sites, pages or even dedicated items on a page. These concerns are currently not commonly found either in Web engineering methodologies or in Web applications. For this reason we do not consider them in the review of related work directly but include support for these concerns in the evaluation.

We measure the degree to which a method supports separation of concerns by giving the number of how many of the above five concerns are explicitly supported in the method. We further add (+) to this number to indicate that a method supports additional concerns or supports a concern much better than the other methods. We add (-) if a method identifies a concern but falls short in explicitly supporting it as a separate concern.

The second criterion we use in this evaluation is *Reuse of Artifacts*. Reuse is one of the major contributors to improve quality, support faster development cycles and ensure consistency. This property of a methodology is often closely related to its ability to separate concerns and make them reusable as separate entities. However, this is not always the case. While a method might identify the layout as a separate concern, the use of HTML fragments to add layout information is often page-specific and does not allow for layout reuse.

We define three levels of reuse: on the highest level (3), all artifacts can be reused. Depending on the method, this refers to both design and implementation artifacts. On level (2), some artifacts can be reused, but the method also requires non-reusable artifacts. On the lowest level (1), no explicit support for reuse is provided.

Another criterion we include in the evaluation is *Complexity* of a method. This criterion is especially important to set a context for the measurements of the other criteria. Some development methodologies are intentionally kept simple for educational use or beginners in the field of Web engineering. Obviously, they cannot achieve the same degree of evaluation results compared to methods that are not concerned about complexity. Thus all other evaluation results must be viewed relatively to a method's complexity value. The complexity value includes whether a method is easy to understand, the complexity of the tasks proposed by the method, the required knowledge to apply the method and whether a method uses standards rather than proprietary notations, diagrams, models or technology.

We rate a method of having complexity (1) if it is easy to learn and apply, does not have a steep learning curve in terms of new concepts and mainly relies on existing, well-known technologies. Methods with a medium complexity (2) require acquisition of some new knowledge and extend existing technologies to match their requirements. High complexity (3) methods, finally, use whatever techniques they see fit to achieve their goals as best as they can. They are

not concerned about complexity and assume a developer is willing to learn the new notations, diagrams and techniques that come with the method.

An important aspect in all state-of-the-art Web applications is the ability to dynamically create pages and process user input in (sometimes legacy) business processes. *Integration of Application Logic* covers both of these more and more important issues.

We say a method has low (1) support for dynamism if no support for the integration of dynamically generated pages and user input processing is included in the method. Medium support (2) means that there might be a set of pre-defined operations that can be used for dynamic page creation (e.g., a set of templates to include database queries). These operations, however, are not extendable and no user input processing relies on any sort of CGI programs. A high support (3) in this category means that a generic way of integrating dynamic content into pages exists; ideally accompanied by a way of defining an explicit interface between the back-end business application and the user input in a Web page.

The next criterion distinguishes between *user-centric* and *data-centric* approaches. Most existing methodologies are rooted either in software engineering or in database design. Since the early development of dynamic Web applications was driven by the requirement to integrate (relational) database content, several methodologies start with the data model at hand and map it into a Web application. Other methodologies focus on an object-oriented data model of the Web application. Another category of approaches is based on user expectations and user analysis. In this scenario, a Web application is modeled to meet the expectations of a user community and the actual implementation in terms of software modules and data organization is derived from that.

Often, a user-centric, requirements-driven approach is preferable to creating the Web site according to the underlying data model. In some data-intensive publishing scenarios where the goal is to publish a large amount of highly structured and consistent information, the data-centric approach might be the better choice. We thus classify a method in being either user- or data-centric.

Transformability (Model to Implementation Mapping) indicates how difficult it is to map the design of a Web application (i.e., conceptual models, layout templates, etc.) to an actual implementation. We do not focus on any special implementation technology but consider the generic mapping process to any implementation. Depending on the level of abstraction a method is operating on and the availability of development tools supporting the method's tasks, we distinguish three different levels for this characteristic:

- Level 1 - Conceptual Modeling Only: in this approach a conceptual and often very expressive model can be formulated. The methodology only lives in the conceptual space and does not provide any support to transform the conceptual design into an implementation. It is left to the developer to implement such a model in terms of an existing technology.

- Level 2 - Partial Generation: in this case, the conceptual model is analyzed and the actual Web application can be partially generated from it. Usually a set of templates and skeleton files that need further manual adaptation or extension are generated.

- Level 3 - Automatic Generation: here the complete Web application is automatically generated from the model and does not need any further manual adaptation. In this scenario, any update or evolution task takes place on the conceptual level and the Web site is merely generated from the updated model.

Methodologies with a high degree of abstraction frequently do not provide implementation support. A lower level of abstraction, on the other hand, renders automatic application generation possible. This becomes especially important in the maintenance phase when updates, extensions and modifications have to be integrated. With only a conceptual model, no support for integrating changes is possible; using the second approach, care has to be taken to not overwrite previous manual changes; if the model is automatically transformed into the final application, only the re-generation of the application is necessary to reflect the updated state of its model.

A characteristic that reflects the rapidly changing requirements and short life-cycles on the Web is *Concurrent Development*. It reflects the ability of a method to reduce the overall development time by decoupling and doing as many tasks as possible in parallel. Most existing Web engineering methodologies do not support parallel development but stick to a strictly sequential process. This property only focuses on the implementation of the Web site; during the design, parallel development is hardly possible and usually not desirable.

We again distinguish three levels of parallelism in Web engineering methodologies. On the lowest level (1), all tasks are executed in sequence and depend on each other. No parallelism is possible. On a medium level (2), at least some tasks can be executed in parallel. The highest level (3) requires that most of the tasks involved in creating the actual Web implementation can occur concurrently.

Finally, we include the *Additional Value* characteristics that covers any special properties and features the other characteristics did not cover. The span of existing Web engineering methodologies is so broad, a classification can never include all properties of every method. Thus we add this characteristic to capture any special value a method has. A value of (1) means that a method has no or insignificant additional value other than included in the other characteristics. A value of (2) means that there are some useful and interesting features that are superior to what other methods provide. Finally, a value of (3) means that a method provides many additional features and concepts that do not fit in the above classification but contribute significantly to the usefulness of the method.

We considered but ruled out *Process coverage* and *Applicability* as further criteria. Process coverage gives an indication of how much of a Web application's life cycle is covered by a methodology. Several of the methodologies presented below do not explicitly include phases such as feasibility, requirements analysis or maintenance. At a closer look, however, it becomes obvious that all of them understand the importance of these phases but (mostly for space reasons) did not include them in a detailed discussion. It would thus be misleading to include such an evaluation criterion here. Applicability discussed the potential application domains of a method. After evaluating the methods, it became clear that this criterion can be described in terms of the data-centric vs. user-centric property and the level of support for the integration of application logic. The more user-centric a method is and the better integration for application logic it provides, the broader its application domain is. A data-centric method, on the other hand, that does

not support dynamic page generation, is only applicable if large amounts of static information, e.g., in classical Web-based information systems, is published.

In the remainder of this chapter, we discuss several Web engineering methodologies with respect to these criteria. We evaluate each of them with respect to the criteria presented above and compare the methods to each other at the end of the chapter.

3.2 A Discussion of Existing Approaches Towards Web Engineering

This section starts with the discussion of the most widely known and referenced methodologies including HDM/OO-HDM, RMM and extended RMM. These methodologies have a strong background in hypertext and database research respectively. While they form a solid foundation for our discussion, they are both relatively old approaches and do not take into account newer requirements such as non-structured information domains, functionality-centric dynamic Web applications or device independence.

Several newer and less-known approaches are proposed in literature and presented next. They include a diverse range of perspectives from user-centered methods to purely object-based models.

3.2.1 The Relationship Management Method - RMM

The relationship management method (RMM) [77, 79] views hypermedia as managing relationships between information entities. The RMM data model (RMDM) is inspired by and based on the hypertext design model (HDM) [62] and its successor HDM2 [63]. Unlike HDM which is a data model in its own right, RMDM is embedded in a methodology that shows how to create and use the data model.

RMM method defines seven phases starting with the definition of the entity relationship (E-R) design. The well-documented E-R approach [139] was chosen since it is easy to use and already commonly applied in database design. Just as for databases, the diagram defines entities representing the actual information items that have attributes/fields holding the content. Also derived from database design, one-to-many and many-to-many relationships among entities can be introduced. In the next step, the entities are further split into *slices* that group those attributes of an entity that will appear together on the final Web page. Structural links are added to navigate the slices within an entity. The third phase deals with navigational design. Navigational design is based on the inter-entity relationships in the E-R diagram and includes a variety of access structures such as indices, guided tours, and groups. The output of this phase is the relationship management data model (RMDM) that enriches the E-R diagram by all access structures (inter- and intra-entity) that have to be implemented.

The RMDM is the input for the conversion protocol design that defines how the elements of the RMDM are mapped to objects in a particular development environment. These objects are

then integrated in the user interface design phase with the actual layout information that describes how RMDM objects shall be visually represented in the final page. The runtime behavior design phase defines how navigational access structures are implemented and how content is generated dynamically. Finally, the construction and testing phase completes the development process and turns the output of the previous phases into the final Web application. This last step can happen automatically with appropriate tool support.

Based on their experiences with RMM, Isakowitz et al. presented an extended version of the RMM in 1998 [78]. The two major changes were the extension of the slice model to *m-slices* [77] and the replacement of the navigation model by the application diagram. The concept of m-slices overcomes the restriction of the original RMM that a slice can only contain attributes of a single entity. The original design resulted in overly complex navigation structures since no page could contain information from more than one entity. Also users easily lost trace of the navigation context. Further m-slices can be composed with other m-slices to form larger information items and are the core component in RMM. The application diagram was introduced to modify the development process to not require a strict top-down approach. First, the requirements analysis phase missing in the original version was explicitly added to the development process. A top-down view of the application diagram then contains all the Web pages and the hyperlinks between them. The m-slice design then defines the content components and how they are related. Next, a bottom-up approach towards the application diagram is undertaken by adding all identified m-slices to the diagram and defining the links between them. Depending on how much of the application is already implemented and whether the requirements were correctly captured in m-slices, the result of the bottom-up application diagram should match the envisioned top-down version. Also navigation modeling is better supported now. Hyperlinks consist of a content anchor, a starting point, a target and an ending point.

The relationship management method defines a sequential process that covers all phases of the life-cycle of a Web application and starts out at an abstract level that gets refined towards the actual implementation in every step. RMM is not bound to any concrete implementation technology or platform; a case tool called *RMCase* supporting the methodology exists [46]. Separation of concern is an issue in terms of content, structure, layout and navigation. Reuse is supported on the level of m-slices. In [78] the authors use a database report generator to add HTML layout to the RMDM objects. This approach falls short in fully separating the layout information and it remains vague how application logic is integrated.

The most limiting factor of the RMM definitely is its tight coupling with relational data models. While RMM works great if all content is stored in a database and the information domain is highly structured, semi-structured data such as XML documents are not supported. Considering that RMM was first presented in 1995, it is not surprising that XML, device-independence, or logic centric Web applications are not a major concern.

3.2.2 Analysis and Design of Web-based Information Systems

An early approach towards the analysis and design of Web-based information systems that uses RMM (see 3.2.1) as foundation is presented in [137]. The authors point out that RMM and

OOHDM (see 3.2.3) are mainly suited for information dissemination but largely ignore user interaction, application logic and maintenance. Thus the method focuses on the architectural and functional aspects of a Web site explicitly including the maintenance costs; it is not concerned about the graphical appearance. The proposed method consists of two approaches: a static and a dynamic approach. The static approach exploiting RMM's entity relationship modeling and the dynamic approach using scenarios to trace how and by whom resources are accessed and updated.

After the requirements analysis, the E-R analysis of the problem domain identifies entities and relationships and creates an E-R model. The entities in the model are further classified as agents, events and products. Following this terminology, agents are actors that conduct events; the outcome of an event is a product. Members of a committee would be actors, a meeting would be an action and the meeting notes would be the product. This distinction is necessary to better structure and execute the maintenance activities later. In the next step, the scenario analysis captures potential users, their goals, the Web resources needed and how the system is going to interact with the user. Scenario tables detail on how a user can achieve its goals. They list the sequence of steps (a navigation path), the agent executing a step, the action and the Web resource(s) involved . They also build the backbone for the system's architecture design.

The architecture design is a data model diagram that again exploits RMM's data model. It also contains the navigation structures for the final site and maps entities to actual Web resources. As in the first versions of RMM, entities cannot be combined and placed on a single page. The method supports three navigation methods: guided tour, index, and indexed guided tour. The scenario design and the architecture design are performed concurrently and cross-checks between the two models ensure consistency. A final step defines the attributes for the Web resources. Attributes are meta-information on the resources themselves (e.g., last modified date, managed by, expires, etc.) that is used for maintenance activities. A notification system keeps track of the expected and actually performed changes and can report missing maintenance activities or changes in the system.

Meta-level links are introduced to express a semantic relationship between Web pages outside the pages themselves. For this purpose, the method requires an extended Web server and client. The meta-navigation information is carried in the header of the HTTP request and response messages and pursue the same ideas as the out-of-line links recently presented in the context of the W3C's XLink/XPointer recommendations. The problem with this sort of meta-links is that they cannot be linked to a specific position or semantic context in a document. Thus such a link only establishes a relationship between pages but does not have a dedicated source or target within the pages themselves.

Though this approach also suffers from the requirement of a structured information domain, it identifies several important aspects that are still rarely found in other methodologies. First the application logic is explicitly considered (though no concrete hint is given how it is modeled or implemented). Second the user of meta-data (in this case for maintenance) is included in the method. Finally, the authors state the need for meta-level links that do not require modification of the source or target documents; a requirement that is, even today, rarely considered.

3.2.3 Object-Oriented Hypertext Design Method - OOHDM

The object-oriented hypertext design method (OOHDM) [131–133] uses abstraction and composition mechanisms in an object-oriented framework to support the description of information items, navigational patterns and interface transformations. OOHDM splits the development process into four phases supporting incremental modeling, i.e., each phase adds new object-oriented models or enriches existing models from previous phases.

The conceptual design phase defines an object-oriented model of the application domain. It is only concerned about the semantics of the domain and does not include any user or task related considerations. The model itself is a slight extension of the well-known class diagrams used in UML. The only difference is that relationships can be given an explicit direction and attributes of classes can have enumeration types (e.g., a sequence of value types rather than a single type). This is similar to defining collection types over an existing type system.

The main focus of OOHDM is on the navigational design. The navigational design defines an application as a navigational view over a conceptual domain model. The navigational model contains navigational classes including nodes, links, access structures and indices. The important concept of a node defines what parts of the conceptual model are aggregated in a single node and can be compared to m-slices in RMM. Navigational nodes can also be thought of modeling the actual page structure of a Web application. A query language similar to the one presented in [88] is used to express what a navigational node should comprise. Links between nodes are derived from the relationships defined in the conceptual model. Once the navigational model is completed, the navigational classes are grouped in so-called navigational *contexts*. A context describes how the navigational model is accessible to the user, e.g., using guided tours, indices, etc. Contexts further structure the navigation space and form the context model. An approach how navigation designs can be synthesized is presented in [74]. First user profiles and corresponding scenarios are created; then a simple diagram of navigational paths is created which is then refined in a partial and, eventually, a final context diagram.

In the next step an abstract representation of the user interface is created using abstract data views (ADVs) [41]. An abstract data view defines user interface classes that describe a page as a composition of primitive user interface classes such as buttons, images, and text. The structural composition is enriched by defining how user interface events are handled and what events result in a change of the currently visible view of the navigational model.

The final phase of OOHDM is the implementation of the interface classes in terms of an actual development environment and platform. OOHDM does not propose or define any implementation technology or platform but suggests to store all modeling artifacts (conceptual classes, navigational classes, contexts, etc.) in one or more databases. Navigation contexts are then implemented as stateful objects that always keep track of the currently visible page and the other pages in the same context (e.g., to correctly switch to next or previous pages in a guided tour). The integration of an layout information is envisioned using HTML templates that are enriched by function calls to objects of the conceptual model to retrieve and embed dynamically calculated values.

OOHDM-Web [130] is a development environment for OOHDM that is based on the CGI Lua scripting language and a module to connect to relational databases. These databases store

all information and modeling artifacts and provide a library of functions to complete HTML templates with dynamic information.

The high level of abstraction of OOHDM enables it to be used in a broad variety of application domains. OOHDM is missing an analysis phase in the process and also does not explicitly consider testing, maintenance or evolution. Separation of concerns is only achieved on the conceptual level but lost on the (suggested) implementation level where content, application logic and layout information is again intermixed. Reuse of conceptual classes is supported by separating the conceptual classes from the navigational nodes. The abstract data views also benefit from the reuse of existing interface objects. The suggested implementation environment does not support separation of concerns or reuse of implementation artifacts. The transformability from the conceptual model to the implementation space remains unclear.

3.2.4 A Schema-Based Approach to Web Engineering

Unlike RMM and OO-HDM, the schema-based approach towards Web Engineering presented in [100] takes a document-oriented view of the development process. The creation of a Web site is considered a combination of document engineering and software engineering. The method, however, only considers the document engineering part including detailed content structuring, quality factors for content, and content adaptation for selected target audiences and contexts (language, culture, etc.).

The development process comprises phases for analysis, design, authoring and production. While maintenance and evolution is not mentioned at all, testing is seen as integral and continuing task throughout the process (though no details are given on how and what should be tested). The phases themselves are again document-centric. As such the analysis phase talks about editorial guidelines and writing standards but does not include functional requirements, target devices, or platform decisions. The design and authoring phases are said to be inseparable and often indistinguishable and thus effectively collapsed. Both phases, however, are split in into two steps: design/authoring-in-the-large and design/authoring-in-the-small. As a consequence, it would be more appropriate to separate the *in-the-large* phase from the *in-the-small* phase rather than authoring from design. According to the authors, the production phase should merely consist of turning the designed hypertext components into an actual Web site.

The method is based on the classic hypertext model of (information) nodes and hyperlinks. Kuhnke et al. [100] introduce different classes of nodes and link types that are designed during the authoring-in-the-large and design-in-the-large phase. The result of these phases are patterns and templates that can later be used to create instances of nodes and links that correspond to actual Web pages and hyperlinks. The *in-the-large* phases can to some degree be compared to the idea of having a contract for the content concern or an XML schema describing a class of XML instance documents. This approach is different from most other methods in that it focuses on the structure and content of the actual information nodes rather than using a given, underlying data model and transforming it into a hypertext model.

Separation of concerns is identified as an important issue; however, the content is clearly the dominant concern. Other concerns such as layout and navigation information are added by

so-called expanders, macro-like code fragments that operate on the content. The idea of having standardized catalogues of transformers to solve standard problems for standard parameters cannot make up for that. Reuse is supported where the design process allows it. Nodes and link types can be reused and expanders can be applied to multiple nodes to solve the same standard problem, also requiring the nodes to conform to a standard set of parameters. In extension of the original hypertext model, links play an important role and navigation paths and structures between nodes are well supported on the conceptual level. Unfortunately, their implementation again relies on expanders strictly limiting the approach to standard solutions.

The decision to include only the static, document-oriented part of the development process in the methodology allows tools to automatically generate (static) Web sites from hypertext models. SchemaText is a commercial tool that supports this approach. The major drawback of this decision is that only static Web sites can be supported. Hardly any Web site or application today consists of only static pages. On the contrary, more and more Web applications are developed that do not contain static information at all but generate all pages dynamically. The integration of application logic with hypertext concepts turns out to be one of the key requirements for state-of-the-art development methodologies and is completely ignored in [100].

3.2.5 SWM - A Simple Web Method

The simple Web method (SWM) [40] tackles the problem that many developers find methodologies too complex and hard to understand [10]. SWM is primarily intended for educational use and for inexperienced users. The fundamental philosophy for this approach besides being simple is to strongly support the early phases of the life-cycle of a Web application, provide tool support and traceability of changes.

SWM distinguishes five phases. The first phase, planning, is concerned about feasibility and project management. The following analysis defines the target audience, the content to be presented, the market situation and constraints such as copyright, developer expertise, etc. In the design phase, the structure, the visual layout and the navigation style is defined and results in a set of storyboards. In the building phase, the actual Web application is built. The maintenance phase finally covers all activities after the initial deployment of the Web site.

The presented method extends existing software engineering techniques by a navigation chart and page mockups. No more detailed description of the phases or the development process is given. Consequently, SWM operates on a very abstract level that does not provide much guidance for developers.

A more innovative approach is taken in order to support the whole life-cycle of the application and project management. The need for process modeling, project management, quality management, and configuration management tools interacting with development tools to appropriately support the developer is stated. PAWS, the Project Administration Web Site, is the attempt to combine several such tools in an interactive Web site accessed and updated by the developers. PAWS supports a simple process model, action minutes, deliverable and tasks. Developers and managers can obtain a detailed status on all tasks and deliverables. In a small case study, most

participants understood the SWM mostly or completely. Action minutes and task management were found to be the most useful aspects of PAWS.

Due to the high abstraction level and only vague definition, SWM does not easily fit in the categorization scheme presented in 3.1. It does not touch separation of concerns, support of reuse or its applicability. It does, however, support the full life-cycle of a Web site and practicability is good in that the method is easy to understand and apply. Unfortunately, no existing standards or tools are explicitly supported.

Regarding many of today's application scenarios, SWM cannot be considered the appropriate methodology. It leaves out too many crucial requirements such as concrete modeling diagrams, support for reusable components and separation of concerns. Since SWM was designed to be simple and for educational use, it probably also was never intended for complex Web projects. SWM contributes and outdoes other approaches integrating process and project management into the actual development process.

3.2.6 The Object-Oriented-Hypermedia Method (OO-H)

Like other approaches ([37,105]), the object-oriented hypermedia method (OO-H) [70,71] looks at a Web application as an object-oriented software artifact. It extends traditional object-oriented development techniques with two new views: the navigation view and the presentation view. The navigation view extends a class diagram to include link information between information items, the presentation view provides default rules to transform content into the final output.

The OO-H design process starts with a class diagram modeling the information domain. Then several navigation access diagrams (NADs) are added, one per target device or audience. Based on the NADs and a set of default rules, abstract presentation diagrams (APDs) are created to help the developers map the information to the desired target language. A pattern catalogue provides user-centered solutions for presenting information and capturing user interaction. Other phases such as analysis, testing or maintenance are not explicitly covered in the design process but can easily be added.

The abstract presentation diagram separates five concerns contributing to the final Web application: the content (tStruct), the layout (tStyle), the user input (tForm), client-side scripts (tFunction), and views/windows shown to the user (tWindow). This clear separation is based on XML files for each concern that are backed by a custom document type definition (DTD). The DTD defines the valid elements and interaction styles for the Web application. This is necessary to enable automatic generation of a default implementation. On the other hand, it restricts the practicability of the approach and does not support alternative ways of defining information and/or interaction. For instance, all content has to be provided in as a collection of objects and links with specified attributes. Such a DTD can be seen as page description language that arranges a given number of objects and links on a page. It does not support other structures or the introduction of new elements. Transformation rules in an OCL-like syntax (object constraint language, [121, 149]) can be used to transform artifacts, e.g., a template in the APD into an HTML page.

OO-H builds on existing standards (XML, DTD, OCL, etc.), defines a sequential development process similar to other object-oriented approaches, clearly separates concerns and provides high transformability. Among the goals of OO-H are flattening the learning curve and decreasing the cost of Web development. While the latter is likely to be achieved, the method as a whole remains rather complex, especially regarding the definition and syntax of transformation rules. Also the integration of and interaction with application logic is not clearly defined though the *tForm* concern mentioned above provides some support for user input. It remains unclear, however, how the claimed device independence features are achieved, how the input is handed to the back-end business processes and how the application logic interfaces for interacting with the content templates work. OO-H is, at least conceptually, one of the most state-of-the-art Web engineering methodologies.

3.2.7 The Five-Module Framework for Internet Application Development

The five-module framework for Internet application development [48] proposes a novel architecture for Web applications. It extends the commonly used three-tier architecture (presentation, business logic, and system layer) into five modules: the presentation, UI component, business logic, data management, and system infrastructure module. The underlying theme of this work is the tight coupling of the various aspects involved in Web development. In the five-module framework, a looser coupling and an object/component-based approach towards Web development is envisioned resulting in the ability to independently evolve the business logic.

The newly added user interface component layer consists of a set of objects that represent a page in terms of its user interface. The presentation layer then transforms each such object into its corresponding HTML representation. The claimed benefit is that UI components can be added dynamically while not modifying the presentation information. This is only true if all possible components are known in advance or only components of already existing types are added. We doubt that this behavior justifies a separate layer.

A further decoupling is attempted by introducing the data management layer that represents a system- and storage-independent view of the code and data in the database. While this allows changing the underlying database system, the benefit over existing standards such as ODBC or JDBC is questionable.

Finally, a broker facility is introduced to decouple any direct method invocations between modules. Thus, any object in a module first contacts the broker to connect to another object offering the desired service. While this approach facilitates the modification and upgrade of the application at runtime, it also introduces a significant overhead and performance penalty.

Though the five-module framework proposes a new way of developing Web applications, it cannot be considered a methodology. It also ignores many crucial factors such as other output formats than HTML, other data sources than relational databases, and navigation design artifacts.

3.2.8 THE WWW DESIGN TECHNIQUE - W3DT

The World Wide Web Design Technique (W3DT) [23] consists of two parts: a modeling part that supports graphical models of the Web site and a computer-based design environment for the implementation of the model site. This is in contrast to the approaches presented so far that mainly focus on supporting the modeling part. One of the goals of W3DT is that the models should be clear and intuitively comprehensible at all times. Another important aspect is its modularity that supports distributed development, hierarchical decomposition of a site, and the development of distributed Web sites.

Another important difference is that unlike RMM and OOHDM, W3DT does not start with a data or domain model but is user-centric in that it models the structure and pages of the final Web site and derives the data requirements from them. It also introduces another level of abstraction by introducing the W3DT meta model that defines how the modeling primitives are related (e.g., a site consists of a set of diagrams, each consisting of pages, layouts and links, etc.).

A concrete Web site is modeled using various modeling primitives such as pages, indices, forms and links to create an instance of the meta model that represents the structure of the site. Further, W3DT distinguishes between static information and dynamic information (i.e., information that is collected or created at runtime) already in the design phase. Similarly, a distinction between static and dynamic links is made.

The methodology does not distinguish separate phases; the building of the site model is the main task. Once the site model is finished, it can immediately be implemented in W3DT's development environment called WebDesigner. Web pages are implemented using HTML templates from which skeleton HTML files are generated that have to be completed in an HTML editor. Separation of concerns is supported only marginally. Layouts are separated from the actual page but consist only of attributes for the background color, the background image, a header line and a footer line. While the simplicity of the method makes it easy to understand and the models simple, it is not suitable for the much more sophisticated requirements of today's Web applications.

The extended WWW design technique [12, 129] adds a new development process including an analysis, design, implementation and recurring evolution phase. It also conceptually separates the content production from its technical separation (i.e., the roles of the content manager and the programmer are separated). Further, user input processing is explicitly included but only to the extent as user actions directly manipulate the content of a database. More sophisticated interfaces to business process and their application logic are missing. The simplicity of W3DT models is dropped in favor of more complex diagrams. A new notation explicitly includes priorities, responsibilities and expected maintenance effort on the page level.

However, key requirements such as separation of concerns or support for reuse are not addressed. The method also states that it is intended to be integrated into higher level methodologies. An implementation environment similar to the one included in W3DT is missing.

3.2.9 LIFEWEB: AN OBJECT-ORIENTED MODEL FOR THE WEB

LifeWeb [119, 144] proposes an object-oriented model for the Web that not only models single Web sites but also the Web in its entity. It further tries to capture the dynamic nature of the

Web (i.e., its fast growth, rapid changing information services, etc.). LifeWeb tries to model this dynamism by introducing the concept of *life*. Similar to the concepts in genetic programming, it tries to exploit fundamental principles of life such as reproduction, interaction, existence and evolution. Objects are enriched with object genes that carry meta-information about the object, the class hierarchy is compared with the species hierarchy in biology and so on.

The model consists of four levels: the instance, genetic, meta-genetic, and meta-meta genetic level. The instance level comprises the actual Web documents as seen by the client, the genetic level deals with the document's object (in the case of LifeWeb XML) representation on the server, the meta-genetic level defines the DTDs for the XML documents, and the meta-meta genetic level defines the grammar to express DTDs.

The evolution process is incremental and defined by the evolution on every level. *Evolutionary thresholds* model the likeliness of evolutionary actions. Only if a certain threshold is reached, evolution becomes possible and changes are propagated to higher levels. Thus, a DTD for an XML document can effectively change if the threshold for such a change is reached. According to the model, all documents based on this DTD would be changed, too.

Evolution actions happen in response to user interaction or long-term behavior (e.g., how often a page is requested, which content is popular, etc.) and cumulate until a evolutionary threshold is reached.

Technically, LifeWeb formulates publication on the Web as an object-oriented activity using the same model for single sites as for modeling the Web itself [120]. It introduces structural and presentational subclasses but maps the object hierarchy to a set of XML documents. Separation of concerns is achieved to some degree as result of the distinction between structural and presentational subclasses.

LifeWeb is not a full methodology since it only proposes a model but no concrete development process. Though it is a fact that the Web evolves and changes all the time, it is questionable whether it is an advantage to think of a Web site as an autonomously evolving entity. Usually, this automatic behavior is not desired; however, the concept of evolutionary thresholds can also be mapped to access statistics, Top 10 page listings, and similar artifacts that are commonly found on today's Web sites.

Reuse is supported to the extent that the class hierarchy supports it. As mentioned above, LifeWeb does not cover the whole life-cycle of a Web application but merely the modeling and implementation part. Analysis, test, or maintenance are missing.

3.2.10 Conceptual Modeling and Web Site Generation using Graph Technology

In [69] the authors discuss an approach towards developing Web sites that exploits the power of graph theory. As in many state-of-the-art methodologies, a conceptual model is used to present the application domain and understand the relationships and constraints. Correctly the authors point out that such a model is well suited as a means of communications among the roles involved in the development process (content managers, graphics designers and programmers).

Similar to the hypertext model where nodes are connected via hyperlinks, [69] models a Web site as a Web of vertices and edges. The modeling approach is based on EER/GRAL [80] and thus again especially suited for structured information domains. The conceptual graph model also provides type information for the class of possible Web sites conforming to the conceptual model; as a consequence automatic consistency checking is possible. All instances of Web sites that conform to the conceptual model are said to be in a valid state with respect to the model.

Once the conceptual model is stable, an actual instance of the graph model is created by acquiring and populating the conceptual model with actual content. It is interesting to note that not only the conceptual model but also the concrete instances (i.e., the content) is modeled and stored as a graph. XML documents are used as generic data format though the mapping of the semi-structured XML data to the EER/GRAL based conceptual model is only possible if the document type definition of the XML content is significantly close to the structure of the conceptual model. In other words, only XML documents that represent the conceptual model can be used.

The site generation phase follows the modeling phase. In this phase the graph's content is extracted using graph queries and written into text files. The result of the queries is a set of so-called XML bags, XML documents with a specific structure reflecting the content in the graph. This intermediate step is necessary to consecutively apply XSL transformations to the content and generate the final Web pages.

Apart from neglecting requirements analysis, testing and maintenance, the main point of criticism is that only static pages are generated. Although the authors claim that their ideas can be extended to also cover dynamic pages, it remains unclear how this could be done in a modular way. Separation of concerns is only supported for content and layout and these artifacts can be reused. Navigation is directly inferred from the edges in the graph which roughly corresponds to the approaches taken in other methodologies. The application of the same concept (i.e., graph modeling) for both the conceptual and the implementation model is remarkable since it gives you automatic consistency checking for free. With the inclusion of dynamic pages, however, this becomes much more difficult.

3.2.11 The Web Modeling Language (WebML)

The Web Modeling Language (WebML) [34, 35] is a language for high-level, conceptual modeling of Web sites. The model-driven development approach is based on distinct orthogonal perspectives manifested in four models: the structural, the hypertext, the presentation, and the personalization model.

The structural model captures the content (i.e., data model) of the site using entities and relationships. It uses existing notations such as E/R models or UML class diagrams for this task. The hypertext model consists of the composition and navigation sub-models. The composition sub-model specifies what pages will make up the site and what content is included in the pages. The navigation sub-model talks about the relationships of pages and content units in terms of contextual and non-contextual links. Contextual links are derived from the relationships in the data model and connect semantically related items of information. Non-contextual links simply

connect non-related pages (e.g., a link to a site's search engine that is reachable from all pages). Next, the presentation model describes the graphical appearance in an abstract XML syntax thus remaining independent of the target output language and device. Finally, the personalization model supports the definition of users and user groups and the dynamic adaptation of the site based on high-level business rules (e.g., the shopping behavior of a user can automatically make him member of different user groups).

The proposed development process is an iteration of requirements collection, data design, hypertext design in-the-large and in-the-small, presentation design, user and group design, and customization design. When the requirements are established, the data design models the underlying information domain. As other approaches, WebML is especially well suited to highly structured data domains and data-centric applications where rich relationship and constraint information is available. It defines six units to model data representation based on single or multiple entities, relationships and lists of information items. The hypertext design in-the-large talks about the whole site, the pages that should be included and their relationships. Hypertext design in-the-small is consecutively concerned about single pages and page-level refinements. The navigation model created in the hypertext model can be used to express semantically rich navigation structures such as multi-step indices, filtered indices, indexed guided tours and rings. The presentation design adds the presentation information to the pages and the user and group design creates user profiles based on the intended personalization behavior. Eventually, the customization design takes advantage of the user profiles and defines business rules that specify how the site is to be customized based on the user profiles and customization requirements.

Given a structured information domain, WebML achieves many of the desired characteristics presented in 3.1. It clearly separates the concerns content, structure, presentation, navigation, and personalization. Especially personalization is not covered by any of the other methods. The level of abstraction reaches from high when modeling the site in-the-large down to the implementation level where the Toriisoft development tool turns a WebML specification into an actual implementation. Reuse of design artifacts is not as clearly supported as in other approaches. Further, the modeling process only supports a limited number of concepts (e.g., the six data units, the abstract layout specification, the pre-defined classes of links, etc.). While this makes it easier to deal with the method and to create a supporting tool, it might fall short in covering the sophisticated requirements of today's Web applications. Also a discussion of dynamically created pages and how to integrate (existing) application logic is missing.

In other areas, however, WebML proposes innovative concepts such as the notion of a valid hypertext that support automatic checking of a Web site for design flaws. Further, the explicit support for generating a default hypertext from the specification to validate the model with a working prototype at all stages is a valuable feature. It strongly supports rapid prototyping and the exploration of design alternatives. Proposed extensions also include the introduction of operations to model user-triggered write access to the underlying (relational) data repository, i.e., allowing users to actively modify the content of the site.

3.2.12 WebComposition: An Object-Oriented Support System for the Web Engineering Lifecycle

WebComposition [56, 58, 64, 65] is an object- and component-oriented model for Web development. The changing characteristics of Web application from mere information systems to full-fledged software applications and the fast pace of changes on the Web result in WebComposition's goal to strongly support reuse and maintenance scenarios. A better modeling of the Web's coarse-grained, document-oriented structure to preserve design decision artifacts and component definitions is identified as critical success factor.

To achieve its goals, WebComposition defines component-based Web Engineering as a disciplined Web Engineering approach that supports reuse of components and domain knowledge. WebComposition supports components on different levels of abstraction. Its components can be as fine-grained as single attributes (e.g., the font attribute for an HTML tag) or as large as whole pages. Components have simple properties (e.g., key/value pair attributes, text content, etc.) and a `generateCode()` method to produce the components representation in the target language.

A component is called primitive if it is not further decomposed; composite components consists of one or more other (primitive or composite) components. Further prototype inheritance is supported using dedicated prototype components other components inherit from. This is similar to having HTML template files and deriving instance pages by refining the template. The inheritance and composition hierarchy of all components and together with their definitions are stored in the component repository effectively capturing the design artifacts required to generate the Web site. The component repository [59] uses an extensible architecture consisting of the actual component stores, meta-data stores and the actual repository tool that combines component stores with meta-data stores to provide sophisticated query and retrieval mechanisms for components.

The development process starts with the creation of the WebComposition model containing all components and their inheritance and aggregation relationships. While WebComposition suggest a not further specified structured approach that incrementally refines the model, it also acknowledges the requirement and need for a tool to re-engineer existing HTML designs. Component factories are used to create default content component from an underlying relational database. When the initial model is created, a refinement and abstraction process towards the final component model concludes the design phase. If the model is fully specified, the model can be executed at runtime or, in the case of more static information, file resources containing the Web pages can be generated and stored in the file system that acts as a cache for the Web server. An important feature of the WebComposition system is that it supports evolution and maintenance activities at runtime. As soon as the component specification is updated (and existing file resources re-generated), the changes are reflected in the final Web site. This is possible because the component repository serves as store for the design changes as well as a source for the Web page delivery process.

WebComposition components are defined in the Web Composition Markup Language (WCML) [57, 60], an application of the XML. A component has a unique identifier, a set of attributes, can inherit from one or multiple prototype components and can itself be a prototype

for other components. Components can override inherited attributes and be parameterized resulting in a component template mechanism that supports the implementation of design patterns on top of WCML components. WCML further supports the definition of hyperlinks on the conceptual level by linking to other components. These links can be defined outside the components themselves, resulting in a powerful concept to define different navigation structures for a given component model. The decorator pattern is used to extend existing content components with device-specific layout information. Thus the same content can be presented differently depending on the target device.

The WebComposition system provides a powerful component and composition technology that supports reuse and maintenance of components. Separation of concerns is not directly supported; layout information can be separated using the aforementioned decorator pattern. The design process supports a smooth transition from the component model towards the implementation. The component model, however is close to the implementation since it already contains the final content and layout information. A real conceptual model and higher levels of abstractions are not supported.

3.2.13 Synthesis of Web Sites from High Level Descriptions

The work reported in [33] attacks the problem of increased complexity of Web sites and their development from a different angle than the research discussed so far. It defines a framework to automatically synthesize domain-specific formal representations of a Web site to make its design more methodical and maintenance less time consuming. Information processing is abstractly represented by computational logic. To prevent developers from having to deal with the logic representation directly, domain specific dialects of a logic are defined.

The development is structured as a simple three-level approach. An informal problem description is mapped to an intermediate representation in a logic dialect and eventually mapped to an actual implementation. The content and structure of the Web site is separated from the layout which is provided by stylesheets. When the intermediate representation is finished, the final Web site can be automatically derived from it. The content itself is modeled using predicate logic. The distribution of the content on Web pages and links among pages are modeled the same way. Thus developers still have to deal with logic and inference rules, though their complexity is reduced by the introduction of domain-specific predicates.

Navigational structures and paths are described using concepts from transactional logic. *Paths* are described by defining the valid sequences to visit Web pages. Sophisticated rules such as 'page x must always be visited before page y' can be enforced using this concept. For large Web sites, however, the rule base gets large. An important concept in the context of links is the distinction between links and operations. Links represent traditional hyperlinks that connect two (static) Web pages. Operations on the other hand refer to executable (CGI) programs that execute some application logic, take input parameters and return a dynamic page as result. This distinction is not found in other approaches but can help to better model the integration of the application logic and the overall workflow of the Web application.

This approach is operational rather than declarative, given a description of how to assemble the final Web page from rules and functional entities. It achieves separation of concern for content and layout by using stylesheets and cleanly defines the interfaces to the application logic. A conceptual model, other than the domain-specific predicates, is missing; the informal description is directly converted into the intermediate representation. Though the intermediate representation serves as input for the automatic generation of the final site, it is doubtful whether the complexity of this representation and the absence of a conceptual model achieve the desired effect of making design more methodical and easing maintenance.

3.2.14 The Extensible Web Modeling Framework (XWMF)

As several other methods, the Extensible Web Modeling Framework (XWMF) [96, 97] proposes an extensible, conceptual model of a Web site consisting of classes and objects similar to concepts in object-oriented software development. Unlike the other approaches, however, XWMF is an application of the resource description framework (RDF) [101] and defines an extensible set of RDF schemata and descriptions for defining Web applications. The idea behind using RDF is to not only get syntactic interoperability as when using XML directly but also get a machine-understandable description of the semantics of the data and the meta-data of a Web application.

XWMF models a Web application as a single graph-based data model facilitating validation of the semantics of the model based on first order logic. The framework defines generic Web application schemata that are specialized into Web application schemata for a given problem domain. These schemata provide the vocabulary for Web application descriptions that represent the conceptual model of the Web site in terms of components. The descriptions are eventually converted into the final Web application.

Components in XWMF can either be simplexons or complexons. Simplexons are leaves in the graph of a Web application description and contain the actual content. Complexons represent the structure of the application by grouping simplexons and other complexons the larger entities. These constructs are closely related to the primitives and composites in WebComposition (see 3.2.12).

Separation of content and layout is only achieved to some degree. To add different layout styles to the same content component, simplexons have to inherit from the content component and enrich it with the layout information. Thus the content can be reused but a clean separation is not provided. Further each such simplexon only contains a small part of the actual layout information (e.g., a table data or row in an HTML layout; the table itself is defined in another complexon). As a consequence of this fragmentation, reuse of layout information is not possible. Also the integration of application logic is done via extensions of existing complexons. Again the problem of fragmentation and the lacking ability to reuse implementation code limit the approach. The only form of implementation reuse is that multiple objects of the same class use the same implementation code. Definition of navigational structures on the conceptual level is not supported.

3.2.15 STRUDEL

The Strudel system [54] takes a data management perspective on Web Engineering. It starts with the definition of a data model and applies data base management techniques to transform the data model into an actual Web site. Unlike most other approaches, Strudel uses a semi-structured data model based on labeled directed graphs and thus avoids the shortcomings of relational models. Data sources are wrapped by translator components that integrate the content into Strudel's internal, graph-based data structure. As Strudel's data model is targeted at the Web, it contains atomic data types such as URLs, images and text. The authors also state that the integration of multiple data sources is a key success factor and likely to become ever more important in the future.

The key idea in Strudel is to separate the modeling of the data, the site's structure and the site's appearance (i.e., layout) from each other. the Strudel methodology starts with a semi-structured data model, the data graph. This graph is similar to approaches such as WebComposition 3.2.12 that use objects and components containing key/value pairs of information. A so-called *site definition query* is applied to the data graph to create the site's structure. This query defines the pages the site will comprise and the content they included. The result of this query is the site graph that is similar to the conceptual models in many other methods except that it is not a relational but a semi-structured model. Since the site graph contains both the content and the structure of the site, it can be transformed into the final Web site by applying Strudel's HTML template language.

Strudel's flexibility mainly stems from the StruQL, Strudel's query language. The complete Web site can be created from several queries, each specifying a smaller part of the overall site. Also different views of the same site or evolution scenarios of a Web site are expressed in creating new or updating existing site queries. Strudel applies graph-based structures for all its models thus facilitating the definition of integrity constraints on top of this structure and incremental updates by evaluating only selected queries.

The application of formatting information is supported by Strudel's HTML template language that applies enriched HTML fragments to the site graph. Such a fragment consists of a traditional HTML code extended by format, conditional and enumeration expressions. Similar to the approach taken in Active Server Pages (ASP) and Java Server Pages (JSP), these expression support if-statements and loops in the HTML fragment and are resolved at page generation time.

As with other methods that use HTML fragments to create the final Web page, layout reuse is hardly possible since the page creation process depends on many such HTML fragments scattered over many nodes in the site graph. Another major drawback of Strudel is that the method does not support dynamic page creation or the integration of existing application logic. Thus Strudel is only applicable to pages with rarely changing, static content. This type of Web sites seems to get less and less frequent while dynamic sites grow in importance.

Strudel makes an important contribution in using a semi-structured data model rather than a relational one. However, the missing support for dynamically generated pages and the use of HTML fragments limit its benefits to a small number of Web sites. The site graph is an important model to communicate the site's structure and distribution of content. Navigation and

linking issues are not covered at all in Strudel and rely on the definition of HTML links in the layout fragments. According to the authors, several potential users of Strudel also requested a graphical representation or easier way to define queries in StruQL.

3.2.16 WOOM - The Web Object Oriented Model

In [36] the authors present a strong case for a structured Web engineering methodology. They correctly claim that often Web development is focused on the fine-grained implementation details and happens in an ad-hoc manner. Software engineering principles such as requirements analysis, specification and design are often ignored. This situation largely stems from the lack of appropriate abstraction and modeling concepts for Web applications. Also the need for a technological solution that bridges the gap between high-level Web site designs and the actual implementation technology is identified.

The authors define a World Wide Web software process that starts with a requirements analysis and specification phase. The needs of the stakeholders are assessed in terms of contents (the actual information), structure (the organization of information in pages), access (navigation and access structures to the information) and layout (the graphical appearance of the contents). Other aspects such as the application logic, multiple output devices or security requirements are not covered. Since WOOM is an object-oriented approach, the structuring of a Web site is modeled using object-oriented concepts: views and relationships. Views select a subset of the contents to be presented and relationships establish semantic connection among such contents (e.g., *is-a* relationships).

The requirements are further refined in the design phase. The overall structure of the Web site, the navigational structures and information organization are described. The design effort can use various design methodology for hypermedia applications such as the aforementioned HDM, RMM, or OOHDM approaches. Together with the advantages of these modeling techniques, WOOM also inherits their drawbacks — most prominently the missing transformation of the conceptual model into an implementation model.

The implementation phase is further structured into several steps: first the conceptual model is mapped onto primitives of the implementation technology. Second, the actual content is added to the site, i.e., the site is *populated*. Finally, the site is delivered by making it accessible to clients using standard WWW technologies. Again the semantically poor and low-level Web technologies require the developer to 'manually' bridge the gap between the conceptual model and the actual implementation.

Unlike many other approaches, the maintenance phase receives significant attention in WOOM. Maintenance is identified as a crucial phase for Web engineering and the WOOM process due to the dynamic nature of Web sites, even more than in the case of traditional software engineering. Drawing from the field of software engineering [67], WOOM distinguishes corrective (bug fixes), adaptive (adjustments to the outside environment) and perfective maintenance (improvements and extensions). In XGuide, we propose a slightly different but essentially compatible characterization of maintenance activities.

WOOM abstracts from the low-level implementation details and introduces a fully object-oriented framework for modeling Web sites. The main entities can be arranged in direct acyclic graphs (DAGs) to express their dependencies and relationships. Containers group sets of resources to make them accessible using a common navigation or access structure and external resources can be integrated using references to them. WOOM also provides mechanism to work on the low-level HTML implementation details (e.g., attributes for the `BODY` tag). Every WOOM resource can appear in several contexts depending on its position in the site DAG. Eventually, the *translate* operation of a resource turns the WOOM entity into an HTML file. This translation operates on the site graph using a set of transformers on the resources. This concept of transformers is the central mechanism to keep the contents separate from its various occurrences on the site and its graphical representation.

The WWW Object Oriented Model is a direct extension of software engineering methods to the Web engineering field. It emphasizes the need for a structured and methodological approach towards Web development and postulates a conceptual model that can be smoothly translated into an implementation. The WOOM approach is strictly object-oriented and does not take advantage of recent Web technologies such as XML or XSL. Separation of concerns is only covered for the contents, the structure of the Web site, and the graphical appearance. The integration of application logic and dynamic pages is not discussed at all.

3.2.17 Comparison of the Presented Web Engineering Methodologies

Based on the comparison of the Web methodologies presented in this chapter, we can make several interesting observations. First, most methods are strongly data-centric and the data model (relational or object-oriented) of the problem domain usually drives the development process. To achieve high user acceptance and good usability, we believe it is beneficial to think of the user's needs first and define the structure and functionality of the site based on the target audience and their expectations. Then the data model and structure of the target domain can be modeled in the required granularity and integrated with the user scenarios.

Another observation is that the integration of existing application logic and its interaction with the Web application is poorly specified in most cases. Methods that rely on object-oriented frameworks frequently require a runtime process to transform the object hierarchy into a Web page. These methods can easily integrate additional application logic, though detailed interface specifications are rare. Another simplistic approach to integrate application logic is to re-route a complete HTTP request to some external process that subsequently calls back into the development framework or independently sends a response to the client. If a clear separation of concerns and the ability to develop in parallel is a goal, this is not acceptable.

Parallelism and time-to-market, which are key foci of this work, are only poorly addressed in the presented methods. Most frequently the development process is a strictly sequential set of steps where each step depends on the result of the previous step. Also sequences of process steps are commonly iterated to further refine a model or integrate maintenance and evolution activities. It is evident that a Web engineering process can never be fully parallel. At least at the

Table 3.1: Comparison of Web Engineering Methodologies.

	Concerns	**Reuse**	**Complexity**	**Appl. Logic**	**User/Data**	**Transformability**	**Parallelism**	**Additional**
RMM (3.2.1)	4 (-)	2	3	1	data	3	1	1
Web-Based Inf. Systems (3.2.2)	3(-)(-)	1	3	2	data+user	2	1	2
OOHDM (3.2.3)	5(-)	2-3	3	2	data	1-2	1	2
Schema-Based Approach (3.2.4)	4(-)(-)	2	2	1	data	3	1-2	2
SWM (3.2.5)	2(-)	1	1	1	user	2	1	2
OO-H (3.2.6)	4	2-3	3	1	data	3	1	2
Five Module Framework (3.2.7)	3(-)	1-2	1-2	2	data	1	1-2	1
W3DT (3.2.8)	3(-)	1	2	1-2	user	3	1-2	2
LifeWeb (3.2.9)	3	1-2	3	1	data	1	n/a	2-3
Graph Technology (3.2.10)	3(+)	2	3	1	data	1-2	1-2	2-3
WebML (3.2.11)	4(+)	2	3	1	data	3	1	2-3
WebComposition (3.2.12)	2(+)(+)	3	2	2	data	3	1	2
High-Level Descr. (3.2.13)	4(+)	2	3	2	data+user	3	1	1(+)
XWMF (3.2.14)	3(+)	1-2	2-3	2	data	2	1-2	2
Strudel (3.2.15)	3(+)	2	3	1	data	2-3	1	1(+)
WOOM (3.2.16)	4	2	3	1	data	3	1	2

beginning, a joint effort has to be made to create a conceptual view of the envisioned Web site or application. Only then and only if appropriate supporting concepts are available, the work can be parallelized. In this thesis, we try to extend the level of parallelism as far as possible, reducing the number of sequential, dependent tasks.

Separation of concerns is the main vehicle to achieve this goal. Many of the existing Web engineering methodologies fall short of achieving full separation of concerns. A prominent example is if a method uses some HTML-based template language that is enriched by special tags to embed content. Clearly, formatting and content information are intermixed in such an approach, resulting in a loss of reuse potential and subsequently prohibiting device independence and a flexible and easy-to-change graphical appearance. A similar problem exists with approaches that embed application logic directly into the content. More subtle examples of not cleanly separated concerns include the derivation of the structure of a Web site from the underlying data model, the implicit modeling of navigation information as part of the layout, and storing layout information (e.g., line breaks) in content databases.

We strongly believe that XML and its related standards are the technologies of choice to achieve separation of concerns on the World Wide Web. As a consequence, this work focuses on deploying XML technologies for Web engineering. We further exploit the advantages of fundamental concepts from software engineering such as interfaces, component-based development and design-by-contract for Web engineering. These ideas are key in our attempt to support parallel development throughout the development process.

The remainder of this thesis presents our approach towards contract-based, concurrent Web development with XML technologies. We first present an overview of XGuide, our method of parallel Web development, before detailing on the concepts and technologies involved in the XGuide process. We further discuss a tools suite supporting the XGuide development process and early results from applying XGuide to the Web site of the Vienna International Festival.

CHAPTER 4

XGUIDE - A NOVEL APPROACH TOWARDS XML-BASED WEB ENGINEERING

The important thing in science is not so much to obtain new facts
as to discover new ways of thinking about them.

Sir William Bragg

There won't be anything we won't say to people
to try and convince them that our way is the way to go.

Bill Gates

Having analyzed existing Web engineering methods, this chapter presents XGuide, our approach towards XML-based, concurrent Web engineering. XGuide promotes a parallel process that reduces development time and extends the concept of separation-of-concerns to a new level by introducing concern specifications called *contracts*. Today the prevailing topics in Web engineering research are conceptual modeling of Web applications and support for separation of concerns. XGuide further adds standards conformance, strong support for the integration of legacy systems and application logic, and user-centered design to the list. Finally, XGuide proposes a way to overcome the gap between the conceptual and implementation world. All conceptual artifacts and concern contracts form the basis for the generation of implementation skeletons that can be directly deployed.

This chapter first gives an overview of the XGuide development process using a small example before each phase is discussed. We focus on the methodological and conceptual aspects of the process and defer a detailed discussion of concern contracts and contract composition to the next chapter. Chapter 5 presents a formal model of contracts, their representation in XML, extensibility properties of contracts and the semantics of contract composition.

4.1 An Overview of the XGuide Methodology

Before we introduce the phases of the XGuide methodology, we briefly discuss what a methodology is and how XGuide fulfills the requirements of a methodology. Kronlof et al. [99] define a method as consisting of the following (as presented in [40]):

- An underlying model, or set of models, which is the class of objects represented, manipulated and analyzed by the method (e.g., a data model).
- A language, or set of languages, which is the notation technique for model(s) (e.g., a data model can be represented by an entity relationship diagram).
- A process model, which is the method's defined stages and the ordering of these stages.
- Guidance, which is the manuals, handbooks, and guides which explain the method.

XGuide complies with this definition. It defines a set of conceptual models on various levels of abstractions and iteratively refines them towards the implementation model. XGuide relies on XML as a notation for the models and defines a partially concurrent development process. The discussion of the phases in the XGuide process is a handbook on how to use the method and its artifacts in real-world projects. XGuide is also consistent with an extended definition from [16] where the method's input and output and the underlying philosophy are added to the list of requirements.

A methodology, however, is not sufficient. In the previous chapter, several methods for modeling, understanding and building Web applications were presented. Still a recent survey [10] shows that they are not used. Almost one-quarter (24.6 percent) of the respondents (companies from general and multimedia industry) stated that they do not use a methodology at all for building their Web presence. The main reason is not so much the difficulty in understanding the methods but that they are perceived as too cumbersome to use for real-world projects. Of the companies who use a methodology to develop Web applications, only 2 out of 19 companies use Web engineering methodologies such as HDM, OO-HDM or RMM. One of the results of the survey is that tool support for a method is necessary—otherwise the method is considered 'useless'. A typical example for such a method is OO-HDM. Three-quarters of these companies use in-house methods rather than existing ones. The reasons given for this behavior are that existing methodologies are not cost effective and too time intensive. Still most of the participants felt that a structured approach would improve the current situation and 77% expect the importance of methods to increase.

With the problems of today's Web development practice and existing methodologies in mind, we designed the XGuide process to be simple enough to be understood and used by Web developers and powerful and flexible enough to cover large and complex domains. Special emphasis lies on the modeling and design phase, the concurrent implementation and the recurring maintenance and evolution activities. We also provide a tool suite to support the XGuide process that is presented in Chapter 6. Figure 4.1 depicts the activity diagram [27, 122, 128] for the XGuide process.

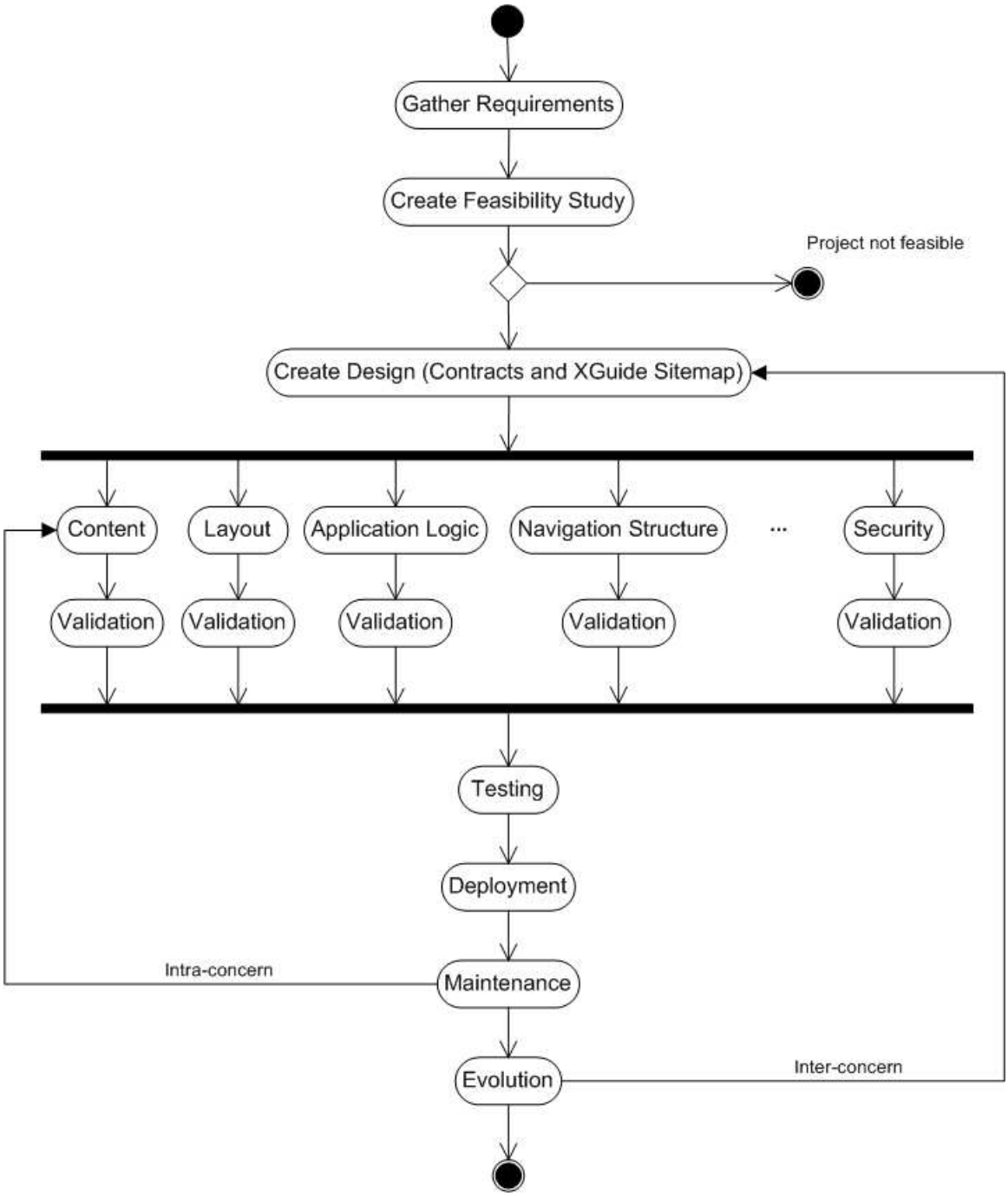


Figure 4.1: The XGuide Development Process

At the beginning of the process, a detailed requirements analysis creates a common vision of the goal and a shared vocabulary and domain understanding among all the parties involved. At the end of the requirements phase, all requirements are mapped to a high-level sitemap. Once the requirements are known, feasibility considerations clarify whether the project can be handled with the available resources (e.g., human resources, knowledge, hardware, technologies, money, etc.) and time. The requirements captured in the high-level sitemap (called the *requirements diagram*) is then refined and expanded to cover all design decisions, components, and dependencies and represents a full-fledged sitemap and conceptual model of the Web application to be developed. From this model, a set of contracts is derived that act as specifications for the separate pages and components. The contracts encapsulate all information about the aspects present in a given Web page (e.g., content, structure, layout, API, etc.) and enable developers to work on them in parallel. Once such an aspect (or concern) is implemented, it can be validated against the contract. All concern implementations are combined and tested locally before they are deployed to the production environment. We distinguish between maintenance and evolution. As indicated in the diagram, maintenance deals with intra-concern activities that are independent of other concerns. Evolution, on the other hand, is an inter-concern task and can be regarded as a miniature project in itself; for major evolution scenarios, not only the design and the contracts are adapted but the process starts anew with a feasibility analysis.

We use a simplified version of the *Orange Juices, Inc.* Web site as an example throughout this chapter to demonstrate how XGuide is used. The Web site backbone consists of a homepage, the product catalogue, a sitemap, and a search facility. The homepage displays up-to-date information about new products or special offers depending on the current date. The product catalogue lists all available products in an overview page and provides links to more detailed product pages. The sitemap is a static page that offers general information about the site and the information and services available. On the search page, a full-text search engine is used to search all pages for keywords. This simple example is only used to demonstrate the XGuide concepts; our experiences using XGuide for a real-world Web project are presented in Chapter 7.

The remainder of this chapter presents the XGuide process with its phases and introduces the terminology used in each phase.

4.2 Requirements Analysis

As in many other software and Web engineering approaches, the requirements analysis phase is the initial phase in the XGuide process. Several roles such as project managers, content managers, graphic designers and programmers are involved in a Web project. Project managers are concerned about the scope and time frame of the project and have to make sure that the project deadlines and objectives are met. Content managers are responsible for the information to be published on the Web. Graphic designers deal with the visual representation and formatting of the content on the Web (e.g., based on a corporate identity policy, accessibility guidelines for the Web, etc.). Programmers provide the application logic that implements the business processes and selects, transforms and combines the content and the formatting templates. The XGuide

process defines a domain analysis activity as first step. This is more than an initial project meeting. Here the business domain of the customer should be presented and discussed. This offers external roles the possibility to get accustomed with the processes and problems in the domain, the terminology used, the IT infrastructure at the customer, the existing in-house knowledge, and any developers that might be assigned to the project or have been maintaining an existing Web site (for more information on domain analysis see [4, 118, 123, 146]; requirements analysis issues are discussed in several books including [29, 81, 98]).

Since the domain of our demonstration example is simple and we do not (yet) include much interaction with other business processes, the domain and its terminology should be easy enough to understand. Still information about the various product categories and products, the existing infrastructure, the data repositories used, and potential problems and experiences with an existing Web site are important to understand before continuing the development process. Lacking this information can easily lead to misunderstandings that result in increased development effort, higher costs and a longer project duration. This does not even include problems on a social interpersonal level when project members in either role feel misunderstood or ignored. Motivated, open-minded team members and an informal and productive attitude and communication among the various roles are key success factors for such a project. Since Web projects can hardly ever satisfy all the wishes and desired features of the customer, it cannot be over-emphasized how important it is to create and maintain an environment where problems and potential solutions or compromises can be discussed. To make this work, however, it requires developers that are flexible enough and willing to contribute to the overall project goals rather than insisting on their particular, however justified, demands. The other important dimension here is a requirements analysis as discussed below that helps to avoid problems and misunderstandings in the first place.

When everybody has a good understanding of the customer's business (or problem) domain, a common vision for the project should be created. This means that all parties should have a similar understanding of what the goal of the project is, what functionality the Web application will provide, and what will be excluded from the project.

In several of our Web projects, we found that it is not sufficient to reach this common vision but that it also has to be documented. Further any additions or modifications to this initial vision of the project again need to be documented and traceable throughout the project. This is especially important if, which is often the case, the customers do not completely know what they want. In a first attempt, we tried to capture the vision of the project by describing the functionality that should be supported by the Web application. We soon learned that customers cannot easily visualize a set of functional requirements and connect them to the resulting Web application. As a consequence, XGuide introduces a simple, graphical notation (the XGuide requirements diagram) that models the pages and dependencies of the Web application and facilitates addition of further requirements to all artifacts. This notation relies on a simple 'boxes and arrows' diagram and is easy to understand and use for customers.

The main modeling artifact in this diagram is the *Simple Page*. A simple page represents a traditional Web page. Typical examples for simple pages are homepages, sitemaps, or search pages. Figure 4.2 shows the simple page element characterized by the single page icon. The only

information about such a page in the model is its identifier, i.e., the page name. Additional annotations and notes capturing page-specific requirements can be added on a separate requirements card for the page. If a page has such a requirements card, this fact is reflected in the diagram by adding the additional requirement icon (a black plus symbol) at the lower right corner of the element. The 'Search' simple page in Figure 4.2 indicates that it has an additional requirements page.

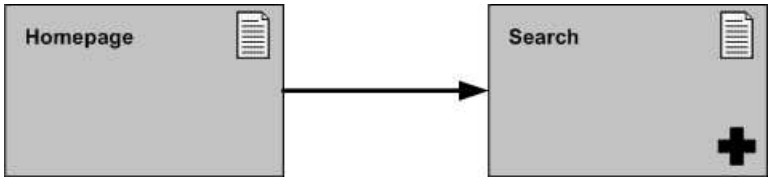


Figure 4.2: A simple page named 'Homepage' that has a navigational dependency to the simple page 'Search' that has additional requirements associated with it.

Navigational dependencies, i.e., hyperlinks, between any two model artifacts are expressed using arrows connecting the source artifact (e.g., page) and the destination of the hyperlink (see Figure 4.2). Such navigational dependencies do not describe what the source or destination element in a page is but state conceptually that the destination page is directly reachable from the source page. To increase the expressiveness of the model, we introduced *External Pages* and *Multi Pages* as shown in Figure 4.3.

External pages (with a gray page icon) are similar to simple pages but are not included in the scope of the project. Examples for such external pages (or services) could be third-party Web sites that act as part of the Web application or legacy systems that have to be integrated. External pages have an associated short description to clarify the functionality of the external entity. Furthermore, additional requirements can be associated with external pages just as with all other diagram artifacts using the additional requirement icon.

Multi pages, on the other hand, represent a set of similar pages. Basically this means that a group of pages share common characteristics (such as layout, structure and navigational dependencies) and only differ in their content. Product catalogues as in our example Web site often use multi pages. They define a single page template and only exchange the content in this template to present all products in a consistent way. Good examples can also be found in other domains with well-structured information such as legal documents, human resources or financial information. XGuide models depict multi pages as rectangular elements with two cascaded page icons. External pages and multi pages also have a unique identifier or name.

With the concept of simple, multi and external pages, XGuide provides a simple but flexible and powerful modeling notation for the requirements of Web applications. Figure 4.4 shows the XGuide requirements diagram for the Orange Juices, Inc. Web site. It defines the simple pages for the homepage, the sitemap, the search page and the product catalogue overview page. It further shows the external service for customer feedback that is not in the scope of the project and the multi page for the product detail pages. The homepage and the search page are further

Figure 4.3: A legacy Web application for customer feedback is modeled as external page named 'Customer Feedback'. The 'Product Details' multi page could be used as a template for a product catalogue.

labeled as having additional requirements: in the case of the homepage it says that the homepage will display special offers depending on the current date; for the search page the additional requirement states that the search functionality must distinguish between a database and full-text search query.

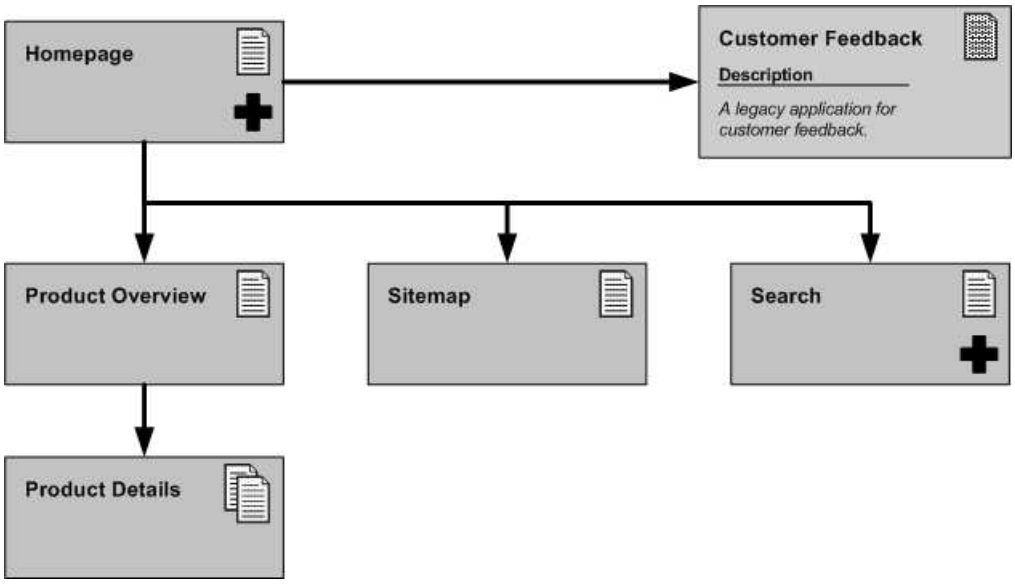


Figure 4.4: The initial XGuide requirements diagram for the Orange Juice, Inc. Web site.

The XGuide requirements diagram, however, is only the first part of the analysis phase. It serves customers and developers as a means of communication but does not yet include non-functional requirements and organizational or environmental constraints. If such a requirement is page-specific (e.g., the performance of the search engine, a specific search engine that must be used, etc.) it can be directly added as additional requirement to the corresponding page. All other requirements such as available hardware, human resources, deadlines, project budget or status reports are collected in a separate requirements document that complements the requirements diagram. Though this depends on the size of the project, we suggest to use the requirements diagram as the main communication facility and thus to integrate as much of the requirements as

possible into the diagram.

In the case of the Orange Juice, Inc. example, the non-functional requirements document might specify that an existing database system on a particular host and/or operating system must be used, the maximum project duration and a weekly progress reporting scheme.

A final note on the requirements analysis phase looks ahead to evolution scenarios for the Web site. By nature, evolution scenarios are not known in advance. However, in some situations ideas for future extensions of the Web application already exist. Our example Web site, for instance, might already plan to migrate the legacy feedback system or integrate an online shopping cart. Such foreseeable extensions might also influence the requirements analysis, e.g., in terms of the infrastructure used or the organization of the data repository. The earlier such considerations are included in the development process, the less effort it will require later on to implement such extensions.

At the end of the requirements analysis phase, the requirements diagram should contain all pages, their navigational dependencies and additional requirements, and the separate requirements document for generic, non-functional constraints. It must be clear to all parties that only what is included in these documents will be in the scope of the project; anything that does not appear in the diagram or the requirements document is excluded from the project.

4.3 The Feasibility Decision

The feasibility decision basically is an initial assessment of the requirements and a commitment to the project. Explicitly including such a decision in a method is unusual but too often feasibility considerations are neglected. In software engineering it is widely accepted and acknowledged that measurements or estimations for the complexity, duration and development effort of a project are important for project planning and management [25].

In the context of Web-based systems only few and preliminary methods exist to define metrics for Web applications. Rollo [126] applies the IFPUG function point method [76, 82] and the COSMIC-FFP [147] from software engineering to Web sites. He concludes that COSMIC is the most flexible approach for counting the functional size of any Web site. In [108] and [109], Mendes et al. state that there is an urgent need for adequate, early-stage effort prediction mechanisms in Web engineering and propose a set of new metrics for estimating the design and authoring effort of Web sites. Their method includes metrics for length size, reusability, complexity and effort. The estimation process is based on linear regression and stepwise regression models. In the future, the authors plan to not only measure the site authoring effort but extend their models to cover the whole Web development life-cycle and be able to compare the prediction results against human estimations.

In XGuide, we apply a more informal feasibility process. We believe that the existing approaches require “too much” mathematical and statistical knowledge and are too time-consuming. Further regression-based models are based on an existing set of homogeneous (i.e., the same group of developers, technologies, etc.) Web applications which is often not available given the many roles involved in a Web project and the fast evolving field of Web technologies.

Instead, XGuide provides a checklist of aspects that might influence the feasibility of the project and explains their impact. It is up to the project manager to evaluate them in the context of the actual project and the available assets and resources.

The importance of the aspects below also depends on the kind (e.g., in-house vs. external) and size of the project being discussed. Thus the following checklist is intended as a way to rule out potential problems as soon as possible.

- **Money.** As so often, money is one of the main concerns. The budget for a Web project consists not only of the salaries of the various roles such as graphics designers, programmers, and content managers. It also has to take into consideration costs for buying new hardware and software, software licenses, consulting, education and training of employees, Internet connectivity, backup facilities, fault-tolerance equipment and the establishment of appropriate (e.g., air-conditioned) environment for the server(s). Especially indirect costs such as on-going maintenance or Internet Service Provider (ISP) fees have to be explicitly calculated or excluded from the project's budget. If, as is seldom the case, an unlimited budget is available, many of the following considerations lapse since almost anything is possible with infinite resources.

- **Time.** Time is a critical resource in all Web projects. The Web evolves and changes so fast that—as a matter of principle—you can never finish a Web-based system early enough. More importantly, however, customers usually have a tight schedule for a Web project and often do not appropriately plan for all the project activities. A too tight project schedule is a hard problem. Customers frequently do not want or cannot extend the time frame and simply adding other resources often does not solve the problem. Especially in today's Web engineering approaches many dependencies among the involved people exist. For instance, nobody can start working before the graphics designer has finished the layout templates. With XGuide and its support for parallel development, we hope to alleviate this problem.

- **People and Responsibilities.** Depending on the size of the project, the number of team members and their distribution on the separate roles can be a non-trivial task. This especially becomes an issue in larger projects where it is not obvious whether adding more people to the content management or programming role would increase the role's productivity. Also the clear and unambiguous assignment of responsibilities is important to ensure traceability of progress, have a dedicated contact in the case of problems and avoid misunderstandings within and among roles. For small projects, it is important to ensure that all roles have been assigned team members and that this assignment and the responsibilities of every role are made explicit.

  Another consideration with respect to human resources is whether project members are guaranteed to work full-time on the project or not. This also has to involve a mid-term and long-term analysis to measure the likelihood of people getting assigned to other projects or tasks. If somebody is also involved in other projects, their project deadlines and priority within an organization should be checked.

Finally, the maintenance and evolution of the Web site when the project is finished should be discussed. Frequently, Web development is regarded as a one-time effort. This is clearly not the case. When the initial design and implementation effort is finished, the maintenance of content, bug fixes, functional extensions and adjustments of the graphical appearance will continue to happen on a regular basis. Thus human resources have to be allocated for these tasks.

- **Dependencies.** XGuide distinguishes two kinds of dependencies: external and internal dependencies. An external dependency describes a relationship with a service of an external third party. External dependencies are important since they usually cannot be influenced and hence potentially dictate interfaces and technologies that must be used. They might further limit a system's availability and flexibility in terms of service evolution or software upgrades. Internal dependencies, on the other hand, refer to already existing in-house systems. This can be an already existing Web site where parts of the content or a dedicated service have to be reused. This could also be a legacy application at the back-end such as a database repository, a business process or sever application.

- **Quality of Service.** As for any distributed system, quality factors such as performance, scalability, availability, security and fault tolerance are important design and implementation criteria. The desired properties for these characteristics not only influence the hardware infrastructure but might also affect the software design and the choice of the implementation language and/or technology. Dependencies of any kind as outlined above often limit the possible choices for some quality factors, e.g., by introducing a single point of failure or providing poor performance characteristics.

  In the area of Web-based systems, scalability and performance are especially important. Often Web sites are initially small and easy to maintain but soon tend to grow quite radically. If a Web site is popular, the number of requests to be served can also increase quickly over very short periods of time. If the project has the potential of growing rapidly or receiving a flood of requests, it should be planned for such developments right from the beginning. In other projects (e.g., intranet Web applications) this is not an issue.

  The quality of service attributes also affect the hardware infrastructure necessary for the project. Most performance, scalability, fault-tolerance and security requirements need some sort of hardware or network device to be satisfied.

- **Know-How.** The know-how of the people involved in the project plays a major role in the overall planning of the project. With an experienced team of Web developers, a project can well be finished in half of the time compared to a non-experienced team. Experience in this context is primarily targeted at experience with Web technologies (e.g., HTML, XML, CSS, XSL, etc.) but also includes experience with software tools, team work, and Web projects in general. Lacking know-how either requires additional training which costs time and money or rules out the unknown technologies and tools a priori.

- **Technology.** On the technical side, the implementation technology and platform is the most important choice to be made. Which of the many available implementation tech-

nologies is the best option for a given Web project, often depends on the requirements and complexity of the project. For Web-based information systems, for instance, support for database access is critical. For more process-oriented Web sites, a flexible integration of application logic is more important. Obviously, also other aspects such as performance and scalability requirements, tool support, third-party services that must be integrated and available know-how influence this decision. Particular attention must be paid to the software supporting a technology: Is it stable? Is it going to be available in the future? What is the copyright and/or licensing situation? A software product that will not exist any more a year from now is hardly a good choice for a Web site with a long lifetime.

The development and deployment platform might also eliminate some technologies and usually depends on the predominating platform in the organization. Apart from the implementation technology decision, standards and tools for documentation, reports, conceptual modeling and communication should be available.

The XGuide process suggests to deal with each of the above criteria and to record the results in a separate document. This feasibility document should explicitly list all the requirements and prerequisites for the project, e.g., the project is planned for 10 people who work full-time on the project and stay with the project for its full duration. Even better, the names of the people with their qualifications and why they were assigned to the project should be included. This document also forms a good basis for discussion with the various roles to provide a rough overview of the required resources and infrastructure. It is not, however, a detailed project plan. In XGuide we do not propose yet another way of defining a detailed project plan. Instead, we regard the project plan as a refinement of the feasibility document. The format and level of detail of the plan depends on the size of the project, the experience of the project manager and the modus operandi of the customer.

When the feasibility document is finished, the project commitment statement is formulated. Depending on the outcome of the feasibility evaluation, the project can be canceled, the scope of the project can be adapted or the stated project requirements are approved.

The cancellation of the project has to be considered if the expectations of the stakeholders differ too much from the expected course of the project. Examples for this scenarios are unsustainable deadlines or too low a budget for the required functionality. More frequently, however, a redefinition of the project goals and requirements in terms of the available resources and time takes place. Alternatively, extensions of the project time frame or an increase in the budget are necessary to fulfill all requirements. Of course, a project can also be approved as-is if the initial planning and assessment was accurate. This, however, requires an experienced project team and stakeholders. In our experience, the immediate approval of a project proposal is rare; the modification of the requirements or the scope of the project is more frequent.

Eventually, if the (initial or redefined) project is approved, the requirements in the form of the XGuide requirement diagram and the descriptions of the additional requirements must be cast into a conceptual model and design. The next phase, the design phase, refines the requirements model to a full conceptual model of the requirements that also captures design decisions such as componentization and navigation contexts.

4.4 Conceptual Modeling and Design

The previous two phases, requirements analysis and feasibility evaluation, directly involved the customer of a project. The diagrams and notations presented so far were thus intentionally kept simple to facilitate unambiguous communication with the various parties involved in the project and to capture the high-level requirements of the project.

In this phase, the architecture and design of the Web application take center stage. The architecture frequently is a classic three tier architecture. A persistent storage layer (e.g., file or database system) is the first layer. The Web or application server is the second tier. The client browser is responsible for the user interface and represents the third tier.

Compared to the requirements phase, the design is a more complex process. In the field of traditional software engineering, established design notations for software systems exist—among which the Unified Modeling Language (UML) [122, 128] has a prominent position today. In Web engineering, however, a widely accepted notation for modeling Web applications does not exist. Instead, every approach defines its own notation to best represent the main focus of the respective method. Well-known examples of such notations include the graphical modeling notations of OO-HDM [130,133], RMM [78,79] and WebML [34]. Conallen [37] takes a slightly different approach in extending the UML towards a Web modeling language.

In the context of the XGuide process, they all share the same inadequacy: they do not support separation of concerns on the conceptual level or the notion of contracts in their models. Though separation of concerns is identified as important design criteria and is realized to various degrees in the implementation approaches presented in Chapter 3, their conceptual models totally ignore it. Since separation of conceptual concerns forms the basis for contracts and concurrent development in XGuide, we introduce a modeling notation that allows us to identify concerns on the conceptual level.

Apart from the separation of concerns on the conceptual level, it even is still unclear what the appropriate modeling primitives for Web applications are. Most approaches use the notion of pages and hyperlinks among pages. Depending on the method, additional primitives for the navigation or the page content exist. Another important aspect of Web applications is the consistent graphical appearance and navigation structure throughout the whole application. Reusable page fragments that appear on several pages with the same or a similar formatting are the solution to this problem. Component-based approaches model pages as a set of components that represent a page fragment and can be reused independently.

In XGuide, we already introduced the modeling primitives of simple pages, multi pages and hyperlinks in the requirements phase. In the design phase we introduce new modeling artifacts and refine the requirements diagram to include additional design-related information. The design phase has two sub phases: the *design in-the-large* (on the architectural level) and the *design in-the-small* (on the module level). Design in-the-large refines the requirements diagram towards a conceptual model of the application; Design in-the-small focuses on the design and specification of single pages and components rather than the whole application or site.

4.4.1 DESIGN IN-THE-LARGE

Starting with the requirements diagram, design in-the-large extends the diagram with additional artifacts to better model the structure and functionality of the application.

Following the component-based approach, XGuide supports so-called *Web Components* to model reuse and composition relationships. We think of a Web component as a reusable, configurable page fragment that can be reused and composed with other components to form the actual Web page.

Further generalizing the Web component concept, XGuide not only supports composition of Web components into pages but also the composition of Web components into larger Web components that can then be reused as separate entities. Thus a page is a special, top-level component that cannot be further composed. Typical examples for Web components are the aforementioned navigation structure of a site and a common header or footer fragment that appears on all pages. XGuide uses the element shown in Figure 4.5 to define components and assign them a component (i.e., unique) identifier.

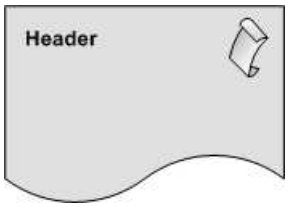


Figure 4.5: A Web component for the header region of the Orange Juices, Inc. Web site.

A first refinement step, thus, is the transformation of the requirements diagram into a *component web*. The component web contains the same artifacts as the requirements diagram and augments it by the definition of reusable Web components.

In the next step, the Web components have to be embedded into the existing pages. To keep the diagram clearly arranged and avoid confusion with navigational dependencies, we do not use arrows to model composition relationships in the diagram. Especially since components are frequently used in all or a majority of the pages, the diagram would quickly be crammed with arrows. Instead, we add a *References* section to the diagram elements that name the Web components a page or component references. Figure 4.6 shows the extended versions of the diagram elements.

Figure 4.7 shows the component web for the Orange Juices, Inc. Web application. It defines the *Header* Web component that is referenced from all pages in the diagram. Thus the page elements list its component identifier in their *References* sections. (For simplicity reasons we did not include additional components for the footer, the navigation bar, etc.)

When all Web components are defined and correctly referenced by the respective pages, the modularization of the site is finished. Depending on the experience and mode of operation of all the parties involved, Web components can already be introduced in the requirements diagram to

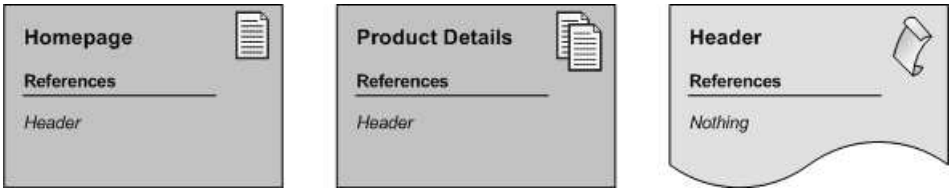


Figure 4.6: The updated icons for the XGuide elements including the *References* section.

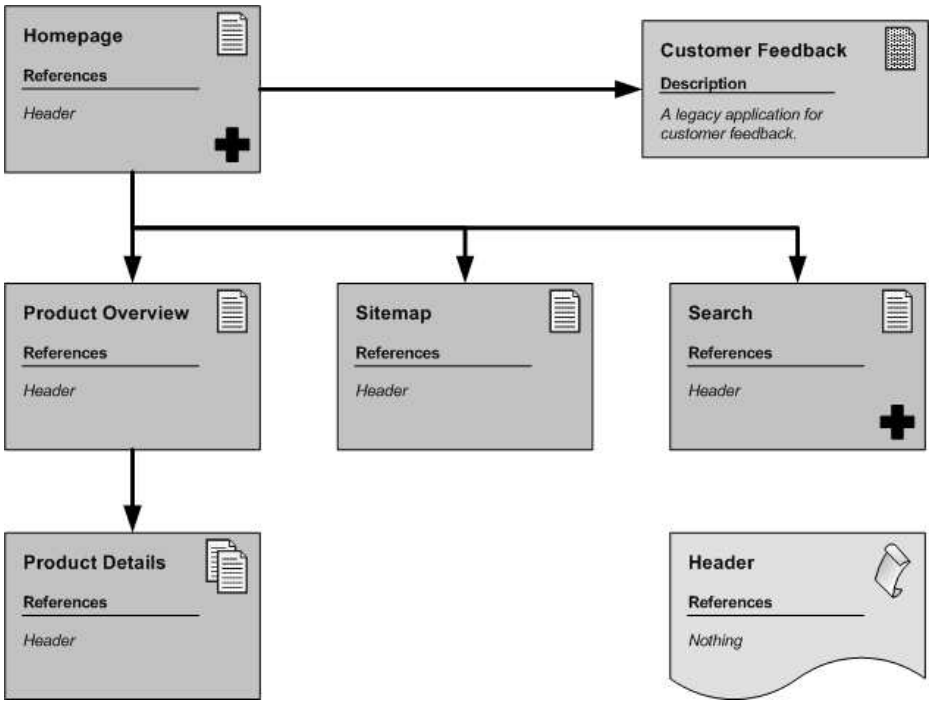


Figure 4.7: The component web for the Orange Juice, Inc. Web site.

better visualize the physical artifacts representing the page in the discussion process. Another alternative is to introduce components along the way rather than as a post-processing measure after all pages have been defined.

The second concept introduced in design in-the-large are *application logic processes*. With the increasing use of dynamic technologies such as ASP, JSP or Java servlets, a growing number of Web pages are generated dynamically at runtime rather than delivered from a static file. The creation of pages at runtime follows a common pattern independent of the concrete technology used. The Web server first receives a request for the page. It then identifies the process or application that is responsible for the requested page and forwards the request. The so-called request handler then executes some application logic to generate the result page (e.g., queries a database and lists all hits in an HTML table). The request handler then directly returns the result page to the client. Application logic processes model the functionality of the request handlers that take a request as input and produce a resulting output page.

Figure 4.8 depicts the diagram element for an application logic process. It has a unique name and a short description of its functionality. Application logic processes are referenced from pages or components and produce another page as output. In the example, the search input page references the application logic process representing the search engine, which in turn outputs the search result page.

Figure 4.8: The 'application logic process' diagram element used by a search page and producing the search result.

Application logic processes provide details on the transition from one page to another. On the conceptual level, they are merely optional constructs. The relationship of the search input and search result page could equally well be modeled without an application logic process since the transition from one page to the other is not the main focus of the design phase. Further it is implicitly clear from the context that a functional unit has to be inserted in between. Nevertheless, in complex scenarios application logic processes are a good means to clarify page relationships and to make them explicit.

The final diagram artifact we introduce for design in-the-large is the *proxy element*. A proxy has a unique identifier and is a representative of the element with the same identifier. It is used to keep the diagram readable, to avoid too many arrows through large parts of the diagram, and to facilitate referencing of elements if the diagram is split across several pages. The diagram artifact for the element proxy is shown in Figure 4.9.

Having split the pages into reusable components and having defined their composition dependencies, the design in-the-large is finished. Design in-the-small then concentrates on the fine-grained specification of the identified artifacts (i.e., pages, components and application logic

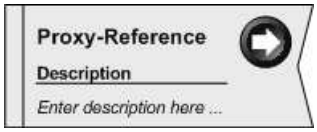


Figure 4.9: The graphical representation of the proxy diagram artifact.

processes). Before we continue with the discussion of the design in-the-small activities, however, a separate subsection introduces the notion of input and output interfaces of components, pages, and processes.

Input and Output Interfaces

So far we only introduced optional application logic processes, but otherwise did not distinguish between static and dynamic pages or any information flow between them. The content of dynamic pages typically depends on some input values provided by the framework or the user. If a page has no such input requirements, there is no reason to create it at runtime. Here are some typical examples for the input requirements of dynamic pages:

- A search result page uses a keyword or search expression as input to display the corresponding search result.
- A currency converter page, for instance, needs the amount to be converted as input.
- A page displaying a user's shopping cart uses the contents of the shopping cart as input.
- A grading service page displaying all grades of a student needs the student's name or identification number as input.
- In the Orange Juices, Inc. Web site example, the product details multi page requires a product number as input to determine which product's details it should present.

As a consequence we classify a page without input requirements as being *static* and a page with at least one input requirement as being *dynamic*. In XGuide, we denote all input requirements of a page (i.e., the set of arguments that the page needs to be created) as the page's *input interface*. Unlike interfaces in object-oriented programming languages, XGuide's interfaces only specify data (i.e., variables) but no behavior (i.e., methods). The behavior of an input interface is implicitly clear since the only supported operation is the creation of the page.

To incorporate input interfaces in the conceptual model, we further extend the diagram to contain the input interface for all components and pages. Input arguments for diagram artifacts are defined in the interface dialog of the element. Figure 4.10 shows this dialog for the product details multi page that indicates that it needs an input argument of type integer.

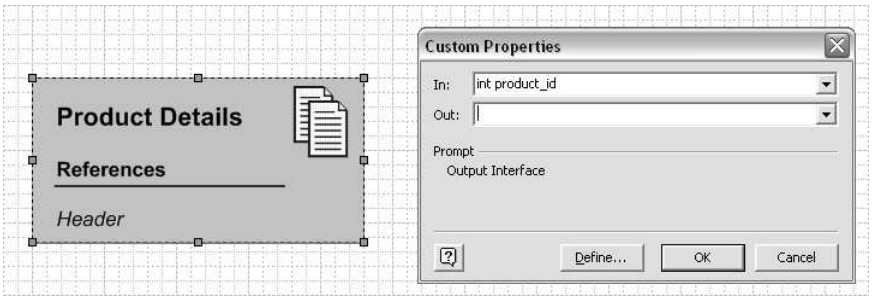


Figure 4.10: The interface dialog for the product details multi page indicating its input requirements.

In addition to pages, we also classify components into static and dynamic components depending on their input requirements. A common use of input interfaces on components are customized navigation bars or menus. Such components take a page identifier as input and render the currently viewed entry in the menu differently than the others (e.g., using a different color). In the case of a hierarchical menu, the branch in the menu tree corresponding to the current page is displayed while the other branches stay collapsed.

After the specification of the input interfaces for the pages and components in the conceptual model, the question arises where the values for the input parameters come from.

In the case of pages, two options exist: (1) the user enters them in a Web form and submits them as input for the subsequent page and (2) the values are derived from an external source in the environment (e.g., the current time or date) not requiring user interaction. In the case of components, a third option exists. The component can derive the value for an input parameter directly from the embedding page. Consider a component displaying a customized navigation bar depending on the page it is embedded in. If the enclosing page specifies its identifier (e.g., as an attribute of the page's document element), the component can derive the value of its input parameter from that value. It does not require user interaction nor an external source.

As mentioned above, a user can only provide input values via Web forms that are embedded in pages or components. XGuide introduces the concept of an *output interface* to describe the set of values a page (i.e., the user filling out the form) provides to the outside world. Since a page can contain several Web forms, it can also have multiple output interfaces as opposed to a single input interface as discussed before.

Consider the simple example depicted in Figure 4.11. A simple login page is used to let the user enter her name. Its interface dialog specifies no input requirements and a single output parameter *name* of type string. When the form is submitted, the *name* output parameter serves as input argument for the subsequent welcome page that takes the submitted value to greet the user with a personalized welcome message. The interface dialog for the welcome page depicts the required input parameter and an empty output interface. Note that the output parameter of the login page and the input parameter of the welcome page have both the same name and the same type. This correspondence is important to be able to match input to output parameters.

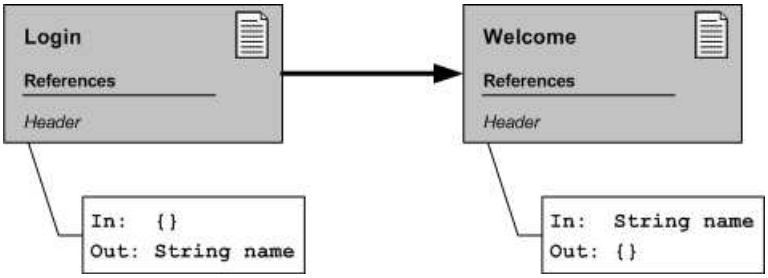


Figure 4.11: A simple example demonstrating the definition of an output interface.

Although this example nicely demonstrates the concept of input and output interfaces and their dependencies, in practice a direct match is rarely found. More frequently, the output interface of a page (i.e., the values provided by form fields) do not directly match the input requirements of the subsequent page. Instead, some application logic first processes the output arguments of a page, transforms or modifies them, and only then provides the required input arguments for the result page. To be able to model such scenarios, we use the previously introduced *application logic process* diagram element (see Figure 4.8).

As the other diagram elements, the application logic element has an input and one or multiple output interfaces. Figure 4.12 demonstrates the use of the application logic element and the definition of its interfaces in the simple search scenario introduced in Figure 4.8. The search page provides a form that takes a keyword as input. Consequently, the search page has an output interface that provides a string (keyword) argument. The search result page, on the other hand, takes a list of matched items as input arguments and displays them. The search engine matches the interfaces; it takes the keyword as input, performs the search and provides a list of matched items as output.

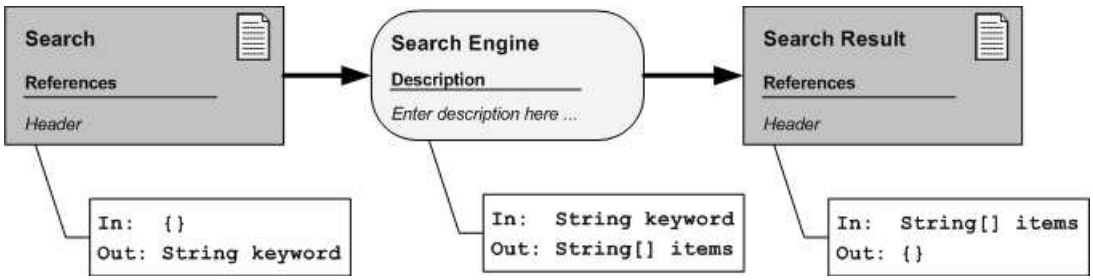


Figure 4.12: The application logic process matches the input/output requirements of the connecting pages.

Figure 4.13 shows an alternative way to model the above search example. In this case, the two pages are directly connected and the application logic element is removed. Since the input and output arguments of the two pages do not match, an application logic artifact that matches the search page's output interface to the result page's input requirements is implicitly added to

the definition. The advantage of this implicit definition of application logic processes is that the diagram remains smaller and easier to understand.

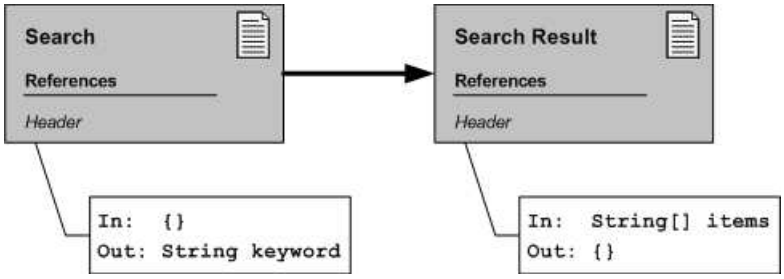


Figure 4.13: The abbreviated notation for the conceptual model of the search example.

In the previous examples, the output arguments of application logic elements were directly used (i.e., specified as input parameters) by the subsequent response page. In complex Web applications, however, input values are often stored across several page requests using cookies or session management. In XGuide, an extended definition of output parameters supports these concepts. If an output parameter is followed by the session qualifier *[S]*, the respective parameter is stored in the user's session for later reuse. If the qualifier is missing, the parameter is not stored across page requests and only available in the immediate response page. Thus *name: string* denotes a parameter that is *not* stored in the user's session, whereas *name[S]:string* adds the parameter to the session.

Finally, we call a conceptual model *strict* if it does not rely on implicitly added application logic artifacts. The advantage of strict models is that the consistency of their input and output interfaces can be checked. The consistency checking algorithm is simple. For every page with an input interface, we first analyze its navigation dependency to identify the pages from which it can be reached. We then have to make sure that the referencing pages satisfy the input requirements of the current page. This means that they must provide a value for all arguments in the current page's input interface. Such a value can either be directly provided by an argument in a page's output interface or by the value of a session parameter. In the former case, the referencing page must specify the output parameter in one of its output interfaces. In the latter scenario, *all* navigation dependencies of the referencing page have to be recursively walked back to ensure that the required session parameter is provided independently of how the referencing page was reached.

Figure 4.14 shows a more complex strict conceptual model. A page $\mathbf{A}$ requires two input parameters $\mathbf{x}$ and $\mathbf{y}$ of types integer and string respectively. The page is referenced by page $\mathbf{B}$ that defines an output interface providing an integer parameter $\mathbf{x}$. Thus the first input requirement of page $\mathbf{A}$ is satisfied. The second parameter ($\mathbf{y}$), however, is not directly provided by page $\mathbf{B}$. According to the above algorithm, we now recursively follow all navigation dependencies of page $\mathbf{B}$ to ensure that a session parameter $\mathbf{y}$ is defined on all paths. In this example we end up at the pages $\mathbf{C}$, $\mathbf{D}$ and $\mathbf{E}$. Pages $\mathbf{C}$ and $\mathbf{D}$ define the missing session parameter and the recursive search stops. Page $\mathbf{E}$, on the other hand, does not provide the parameter and the search continues

recursively—leading to page **F** in this case. Since page **F** provides the correct session parameter, all required input parameters for page **A** were found and the algorithm completes successfully.

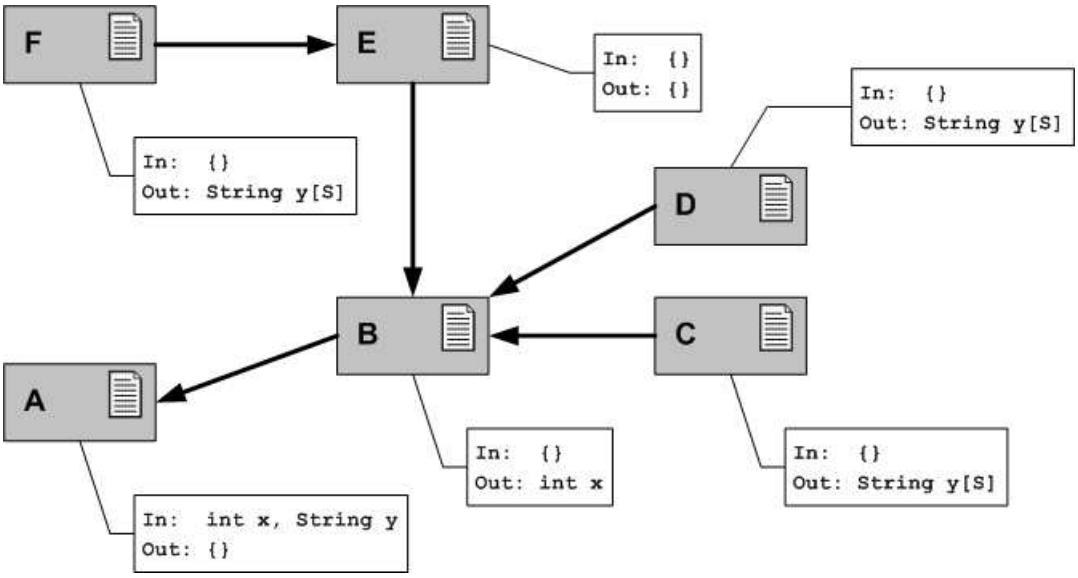


Figure 4.14: An example demonstrating the XGuide consistency checking algorithm.

The definition of input and output requirements resides somewhere between the design in-the-large and design in-the-small activities. Though it defines (page-level) properties of single pages, components and processes, it also requires knowledge about their (site-level) dependencies to correctly connect input and output requirements.

With the definition of the input and output interfaces, the introduction of explicit application logic processes, and a successful consistency check, the conceptual model is complete. We also call this final conceptual model the *XGuide sitemap*. It is a first high-level specification of the Web application and is transformed into its XML representation to serve as input for the more detailed page and component specifications in the design in-the-small step. The structure of the XML representation of a final sitemap including pages, components, composition references, application logic processes, input and output interfaces, and links is displayed in Figure 4.15.

The sitemap's XML representation contains three separate sections for the pages, the components and the application logic processes of the sitemap. The page section is further divided into subsections for the simple, the multi and the external pages in the diagram. Each diagram artifact has an internal (unique) integer identifier and a name derived from the diagram representation. Further pages and components define *References* elements to indicate their composition dependencies. A component reference specifies the component identifier to uniquely identify the referenced component. Navigation dependencies are similarly specified in *LinkInformation* elements that indicate the pages identifiers of the linked pages. Finally, the concept of input and output interfaces is present in pages, components and application logic processes (encapsulated by *Interface* elements). Note that element proxies are not present in this representation. All

```
<?xml version="1.0" encoding="UTF-8"?>
<ConceptualModel xmlns="http://www.infosys.tuwien.ac.at/xguide/sitemap"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://.../xguide/sitemap ConceptualModel.xsd">
    <Title>Orange Juice, Inc.</Title>
    <Pages>
        <SimplePages>
            <SimplePage id="1" name="Homepage">
                <LinkInformation>
                    <Target id="2" type="directLink"/>
                    <Target id="3" type="directLink"/>
                    <Target id="5" type="directLink"/>
                </LinkInformation>
                <References>
                    <Ref id="6"/>
                </References>
            </SimplePage>
            <SimplePage id="2" name="Search"> ... </SimplePage>
            <SimplePage id="7" name="Search Result"> ... </SimplePage>
        </SimplePages>
        <MultiPages>
            <MultiPage id="4" name="Product Details">
                <Interface>
                    <Input>
                        <Param name="productId" type="String"/>
                    </Input>
                </Interface>
                <References>
                    <Ref id="6"/>
                </References>
            </MultiPage>
        </MultiPages>
        <ExternalPages id="5" name="Customer Feedback">
          <Description> ... </Description>
        </ExternalPages>
    </Pages>
    <Components>
        <Component id="6" name="Header">
            <References />
        </Component>
    </Components>
    <AppLogic>
        <Process id="100" name="Search Engine">
            <LinkInformation>
                <Target id="7" type="directLink"/>
            </LinkInformation>
            <Description>Search the site with the given keyword</Description>
        </Process>
    </AppLogic>
</ConceptualModel>
```

Figure 4.15: Structure of an XGuide XML sitemap.

properties and dependencies of proxies are already resolved and integrated in the definitions of the elements the proxies represented.

4.4.2 Design In-The-Small

The sitemap now contains all pages and components that will eventually make up the final Web application. Design in-the-small breaks the high-level sitemap specification down into specifications for the pages and components denoted as *XContracts*. An XContract is structured into several orthogonal concerns that represent different characteristics of a page or component. XGuide currently supports concerns for the content (i.e., information that is offered to the user such as the price of a book), the graphical appearance (i.e., the layout – the formatting information with which the content is formatted for presentation), and the application logic (i.e., the functionality that is necessary for providing the dynamic interaction to the users) of a page. When a sitemap is processed, basic contract templates for all pages and components are automatically generated. The designers then adapt and complete these concern specifications to form the final contracts.

The specifications for all concerns are again an application of the XML and combined in the XContract XML document. They are reusable entities that can be integrated into multiple contracts (e.g., the same XML Schema is used to describe the structure of several pages). Figure 4.16 shows the basic structure of an XContract. The contract elements themselves are defined in the `http://www.infosys.tuwien.ac.at/xguide/contract` namespace. The `<xcontract>` element combines a set of concerns (`<xconcern>`) and a set of composition references (`<reference>`).

The concern elements contain the XML specification for the given concern (often in a different namespace, e.g., an XML schema). An alternative way to specify concerns is to reference an external source that contains the concern specification. A good example is the *structure* concern. The XML Schema that is directly included in the document in Figure 4.16 could equally well be replaced by a reference to an external schema location. Such external concern specifications become reusable since they can be referenced from multiple contracts.

In the composition information, we further define contract composition operations to include the contracts of components in the contracts of the pages (and components) that reference them (depending on the composition dependencies in the XGuide sitemap).

For now, we assume that we already have all XContracts for the pages and components in the sitemap. Concern specifications, more details on XContracts, a formal contract model, and how contract composition works in XGuide is excluded from the discussion of the XGuide development process. We dedicate a separate chapter (Chapter 5) to these key issues. We, thus,

1. defined the concern specifications of all pages and components,
2. created the corresponding XContracts, and
3. composed the XContracts of components with those of the referencing pages.

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract">
 <xconcern name="structure">
  <xsd:schema xmlns:xsd="...">
   <!-- XML schema for component goes here -->
  </xsd:schema>
 </xconcern>
 <xconcern name="interface">
   <!-- definition of application logic concern goes here -->
 </xconcern>
 <compositionrefs>
  <reference to="contract/header.contract">
   <composition type="structure">
     <!-- composition operation for structure concern -->
   </composition>
   <composition type="interface">
     <!-- composition operation for interface concern -->
   </composition>
  </reference>
 </compositionrefs>
</xcontract>
```

Figure 4.16: The basic structure of an XContract.

XContracts form the foundation of XGuide's concurrent implementation phase. They capture all information necessary to implement the various concerns independently of each other. Thus, an XContract for a page has to contain all the information to support the concurrent development and definition of content, layout and application logic.

The translation and extension of the sitemap into a set of contracts is an effective way to describe the transition from the conceptual model towards a concrete implementation. Concern developers directly use the contracts as specifications for their implementation.

4.5 Implementation Phase

Traditionally, the implementation phase of a Web project has three sequential steps. First, the graphics designers work on the graphical appearance of the site and develop layout and formatting templates for all types of pages. Next, the content managers provide the content that should be presented on the pages. Eventually, a programmer inserts the content into the page templates and integrates the application logic.

The major advantage of the XGuide implementation phase over those of other approaches is that the concerns are implemented concurrently. As a result of the strict separation of concerns on the conceptual level and the introduction of contracts, programmers, content managers and layout designers can work independently of each other in the implementation phase. This means that the content, the layout and the functionality of a site can be developed in parallel.

Consider the XContract in Figure 4.17. It defines a *structure concern* and an *interface concern*. From this information, we can derive the specifications for the three implementation concerns: content, layout, and application logic. The content manager only depends on the XML schema that specifies the structure of the content and the valid data types. The graphical designer also uses the XML schema as specification and builds corresponding XSLT stylesheets. The programmer uses the second contract concern, the interface concern, to derive an interface between the page and the application logic. As a result she can use automatically generated stubs until the real content and layout are available.

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract">
 <xconcern name="structure">
  <xsd:schema xmlns:xsd="...">
     <xsd:element name="webpage">
       <xsd:complexType>
         <xsd:sequence>
           <xsd:element name="title" type="xsd:string" />
           <xsd:element name="inputfield"
                        type="xsd:string" maxOccurs="2" />
         </xsd:sequence>
       </xsd:complexType>
     </xsl:element>
  </xsd:schema>
 </xconcern>
 <xconcern name="interface">
   <Interface>
     <Output name="out1" url="/xguide/welcome">
       <Param type="String" name="loginname" />
       <Param type="String" name="password" />
     </Output>
   </Interface>
 </xconcern>
</xcontract>
```

Figure 4.17: A simple XContract for a Web page.

This example already shows that contract concerns not necessarily map one-to-one to implementation concerns. For instance, the content and layout implementation concerns use the same *structure* contract concern as specification. A concrete implementation of the given sample contract is shown in Figures 4.18 - 4.20.

```
       <webpage>
         <title>Welcome Page</title>
         <inputfield>Enter your login:</inputfield>
         <inputfield>Enter your password:</inputfield>
       </webpage>
```

Figure 4.18: The content concern for the contract in Figure 4.17.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="webpage">
    <html>
      <body>
        <h1><xsl:value-of select="./title" ></h1>
        <p>
          <xsl:apply-templates select="./inputfield" />
        </p>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="inputfield">
    <h2><xsl:value-of select="." /></h2>
    <input type="text" name="input{position()}" />
  </xsl:template>
</xsl:stylesheet>
```

Figure 4.19: The layout concern for the contract in Figure 4.17.

```
public class WelcomeServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws IOException, ServletException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String login = request.getParameter("loginname");
        String password = request.getParameter("password");

        if (Utils.validatePassword(login, password)) {
          WelcomePage wp = new WelcomePage(login);
          wp.print(out);
        }
        else {
          ErrorPage ep = new ErrorPage("Login failed!");
          ep.print(out);
        }
    }
}
```

Figure 4.20: The application logic concern for the contract in Figure 4.17.

Figure 4.18 shows the XML content of the page. It adheres to the simple XML schema in the contract. In the general case, XML technologies such as MyXML [85, 93] or Cocoon [107] that support, among others, database access and conditional processing are used to implement the content rather than plain XML documents. Figure 4.19 depicts a simple XSLT stylesheet (welcome.xsl) based on the contract that renders the content as HTML. Figure 4.20, finally, presents the Java servlet implementation class (WelcomeServlet.java) that was derived from the information in the output interface.[1] Note that the class expects the parameters *loginname* and *password* from the page, validates the login information, and then outputs the welcome or error page (not shown in the figure). Note that the welcome page has an input interface requiring a single string parameter (the name of the user currently logged in). Thus it is instantiated with a constructor using a single string value as argument.

Concern validation of the separate implementation concerns (as indicated in Figure 4.1) is the last task of the concern implementation. Depending on the concern, different validation techniques are used. For static XML content as used in the above example, validation simply means to validate the XML content against the schema in the contract. For the application logic implementation, validation means to verify that the implementation only uses the interface variables specified in the contract.

For dynamic content (e.g., retrieved from a database or a Web service) and formatting stylesheets, it is not so clear what validation means. In dynamic pages, the actual content for the page is constructed only at runtime. Thus the schema validation cannot be executed at design time but has to be deferred until runtime, too. An XSLT stylesheet is a loose collection of templates and rules that defines formatting rules based on XPath expressions. Thus it is not possible for such a stylesheet to prove that they correctly process content conforming to a given XML schema. In this worst case, no automatic checking occurs and validation means to manually inspect the stylesheet or content-generating XML document.

When the implementation concerns are finished, they have to be somehow grouped to represent an implementation of the given contract. This group of implementation concerns is called *XPage*. An XPage has an associated contract and provides implementations for all concerns of the page. Normally, XPages are simple containers that represent pages in the final Web site and include references to the (externally defined) implementation concerns. Again, concern implementations (e.g., application logic or formatting instructions) can be reused across multiple pages.

Figure 4.21 shows a sample XPage for the contract and concern implementations of the previous example. The XPage references its contract and provides information about the implementations of the concerns. In the example, we directly included the content concern (*inline concern implementation*) and referenced the layout and application logic concern implementations (*external concern implementation*). This makes sense if we do not expect the content to be reused.

When the XPages for all pages and components in the sitemap exist, the implementation phase is finished and the project enters the next phase: the testing and deployment phase.

[1] We use the Java servlet technology for presentation purposes - the code generation module, however, can easily be replaced to generate source code in a different programming language.

```
<?xml version="1.0"?>
<xpage xmlns=".../xguide/page">
  <concern name="content">
     <webpage>
       <title>Welcome Page</title>
       <inputfield>Enter your login:</inputfield>
       <inputfield>Enter your password:</inputfield>
     </webpage>
  </concern>
  <concern name="layout">
    <ref target="welcome.xsl" />
  </concern>
  <concern name="applogic>
    <ref target="WelcomeServlet.java" />
  </concern>
</xpage>
```

Figure 4.21: The XPage implementing the contract in Figure 4.17.

4.6 TESTING PHASE

As we discussed in the introduction, Web engineering is a multi-disciplinary field. Hence, the testing also distinguishes multiple aspects of a Web application that must be tested: content freshness, correctness and completeness, layout consistency, usability, functional correctness, response time, and many more.

No single testing method exists to date that covers all aspects of a Web application. On the contrary, testing of Web sites and applications is largely ignored today. In this section we sketch a possible testing approach. The granularity and effort put in testing, however, mainly depends on other factors such as the size or complexity of the project. Also the level to which a project is mission critical to an organization heavily influences the testing approach.

In XGuide we already did a first validation step at the end of the (concern) implementation phase. All concern implementations are validated against the respective contract. We already discussed above that the activities involved in contract testing vary from fully automatic to completely manual depending on the concern (e.g., static content can be automatically validated against an XML schema and interface conformance can be automatically checked; contract verification of the layout templates, on the other hand, can only be done manually).

In the context of XGuide's Web components, we follow an approach similar to those found in testing of software components and formal specifications of programs. The assumption is that if the concerns are validated against the contracts and the composition rules are sound, no additional validation needs to be done for the composites [22, 47, 72, 134]. This, of course, requires the contracts and composition rules to be correct. More details on a formal model of concerns, contracts, and contract composition is presented in Chapter 5. As a result, we can break down testing of XPages into testing of implementation concerns.

Testing the internals of pages and components, however, is only the first step. More importantly, the interaction of pages and components on the one hand and the application logic on the

other hand needs to be checked. Also the correctness of the implementation of the application's functionality needs to be 'proven'.

The pages of a Web site and the navigation links between them form a directed graph. If we identify a single homepage as the starting point of the graph, we get a tree similar to the abstract syntax trees of conventional software programs. To fully test the inter-working of all pages, we would have to evaluate all possible paths through the Web site using all possible parameters as input for dynamic pages and application logic calls. Similar to white box testing in software engineering, we can define the *arc coverage* of our test cases with respect to the whole site. One approach could be that each arc must be executed (i.e., navigated) at least once. Another could require one execution of each arc with a set of varying parameters (e.g., exception and error cases, regular values, edge conditions, etc.).

An alternative approach for testing the interaction of pages is *model-based testing*. An abstract model–usually an abstract state machine–models the whole Web site. A state consists of the identifier of the currently viewed page and the state of the model variables. In our case, model variables are the arguments in the input and output interfaces of the XGuide sitemap. They are the only externally visible state of the application. Given the set of states, the links between pages define the possible state transitions. A state transition occurs if the user triggers a new request to the same or a different page. Furthermore, state transitions can have conditions and—depending on external information (e.g., the values the user entered in a form)—different transitions are triggered. As such, the model of the Web application can be directly derived from the sitemap that specifies all pages, component and navigation paths.

Using a model-based testing approach has several advantages. First, test cases for a concrete application can be automatically generated. If we require, for instance, that all navigation links are taken at least once, the abstract model can be analyzed and test cases that satisfy this condition can be automatically generated. A second advantage is that the behavior of the Web application can be monitored at runtime and checked against a model. Thus a deviation from the expected behavior is immediately detected. The implementation of runtime verification (i.e., the comparison of the application's internal state with the expected state as defined in the abstract model), however, is not easy to implement. Section 4.6.1 below presents the abstract state machine language (AsmL) [7–9], a promising approach for model- and contract-based testing and explains how runtime verification is achieved with AsmL.

Testing the correctness of the application logic basically involves the large field of software testing. Various methods exist and can be directly applied to the software representing the application logic. We do not further discuss details of software testing here but refer the reader to related work such as [15, 50, 103, 124, 125].

Another aspect of the testing process is user testing. Depending on the targeted audience of the Web application, users have different requirements and knowledge. The presentation of the content, clear and consistent navigation structures, and the lack of 'surprises' (e.g., unexpected popup menus, misleading link texts, etc.) are of interest from the user's point of view. Also the use of the application on different devices (e.g., desktop computers, laptops, person digital assistants (PDAs), or mobile phones), different operating systems (e.g., Windows, Linux, MacOS) and their various versions, different language settings, different output methods (e.g., voice

output, large text for better readability, text-only output) and different Web browser products (Internet Explorer, Netscape, Mozilla, Opera, Konqueror, etc.) and versions need to be investigated.

Eventually, other aspects such as response times, consistency of the Web site's build process or availability strategies must be considered.

A completely different facet of the testing process is *regression testing* during the maintenance and evolution phases of the project. If single concerns are modified, the application logic updated, or whole new pages and/or components added to the application, regression tests have to ensure that the behavior of the site did not change–or changed only in the expected way.

XGuide proposes XML-based regression tests. When the inter-working of pages and application logic is tested as outlined above (using arc coverage or model-based testing), the input and expected output of the tests is recorded and stored in an XML test repository. Regression testing then means to execute all such tests against the updated application and verify the results. The runtime environment for regression tests can be easily provided using a test driver that simulates the user interaction, i.e., submits the corresponding requests, receives the responses and compares the results with the expected results. While regression suites work well for modifications of the application logic, changes to the content or layout of the site usually require an update of the regression tests, too (since the result pages differ from the original result pages). More sophisticated page comparators that could extract the content of an XHTML page and ignore the formatting statements (e.g., Lixto [13,14]) or focus exclusively on the formatting instructions ignoring the actual content could be a solution. The basic problem with these approaches, however, is that they post-process the result page and do not compare the actual result with the expected result. As a consequence, the post-processing tool might cause or hide potential inconsistencies and conflicts and can only be viewed as a reliable alternative if its correctness is proven.

Extended regression scenarios can specify sequences of page requests to also cover session parameters as used in shopping cart, ticket ordering or similar applications. Regression tests are executed periodically or after every modification of the application. The current status of the regression tests are best exported as separate Web site that allows easy and comfortable checking of the current status of the regression suite.

4.6.1 The Abstract State Machine Language - AsmL

The abstract state machine language (AsmL) is developed by the foundations of software engineering group at Microsoft Research [115]. AsmL is a contract or specification language applying the model-based testing approach. An AsmL specification is only concerned about a software component's externally visible parts, i.e., its public interface. Given the public interface of a software component (a class or a composite of classes that provides a public interface), AsmL provides two alternative ways to define a contract for the component: in the traditional *pre-/post-condition style* or using executable specifications called *model programs*.

Specifications using pre-conditions, post-conditions on methods and class invariants are well-known in software testing. AsmL defines constraints on the method arguments, its return value and the internal state of the component. Further it supports access to an argument's or state variable's *resulting value* in the method's post condition. The resulting value of a variable is the

variable's value after the method was executed. At the same time the variable's original value (i.e., the value it had in the pre-condition of the method) is still accessible. As a result, post-conditions and class invariants can define constraints that relate the original and the new/updated value of a state variable. The set of state variables defined for a component implicitly spans the space of potential model states. A model state is defined by the set of model variables and a unique assertion of values to the variables. Whenever the value of a state variable changes, the model is in a new state. This basic concept is extended by the fact that method arguments and return values can themselves have models. Thus pre-/post-condition expressions cannot only include references to state variables but also to state variables of the models of arguments and return values.

Executable specifications (also called *model programs*) in AsmL use the same basic scheme of models and state variables but instead of defining the state of the model before and after a method execution, it implicitly defines the state transition by specifying a program that transforms the state variables accordingly. Thus instead of defining what the expected state of the model before and after the method invocation is, model programs describe the behavior semantics, i.e., what happens to the state variables when the method is executed. The expected benefit of model programs is that developers usually are more familiar with writing code rather than complex pre-/post-condition constraints which increases the acceptance rate of the technique. Additionally, model programs can be used as high-level implementations of software components until a concrete implementation becomes available. These stubs are useful during the development and testing process and support early testing of implementations (using stubs for all components that are not yet available).

The ultimate goal of AsmL is to provide a mechanism for runtime verification of software components that have an associated AsmL specification. Since AsmL is a full .NET language based on the CLI [49, 114] language, it is compiled into its intermediate language representation just as any other .NET language (e.g., C#, VB.NET, etc) is. A special program called *AsmL Weaver* operates on the intermediate representation of the implementation and the specification and injects the specification code into the actual implementation. As a result, pre-/post-conditions and class invariants are automatically checked when the code is executed. In the case of model programs, the model program is executed in parallel with the real application and the application's state is continuously checked against the model state.

An approach similar to model programs in AsmL might be an interesting area of research in Web engineering. As outlined above, the sitemap provides the possible model states and variables and state transitions can be defined in terms of hyperlinks and request parameters. More information about AsmL and its advanced features is found in [7–9].

In practice, Web application testing often focuses on the most critical parts of the functionality. Time-to-market is important and sufficient time for detailed tests is rarely available. As a result, Web applications tend to be tested and improved after their initial deployment. This again puts special emphasis on the maintenance and evolution phases that are discussed below after the deployment of Web applications.

4.7 DEPLOYMENT PHASE

A common setup for a Web application development environment consists of three separate machines: a development machine, an internal test server, and the final production machine that offers the site to the outside world. Deploying a Web application thus has two steps. First the application is deployed from the development to the internal test server; after testing and final clearance it is then deployed to the production machine. Usually the test machine and the production machine have a similar setup which results in similar deployment mechanisms.

Ideally, deployment of a Web application would only require to copy the content, formatting and application logic files from the development machine to the target machine. More abstractly, we distinguish deployment processes that just copy files from the development to the target machine (deployment-by-copy) and processes that modify and adapt source files and require a new build of the system on the target machine (deployment-by-adaptation).

With XML-based publishing frameworks (e.g., Cocoon) deployment-by-copy is easily possible for XML content files and XSL stylesheets. Application logic usually exists in source code and needs to be compiled into some binary format or library that has to be made available on the target machine. In the case of database systems that are involved in almost all Web applications today, deployment means to perform some synchronization activity on the database (e.g., an updated database structure has to be published to the production machine or some internal tables have to be migrated).

In deployment-by-adaptation, a common deployment activity is to replace the settings of the test environment with those of the production environment. Typical examples are the database connect string, the document root or the installation directory of libraries. In XML-based Web development, document and parameter entities can be used to reduce this task to the modification of a global definition file. Alternatively, scripts applying regular expressions to all source files and triggering a new build process on the production machine are necessary.

In non-XML publishing environments, deployment-by-adaptation can also involve more complex tasks. In MyXML, for instance, all XML/XSL definitions are first translated into XHTML files and source code that can then be deployed to any servlet container and Web server. Generally, the deployment of a Web application is highly dependent on the implementation technology and publishing framework. Depending on the project and the technologies used more or less custom tailored processes are needed.

After the initial deployment of the Web application, it enters the maintenance and evolution phases. The next two sections discuss these important phases and relate them to XContracts and the XGuide development process.

4.8 MAINTENANCE AND EVOLUTION

Maintenance of software applications is a difficult task. It is well known in software engineering that the maintenance phase of software projects consumes about 70 percent of the whole project effort [104, 151]. It is also widely accepted that the earlier a bug is introduced in a system,

the more expensive it is to remove. This also motivates the emphasis of software architecture and design in most of today's software engineering methodologies. The claim is that a well-engineered architecture and a good design significantly reduce the number of (costly) bugs to be corrected in the later phases, especially the maintenance phase.

In [67], the authors distinguish *corrective*, *adaptive*, and *perfective* maintenance. Corrective maintenance covers corrections and bug fixes that were introduced in the implementation phase. Typical examples in the Web domain could be the correction of typos in the content or programming errors in the application logic. Adaptive maintenance is the adjustment of the system to the outside environment. The fast development cycles of software such as Web servers, Java servlet containers, databases and build tools from time to time requires an adaptation of the Web application to comply with new protocols, input/output behavior or configuration settings. Short product cycles are necessary to keep up with the continuously evolving technologies and standards on the Web and, even more importantly, to close security vulnerabilities in the respective products. Table 4.1 shows the incredibly fast development history of the Apache Xerces XML parser [141] and the Jakarta Tomcat [140] servlet container to give an impression of how short the product cycles are.

Finally, perfective maintenance denotes all extensions and improvements of the application. Perfective maintenance is of special interest in Web engineering. Software applications evolve in a step-wise manner introducing new versions of the product. Small changes and patches to existing software products are less common. Web applications are less rigid than software applications and often expected to be frequently or even continuously upgraded and extended. As a result, design-for-change is even more important in Web engineering than in software engineering. Today design-for-change in Web engineering is supported by separating implementation concerns. In XGuide, we further abstract this principle and introduce multi-concern contracts that support separation of concerns not only on the implementation but also on the conceptual level.

Since XContracts also act as specifications for Web pages and components, they also alleviate a second problem found in software maintenance: program understanding. In [55] the authors state that about 50 percent of the overall maintenance effort is program understanding. In the case of Web engineering that means to understand where the content is stored and how it is included in dynamic pages; what components are involved in the page generation process; where the layout is defined and how it is applied to the content before the final page is delivered. If no Web engineering concepts such as separation of concerns is used in the implementation of a Web site, content, layout and application logic definitions are frequently intermixed making it hard to understand what is going on in a complex Web application–this is especially true if the original implementation dates back a couple of months or was done by a different person. While separation of implementation concerns already helps to identify the definition of every single concern, separation of concerns on the conceptual level (in XContracts) provides the information to understand how the various concerns work together and depend on each other.

To better structure and control the maintenance phase in Web engineering, we introduce an orthogonal classification in addition to corrective, adaptive, and perfective tasks as introduced in [67]. Every activity in the maintenance phase is classified as being a maintenance or an evolution activity.

Table 4.1: The development cycles of the Apache Xerces XML parser and the Jakarta Tomcat servlet container.

Xerces version	**Release date**		**Tomcat version**	**Release date**
Xerces v1.0.0	November 9, 1999		Tomcat v4.0	September 17, 2001
Xerces v1.0.1	January 5, 2000		Tomcat v4.0.1	October 14, 2001
Xerces v1.0.4	May 9, 2000		Tomcat v4.0.2	February 10, 2002
Xerces v1.1.0	May 19, 2000		Tomcat v4.0.3	March 1, 2002
Xerces v1.1.1	June 5, 2000		Tomcat v4.0.4	June 13, 2002
Xerces v1.1.2	June 21, 2000		Tomcat v4.0.5	October 9, 2002
Xerces v1.1.3	July 26, 2000		Tomcat v4.1.0	April 26, 2002
Xerces v1.2.0	August 28, 2000		Tomcat v4.1.2	May 14, 2002
Xerces v1.2.1	October 19, 2000		Tomcat v4.1.3	May 29, 2002
Xerces v1.2.2	November 27, 2000		Tomcat v4.1.5	June 14, 2002
Xerces v1.2.3	December 6, 2000		Tomcat v4.1.6	June 28, 2002
Xerces v1.3.0	February 1, 2001		Tomcat v4.1.7	July 5, 2002
Xerces v1.3.1	March 16, 2001		Tomcat v4.1.8	July 23, 2002
Xerces v1.4.0	May 22, 2001		Tomcat v4.1.9	August 10, 2002
Xerces v1.4.1	June 22, 2001		Tomcat v4.1.10	August 30, 2002
Xerces v1.4.2	July 23, 2001		Tomcat v4.1.14	October 29, 2002
Xerces v1.4.3	August 20, 2001		Tomcat v4.1.14	November 14, 2002
Xerces v1.4.4	November 15, 2001		Tomcat v4.1.15	November 26, 2002
Xerces v2.0.0	January 29, 2002		Tomcat v4.1.16	November 26, 2002
Xerces v2.0.1	March 7, 2002		Tomcat v4.1.17	December 17, 2002
Xerces v2.0.2	June 21, 2002		Tomcat v4.1.18	December 19, 2002
Xerces v2.1.0	August 28, 2002		Tomcat v4.1.19	January 15, 2003
Xerces v2.2.0	September 26, 2002		Tomcat v4.1.20	February 12, 2003
Xerces v2.2.1	November 11, 2002		Tomcat v4.1.21	February 25, 2003

Maintenance tasks are intra-concern tasks, i.e., they only affect a single concern and do not require updates to the contract or other concerns. As a result, maintenance tasks are self-contained and can be performed at any time. Several maintenance tasks are common in Web engineering:

- **Content Updates.** The content of a Web application, especially a Web-based information system, changes frequently and content updates are the most frequent maintenance activity in Web engineering. Content updates affect the information provided by the system but do not influence other concerns such as the formatting rules or the application logic.
- **Layout Updates.** Changes to the graphical appearance of a Web application are not so frequent but still relatively common. Information needs to be arranged differently on a page, the color scheme of the Web application has to be adjusted to the corporate identity, or hover effects (e.g., when moving the cursor over a link) shall be introduced. These changes exclusively require modifications to the formatting rules but do not affect other concerns.
- **Bug Fixes.** Bug fixes in the application logic of a Web application are also typical examples of maintenance tasks. The program logic processing client requests is modified but the input and output interfaces to the client, the content and the formatting instructions are left unchanged.

Referring to the classification in [67], bug fixes are categorized as corrective maintenance. Layout updates react to external requirements (e.g., updates to the corporate identity) and can be seen as adaptive maintenance. Content updates evolve the Web site and count as perfective maintenance. Some updates correct existing errors (e.g., typos); others are due to timely changes of content (e.g., on a news portal).

Since XGuide features separation of concerns and maintenance tasks solely depend on one concern, they can be integrated into the Web application at any time. Thus content updates, layout updates and bug fixes can be done independently of each other or even concurrently. XGuide's view of maintenance is a great advantage over the prevalent Web development practice where concerns are not clearly separated but intermixed. A content update in such a system thus can easily cause unintended side-effects on the layout information or the application logic.

Evolution, on the other hand, denotes inter-concern updates. In this case, the contract of a page or component must be modified and multiple concerns are involved. Typical evolution scenarios are:

- **Extending Pages.** Imagine you have a Web page with a three column layout. So far, only the middle column is used to display content; the left and right columns are for layout purposes only. Now you want to place some special information in the left or right column of the page. Such an update involves the content and the layout concerns since both the information to be displayed (e.g., text and images) and its graphical appearance (e.g., font sizes, colors, alignment, etc.) must be defined.

- **Extending Forms.** A form extension is a special case of a page extension. An additional form field is added to an existing Web form. The update not only requires the content (e.g., the field labels, explanation text and default values) and the layout (e.g., to position the new fields correctly) for the new field to be defined but also involves the application logic that must process the value entered in the field (e.g., to store them in a database).

- **Adding Pages.** Adding new Web components or pages to an existing application, obviously also classifies as evolution. A whole new content structure, graphical appearance and potential application logic code need to be defined. In special cases, however, if the page does not require logic processing and can reuse existing formatting stylesheets, adding a new page can shrink to a maintenance task.

Larger evolution scenarios such as adding whole subsystems can be seen as mini-projects themselves starting with a requirements analysis and feasibility decision before the conceptual model and XContracts are updated. A good example for this case would be the Orange Juices, Inc. example. If the existing Web site is extended with a shopping cart functionality, the development and integration of the shopping cart defines the mini-project that is eventually integrated into the existing project.

Documentation and version control of all artifacts plays an important role during all phases of the development process but specifically so in the maintenance and evolution phase. Documentation further eases program and concern understanding which is crucial for any modifications to or updates of a system. Version control of concern implementations and contracts is also indispensable. Consider an update of a Web page that embeds a Web component. Consequently, the Web component's contract evolves together with the page. To not break other page definitions that might use the same component, it is important to uniquely identify which version of a component or component contract is referenced in a page. As a consequence, all artifacts used in an XGuide development process (i.e., XContracts, XPages, content concerns, formatting stylesheets, application logic, etc.) are version controlled and all references or composition dependencies use fully qualified references, i.e., the unique identifier of the artifact together with the desired version.

Although the development process ends with the maintenance and evolution phase, it cannot be overemphasized that a Web site or application is much more dynamic than a traditional software product and requires an ongoing maintenance effort. If a Web site is well-designed, mostly maintenance tasks such as content updates and bug fixes should be necessary after its initial deployment. Evolution tasks should be significantly more scarce. Large-grained evolution activities of the size of small mini-projects should be even more rare than other evolution activities.

Generally, we can determine the state of a Web application at any given time using the classification shown in Table 4.2. If no changes or extensions are made to a Web application, all artifacts on all levels (i.e., the XGuide sitemap as conceptual model, the XContracts, and the actual implementation) are stable. During a maintenance activity, the implementation changes, but the contracts and the conceptual model remains fixed. Evolution scenarios also involve the contracts thus only keeping the conceptual model unchanged. Extensions of the size of small

projects (e.g., adding a shopping cart or another new service to an existing site) require updates of all artifacts including the conceptual model.

Table 4.2: A classification to determine the state of a Web application with regard to updates and extensions.

	Conceptual Model	**Contracts**	**Implementation**
No changes	stable	stable	stable
Maintenance	stable	stable	changing
Evolution	stable	changing	changing
Mini-Project	changing	changing	changing

As in all model-driven approaches, it is important to keep the model information up-to-date. In XGuide, the actual model is represented by the conceptual model (i.e., XGuide sitemap) itself and the XContracts of the pages and components. Thus for any maintenance or evolution activity, first the corresponding model artifact has to be identified and updated. Only then can the implementation be modified according to the changes in the higher-level artifacts. If this rule is ignored, not only is the traceability of the specification information lost but the whole model cannot be used any more. As a result no explicit design information would exist and the implementation would be the only source of information. While such modifications might work in the short term, they are strongly discouraged in terms of maintenance and traceability and can only be compared with the (in)famous statement on documentation in software engineering projects: *"The code is the documentation"*.

Keeping the conceptual model and the contracts consistent with the implementation, on the other hand, ensures—among the advantages discussed above—that side-effects of maintenance tasks are easily detectable, the workflow of the application is always clear, no hidden dependencies among components or concerns exists, and new team members can easily understand the inner functioning of the system.

4.9 Conclusion

This chapter introduced the XGuide methodology and its seven phases: requirements analysis, feasibility decision, conceptual modeling and design, implementation, testing, deployment and maintenance/evolution. First, the requirements diagram is compiled together with the customer. A simple notation for modeling a high-level view of the Web application is used to support the communication among the various roles involved in a Web project; additional requirements

are recorded on separate requirements cards. Lacking established metrics for Web projects, the feasibility decision uses informal decision guidelines to estimate the effort and time frame of the project. If the project is approved, the design phase distinguishes between design in-the-large and design in-the-small. Design in-the-large refines and componentizes the requirements diagram and introduces application logic processes to model workflow and data processing. It also introduces the input and output interfaces of pages and components. Design in-the-small uses the refined conceptual model to define XContracts for pages and components that act as specifications for the implementation. The implementation phase then transforms page and component specifications into XPages, implementations in a concrete technology. Exploiting the strict separation of concerns on the conceptual level, the implementation of all concerns can be done concurrently. The testing phase strongly depends on the requirements of the project. Testing of Web applications involves various testing strategies–often only the correct functioning of the application logic is thoroughly tested. More sophisticated approaches involve XML-based regression testing to ensure that changes do not break other components and model-based testing which can cover more complex scenarios involving several subsequent requests. After the initial, technology-specific deployment, the project enters the maintenance and evolution phase. XGuide classifies maintenance as intra-concern updates such as content or layout modifications. Maintenance tasks do not depend on other concerns and can be integrated easily whenever necessary. In contrast, evolution tasks involve multiple concerns (inter-concern) and range in their complexity from simple page extensions to small Web projects on their own if whole subsystems are added.

Although the importance and central role of contracts became obvious in the discussion of XGuide, we postponed a detailed discussion of XContracts. So far, we only mentioned that contracts must be composable to support embedding of components into pages and extensible to support adding of new concerns. The next chapter catches up on how contracts are specified and composed, what contract concerns are, and how contracts are extensible. It also presents a formal model for contracts that defines the semantics of and operations on contracts and contract concerns.

CHAPTER 5

CONTRACTS AND CONTRACT COMPOSITION

New ideas pass through three periods:
- It can't be done.
- It probably can be done, but it's not worth doing.
- I knew it was a good idea all along!

Arthur C. Clarke

Contracts and contract concerns are the cornerstones of the XGuide methodology. After the high-level overview of their role in XGuide, this chapter presents a detailed discussion of the syntax and semantics of contracts. It also introduces a formal model of contracts, describes the realization of this model as XML contracts, and explains how contract composition is defined.

As explained in the previous chapter, contracts reside between the high-level conceptual model and the concrete implementation of a Web application. Thus they effectively bridge the gap between the big picture in the conceptual model and its implementation in a concrete technology. To start the discussion of contracts, we first introduce a meta-model for Web applications and how contracts are integrated in the model.

5.1 A META-MODEL FOR WEB APPLICATIONS

A meta-model introduces the concepts and terminology used in modeling a given domain. In the context of Web engineering, often an implicit meta-model defining (Web) pages and navigation links is assumed. Similarly, we implicitly used diagram artifacts such as pages, components

or links in the requirements diagram and component web of the XGuide method. This section introduces two existing meta-models used in the Web engineering domain: the Dexter Hypertext Model [75] and Conallen's UML-based approach [37]. It then makes the meta-model used for modeling Web applications in XGuide explicit and relates it to the Dexter model and Conallen's approach.

5.1.1 The Dexter Hypertext Reference Model

The Dexter Hypertext Reference Model [75] is the outcome of two small workshops on hypertext in 1988. The workshop was held in the Dexter Inn in New Hampshire–hence the name for the model. The researchers strived to answer the question what hypertext systems have in common and how they can be classified. Another goal of the workshops was to come up with a terminology for the hypertext field.

The most generic hypertext model consists of nodes and hyperlinks. The nodes represent actual information or content and the hyperlinks provide the possibility to connect two pieces of information with each other. Because of the wide range of uses and the generic nature of the term "node", the Dexter model uses the more neutral term "component" instead. From today's perspective where the term "component" is heavily overloaded, it is questionable whether this change really made the model clearer and easier to understand.

The Dexter model divides a hypertext system into three layers: the *run-time* layer, the *storage* layer and the *within-component* layer. The model strongly focuses on the storage layer which defines the information nodes (components) and hyperlinks of the model. This node/link network structure is the essence of a hypertext system. The within-component layer is concerned with the internal structure and workings of the components themselves. Because of the great variety of possible information nodes and content types, the Dexter model purposefully does not elaborate on the within-component layer. The run-time layer describes how the hypertext network of information components and links is represented and displayed at runtime. The main tasks of the run-time layer is the creation, presentation and unpresentation (destruction) of components. Further run-time operations include *follow-link* or session management functions. On the World-Wide Web, the run-time layer is usually represented by a Web server combined with some sort of container for the business logic (e.g., a servlet container, an application server, etc.).

The focus of the Dexter model, the storage layer, is centered around the notion of an (information) component. A component in the Dexter model is either an atomic or composed information component or a link. An atomic information component only contains the actual information, a composite contains additional references to other components. Support for composites is one of the key features that supports reuse and consistency of information. Secondly, the Dexter model defines a component to consist of a so-called *base component* representing the content and additional component information. This additional information includes all properties of the component other than the content: its graphical representation, linking information, etc. Although not designed as explicit goal, Dexter components feature some support for separation of concerns since the actual information is separated from the other properties of the component. Together with the composability of components, the extensibility of base components with

additional properties result in a flexible and powerful model for information systems.

The missing component for a hypertext system is the linking of information components. The Dexter model not only supports links between information components but also addressing locations or items within components. In the model, this mechanism is called *anchoring* which today is commonly seen on the World-Wide Web where anchors are the only possibility to address locations within an HTML document. To unambiguously identify components, every Dexter component has a unique identifier. The target of a hyperlink thus consists of such a unique component identifier in combination with an anchor specification that locates the referenced position within the component. On the Web, uniform resource locators (URLs) [21] contain both the component (i.e., resource) identifier and the within-component fragment identifier based on existing anchors. With the recent W3C's XLink/XPointer recommendation [43, 44], the need for embedding anchors in a Web resource to be able to reference a specific position in a component vanishes. XLinks uniquely identify the target resource while an XPointer identifier locates a given position in the content. Unlike the anchoring approach, no embedded anchors are required in the content which is especially helpful for referencing third-party documents for which the required write access to embed anchors usually is not granted.

5.1.2 Modeling Web Application Architectures with UML

In [37], Conallen presents a UML-based meta-model for Web applications that takes a completely different approach than the Dexter model. Conallen starts with the definition of a Web application as a software system with a business state whose front end is delivered via a Web system. This definition completely coincides with our definition in Section 2.1. Regarding a Web application as a complex software system and given the dominant role of the UML for modeling software systems, Conallen exploits the UML extensibility mechanisms and defines stereotypes to model Web applications using UML diagrams.

Unlike the Dexter model that is content-centric (i.e. the storage layer), the extended UML notation is focused on the business logic of a Web application. Conallen concludes that modeling Web applications as UML class diagrams alone, is not sufficient. The additional semantics and inner workings of Web pages are thus represented by stereotyped UML classes. The resulting component diagram is similar to a sitemap and conceptually close to the XGuide requirements diagram. To model the collaboration inside Web pages (e.g., client-side or server-side scripting), the functional entities used in a Web page are separated. Again the principle of separation of concerns is used. In this case, however, to separate various logic concerns from each other rather than to separate different kinds of concerns such as the content, the logic, access control or graphical presentation.

In Conallen's model, each component has a client-side and a server-side representation. This is necessary to model the different behavior of a component depending on the location where it is displayed or executed. The client-side view of a component, for instance, includes the JavaScript functionality and the processing of potential Web forms. The server-side view, on the other hand, represents the back-end application logic that is necessary to create the page. The *client page* and

server page stereotype classes are used in the meta model to represent the two different views of the same component.

Hyperlinks, the second ingredients for a hypertext model, are depicted by stereotyped association classes. Such relationships can have quantifiers (e.g., 0..*) to model links to multiple pages. A typical example is the result of a search request in a product catalogue. The result page contains a number of links that point to the respective product details page–each of which is an instance of a stereotyped class parameterized by the product identifier. The notion of quantifiable hyperlink associations is similar to the multi page approach in XGuide. Multi pages similarly represent a collection of typed and parameterized pages (e.g., the set of product detail pages) that can be referenced from another page or component (e.g., the search result page).

A major drawback of the model presented in [37] is that no real notion of a component exists. Pages are the basic entities and only the functional aspects of pages are further refined in an object-oriented approach. Apart from the business logic of a Web application, no componentization (e.g., for page fragments that are composed to form the final Web page or for layout reuse) is supported.

5.1.3 The XGuide Meta-Model for Web Applications

Implicitly we already introduced the entities of the XGuide meta-model in the discussion of the requirements analysis and design phases. Unlike the above approaches, XGuide implements the notion of a Web component as a first-class primitive. It does not focus specifically on the content (as in the Dexter model) or on the business logic (as in the Conallen approach). Instead, XGuide components consist of several equal and independent concerns that represent the content, the layout or the application logic.

Components in XGuide come in two flavors: either as composable or as non-composable entities. A composable component represents a fragment of a Web page that is reused and embedded in other components (e.g., the footer of a page). Non-composable components are the Web pages themselves that have the same characteristics as any other (composable) component but cannot be embedded into other components or pages themselves. The XGuide meta-model distinguishes between composable components, simple pages and multi pages that represent a parameterized collection of simple pages. Dependencies on third-party or external services are covered by external components. A composition dependency describes the embedding of a component into another component or page.

Linking information in the XGuide model is defined by navigational dependencies. Such dependencies represent a hyperlink from one page or component to another. Since XGuide introduces the notion of a multi page, quantifiers on navigational dependencies as found in Conallen's approach are not necessary. Instead, hyperlinks to multi pages transport context information to select the appropriate page to display out of the collection of pages represented by the multi page. Also note that composable components (page fragments) as well as full pages can act as sources for hyperlinks. In contrast, only Web pages but no page fragments are allowed as link destinations. Using page fragments as link destinations would result in the definition of one-to-many hyperlinks since the source of such a link is implicitly linked to all pages that embed the

referenced component. One-to-many hyperlinks are currently not supported in XGuide because they are extremely rare in Web applications. With the (currently missing) broad support of the XLink/XPointer recommendation [43, 44], however, one-to-many links will be easy to integrate into XGuide.

Apart from the classical elements of a hypertext model (i.e., the information components and the linking structure), the XGuide meta-model introduces additional artifacts to loosen the coupling among components and provide support for the increasing use of dynamically generated Web pages.

The most important such concept is the notion of a *contract* for a Web component. Contracts provide specifications of Web components and pages and clear interfaces among the component's internal concerns. Contracts are discussed in detail later in this chapter.

A second extension to traditional hypertext models is the explicit modeling of interfaces and information processing using *input*- and *output-interfaces*. Input interfaces state the information requirements of a component in order to be instantiated. As a consequence, only dynamic pages have input interfaces. Output interface, on the other hand, specify the information a page provides via Web forms–both static and dynamic pages can have output interfaces. If input- and output-interfaces of linked pages do not directly match, a mediator (processing) component has to be inserted between them. Such a processing entity is traditionally called the business or application logic of the Web application. In the XGuide model, application logic processes can be introduced to make the workflow, the dependencies on external systems, and processing of information explicit. Such processes, as all other model artifacts, have a specification that states their input- and output-interfaces to ensure easy and consistent deployment.

Table 5.1 presents a description of all (visual and non-visual) entities of the XGuide meta-model for Web applications.

Regarding the level of detail of the XGuide meta-model, it is positioned between the Dexter and the Conallen models. In the Dexter model, information on the internals of a component is explicitly excluded. The authors state that the internals of such components can be so manifold that their modeling is simply beyond the scope of a generic model. In Conallen's approach, on the other hand, only the application logic is modeled–but in great detail. The XGuide approach lies in between in that it provides a component specification that defines separate concerns and interfaces to other components but otherwise agrees with the authors of the Dexter model that the internals of each concern depend too much on the concrete application to be covered by a generic model. Modeling the content concern, for instance, might involve information mining and structuring techniques; the graphical appearance is often determined using storyboards, UI charts or rapid prototyping; the application logic can be modeled using the standard or Conallen's extended version of the UML.

Compared to the other models presented above, the XGuide model covers a broader spectrum of the design space since it does not focus on a single concern but provides an extensible contract model that supports an arbitrary number of equal concerns. Although Conallen defines a client-side and a server-side view of the Web components, a real component approach that features reusable and composable Web components–as found in the Dexter (composites) and XGuide approaches–is missing. Unlike the Dexter model, we do not explicitly support operations on

Table 5.1: The artifacts of the XGuide Meta-Model for Web Applications.

Name	**Description**
Simple Page	A simple page represents a single Web page that consists of several concerns and can embed other components. Simple pages themselves are components but cannot be further composed.
Multi page	Multi pages are parameterized collections of (simple) pages of the same type. Such pages share the same structure and graphical appearance and only differ in the content they present.
Component	Components are composable entities and represent page fragments to be reused and embedded into other components and pages.
External Page	External pages denote the dependencies to external, third-party services that are not part of the project but need to be integrated.
App logic process	Application logic processes make the processing of user input explicit and act as mediators between the input- and output interfaces of pages and components. Application logic processes often are complex software systems on their own with multi-layer architectures, back-end databases and complex interactions with external services.
Navigational Dependency	A navigational dependency represents hyperlink from the source page (or component) to the destination page. In the case of multi page targets, navigational dependencies transport context information to provide the multi page's input parameters.
Compositional Dependency	Compositional dependencies specify the embedding of components into other components and pages.
Input/output interface	Input- and output interfaces are associated with all sorts of pages and components to capture the information they provide (via Web forms; output interface) and the arguments they require to be instantiated (input interface). The explicit specification of input and output requirements supports consistency checking of component webs and provides a clear interface to programmers.
Contract	Contracts are specifications for pages and components and provide information about their internal structure, the concerns they implement, their navigational dependencies, and their compositional dependencies. Contracts are not directly part of the meta-model but are used to capture and later on refine meta-model information.

Web components and pages. The Dexter model defines both storage (e.g., create component, destroy component, etc.) and run-time operations (presentComponent, followLink, etc.). In XGuide, the life-cycle of a component is implicit, i.e., it is automatically created when it is first requested, automatically displayed as a response to the request, and following links is simply modeled as requesting a new component.

In XGuide, all information on model artifacts such as pages, hyperlinks or application logic processes is captured in contracts. Hence contracts are the actual presentation of the model artifacts and act as specifications for a subsequent implementation. The remainder of this chapter discusses the idea of contracts for Web components and pages. We first introduce a formal model for contracts and contract concerns. The following sections present the actual implementation of contracts in XGuide as XML documents and the semantics of and operations for contract composition.

5.2 A Formal Model for Web Component Contracts

The idea of contracts originates from the domain of software engineering. On the component level, interfaces represent the contract of a component, i.e., specify how the component can be used from the outside. Bertrand Meyer's design-by-contract [83, 111, 112] brings the idea of contracts to the next level. Not only the interface but also pre-/post-conditions and invariants specify the expected behavior of methods. Eiffel [110, 113] provides built-in support for such contracts. The next level of evolution is reached with state machines such as AsmL (Abstract State Machine Language) [6–8]. It supports the definition of contracts for Microsoft's .NET platform. Such contracts of pre- and post-condition and type invariants can then be woven into the actual implementation code to have them checked at runtime.

The idea of a contract in Web engineering—just as the the idea behind interfaces in software engineering—is to reveal all the necessary information to interact with an artifact independently of its inner workings. Similarly, an *XGuide contract (XContract)* describes the properties of a set of pages with the same characteristics. Such a contract could, for instance, specify the structure of the content on a page. It does not, however, say how the content is created or collected.

The structure of the content to be placed on a page is one example. Similarly, contracts define specifications for the development of the graphical appearance and the application logic. The various properties of a page or component that are covered by its contract are called *contract concerns*. In addition to defining a default set of contract concerns, XContracts provide an extensibility mechanism to add and integrate new contract concerns without affecting existing ones.

Throughout this work, we developed an intuitive notion and understanding of what contracts are and how they can be used. This section presents the formal foundation for dealing with contracts in XGuide. It defines what a contract and a concern is and how contract composition works.

An XGuide contract $\mathbf{C}$ is an unordered tuple of concerns c_i that denote the different characteristics of the page.

$$\mathbf{C} = (c_1, c_2, ...c_n) \tag{5.1}$$

The structure of the content mentioned above, the input and/or output interfaces or access control constraints are examples for concerns that are aggregated in a contract.

Furthermore, each concern has an associated type which is used to identify the concern in the contract. A sample contract, thus, could contain a concern of type *structure*, a concern of type *interface* and another concern of type *access control*. The type of a concern can be retrieved using the $type$ function, i.e., the type of c_i is $type(c_i)$. A contract can contain at most one concern of any given type. As a result, the contract definition given in 5.1 has to be extended to

$$\mathbf{C} = (c_1, c_2, \ldots, c_n) \text{ where } type(c_i) \neq type(c_j) \; \forall \; c_i, c_j \in \mathbf{C}, i \neq j \tag{5.2}$$

If the type of a concern $t_i = type(c_i)$ is independent of another type $t_j = type(c_j)$, we call t_i *orthogonal to* t_j. Independent in this context means that concerns of type t_i do not need information specified in a concerns of type t_j to express their characteristics. If t_i is orthogonal to t_j we write

$$t_i \quad \perp \quad t_j \tag{5.3}$$

If type t_i of a concern is independent of *all* other types of concerns in a contract, we call this type of concern *orthogonal to the contract* or simply *orthogonal*.

$$t_i \quad \perp \quad \mathbf{C} \tag{5.4}$$

Since at most one concern of any given type may be present in a contract, we can also call a concern orthogonal if its type is orthogonal. Thus, a concern c_i is orthogonal to concern c_j if the type of c_i is orthogonal to the type of c_j.

$$c_i \quad \perp \quad c_j \quad \Longleftrightarrow \quad t_i \quad \perp \quad t_j \tag{5.5}$$

If the type of concern c_i is orthogonal to the contract, we also call c_i orthogonal to the contract.

$$c_i \quad \perp \quad \mathbf{C} \tag{5.6}$$

Finally, a contract is said to be orthogonal if all the concerns it contains are orthogonal.

The *structure concern* describing the structure of the content of a page is a good example for a concern which is orthogonal to the contract. It does not depend on any other concern since the structure of the content is self-contained and does not require further information from other concerns.

Although concerns are independent of each other in the domain of the final application, their definition might require dependencies among concerns. The *access control concern*, for instance, defines which parts of the content are accessible to a given principal. Clearly, the definition of the

access control concern must depend on the structure concern to be able to refer to the appropriate parts of the content. As a consequence, we call the type t_j of concern c_j *dependent* on t_i if concerns of type t_j rely on concerns of type t_i.

$$t_i \quad \leftarrow \quad t_j \tag{5.7}$$

Arguing as before that the type t_i of concern c_i has to be unique within the contract, we call a concern c_j which relies on another concern c_i *dependent on* c_i denoted as shown in 5.8.

$$c_i \quad \leftarrow \quad c_j \tag{5.8}$$

For XGuide contracts we further enforce some rules on the use of dependency relations to restrict the complexity of contracts. The optimum is to have an orthogonal contract since in this case no concern is dependent on any other. If this is not possible, the number and complexity of dependency relationships should be as small as possible. We thus do not support a concern c_j to be dependent on more than one concern c_i. Multiple concerns $c_j, c_{j+1}, c_{j+2}, \ldots$, however, can be dependent on the same concern c_i.

$$\{c_i, c_{i+1}, c_{i+2}, \ldots\} \quad \leftarrow \quad c_j \qquad \textbf{illegal!} \tag{5.9}$$

$$c_i \quad \leftarrow \quad \{c_j, c_{j+1}, c_{j+2}, \ldots\} \tag{5.10}$$

We allow transitive dependencies of concerns, i.e., c_j is *transitive dependent* on c_i if there are other concerns c_k in $\mathbf{C}$ to form a *dependency chain* from c_i to c_j.

$$c_i \Leftarrow c_j \iff \exists\, c_i \leftarrow c_{i+1} \leftarrow c_{i+2} \ldots \leftarrow c_j \tag{5.11}$$

The length of the longest dependency chain l of a concern a_i can unambiguously be defined as

$$l(a_i) = \mid \{a_0 \leftarrow a_1 \leftarrow \ldots \leftarrow a_i \mid a_0 \text{ is an orthogonal concern}\} \mid \tag{5.12}$$

Additionally, we define $\mathcal{L}_g$ as the set of all concerns with a dependency chain of length g and $l_{max}(\mathbf{A})$ as the length of the longest dependency chain in $\mathbf{A}$.

$$\mathcal{L}_g(\mathbf{A}) \quad = \quad \{a_i \mid l(a_i) = g\} \tag{5.13}$$

$$l_{max}(\mathbf{A}) \quad = \quad max(l(a_i)) \qquad \forall a_i \in \mathbf{A} \tag{5.14}$$

Based on the above definitions, an orthogonal concern a_o has $l(a_o) = 0$ and $\mathcal{L}_0(\mathcal{A})$ is the set of all orthogonal concerns in $\mathbf{A}$.

Extensibility of contracts can now be defined as the addition of a new concern c_n and its corresponding dependency relationships taking into account the above restriction.

For convenience, we also introduce compatibility among concerns. We call a concern c_i *(type) compatible* to c_j if both concerns have the same type. Correspondingly, two concerns are called *(type) incompatible* if they have different types.

$$c_i \text{ compatible } c_j \iff type(c_i) = type(c_j) \tag{5.15}$$

$$c_i \text{ incompatible } c_j \iff type(c_i) \neq type(c_j) \tag{5.16}$$

5.2.1 Contract Composition

Using the definitions of contracts and concerns from the previous section, we now discuss the composition of contracts. Let $\mathbf{A} = (a_1, a_2, \ldots, a_n)$ and $\mathbf{B} = (b_1, b_2, \ldots, b_m)$ be XGuide contracts. We define a composition operator $\bigoplus$ and $\mathbf{C} = (c_1, c_2, \ldots, c_p)$ as the result of the composition:

$$\mathbf{C} \quad = \mathbf{A} \quad \bigoplus \quad \mathbf{B} \tag{5.17}$$

The basic idea of the $\bigoplus$ composition operator is to compose all compatible concerns of the contracts taking all dependency relationships into account. The functionality of $\bigoplus$ is thus defined in terms of $\bigoplus_a, \bigoplus_b, \ldots$—the composition operators for the concerns of types a, b, ... included in $\mathbf{A}$ and $\mathbf{B}$.

The number of concerns p in the composition result is determined by the number of unique types of concerns in $\mathbf{A}$ and $\mathbf{B}$. The following simple condition always holds for p:

$$p \quad \geq \quad max(m, n) \tag{5.18}$$

For the discussion of contract composition we also introduce $\mathcal{T}$—the set of types occurring in the result $\mathbf{C}$ of the composition of $\mathbf{A}$ and $\mathbf{B}$. Since no new types are added to nor existing types are removed from the composition result by doing contract composition, $\mathcal{T}$ is the set of all types which occur in $\mathbf{A}$ or $\mathbf{B}$. Thus for $\mathbf{C} = \mathbf{A} \bigoplus \mathbf{B}$, we define

$$\mathcal{T}(\mathbf{C}) \quad = \quad \{type(c_i) \mid i = 1, \ldots, p\} \tag{5.19}$$

$$\mathcal{T}(\mathbf{C}) \quad = \quad \{type(a_i) \mid i = 1, \ldots, n\} \cup \{type(b_i) \mid i = 1, \ldots, m\} \tag{5.20}$$

Finally, we define $\mathbf{A}|q$ as the concern in $\mathbf{A}$ with type q. For any given contract $\mathbf{A}$, $\mathbf{A}|q$ is either the empty set (if no concern of this type is present) or contains exactly one element (since only one concern of any given type can exist in a given contract).

$$\mathbf{A}|q = \{a_i \mid type(a_i) = q\} \tag{5.21}$$

$$\mid \mathbf{A}|q \mid = 0 \quad \vee \quad \mid \mathbf{A}|q \mid = 1 \tag{5.22}$$

Contract Composability. We call two contracts $\mathbf{A}$ and $\mathbf{B}$ composable iff the dependencies in the two contracts are unique with respect to the rules for dependencies as described above. Specifically, this means that if a type l depends on a type k in $\mathbf{A}$, then l must (if present) also depend on k in $\mathbf{B}$, i.e.,

$$A|k \leftarrow A|l \Longleftrightarrow B|k \leftarrow B|l \tag{5.23}$$

5.2.1.1 Composition of Orthogonal Contracts

To begin with, we discuss the composition of orthogonal contracts, i.e., contracts which exclusively contain concerns with no dependencies. Let again $\mathbf{A} = (a_1, a_2, \ldots, a_n)$ and $\mathbf{B} = (b_1, b_2, \ldots, b_m)$ be contracts with an arbitrary number of orthogonal concerns. Dealing only with orthogonal concerns, we get a set of composed, orthogonal concerns $(c_1, c_2, \ldots, c_p)$ where p is the number of orthogonal types in $\mathbf{A} \cup \mathbf{B}$. Since no dependencies exist, $L_0(\mathbf{A}) = \mathbf{A}$ and $L_0(\mathbf{B}) = \mathbf{B}$. The composition $\mathbf{C}$ of $\mathbf{A}$ and $\mathbf{B}$ is defined as follows:

$$\mathbf{C} = (c_1, c_2, \ldots, c_p) = \mathbf{A} \bigoplus \mathbf{B} = \mathcal{L}_0(\mathbf{A}) \bigoplus \mathcal{L}_0(\mathbf{B}) \tag{5.24}$$

For each $c_k \in \mathbf{C}$ we distinguish three cases: the type of c_k only exists in $\mathbf{A}$, the type of c_k only exists in $\mathbf{B}$, and the type of c_k exists in both $\mathbf{A}$ and $\mathbf{B}$. Thus we define a $c_k \in \mathbf{C}$ as:

$$\begin{array}{rcll} c_k & = & A|k & \nexists\, b_j : type(b_j) = k \text{ (i.e., } \mathbf{B}|k = \{\}) \\ c_k & = & B|k & \nexists\, a_i : type(a_i) = k \text{ (i.e., } \mathbf{A}|k = \{\}) \\ c_k & = & A|k \bigoplus_k B|k & \exists\, a_i, b_j : a_i \text{ compatible } b_j \\ & & & \text{(i.e., } \mathbf{A}|k \neq \{\} \ \wedge\ \mathbf{B}|k \neq \{\}\text{)} \end{array} \tag{5.25}$$

In other words, we simply add all orthogonal concerns to the resulting composite contract that do not have a counterpart of the same type in the other contract. For those concerns which occur in both contracts, we apply the composition operator for the respective type.

Example. Given the orthogonal contracts $\mathbf{A} = (a_1, a_2)$ and $\mathbf{B} = (b_1, b_2)$ with $t_1 = type(a_1) = type(b_2)$, we would aggregate $\mathbf{C} = (c_1, c_2, c_3)$ according to the above rules as $c_1 = a_1 \bigoplus_{t_1} b_2$, $c_2 = a_2$, and $c_3 = b_1$.

5.2.1.2 Composition of Dependent Contracts

In the case of general dependent contracts, the situation is not as obvious as in the orthogonal case. We also have to consider dependencies among concerns including transitive dependency chains.

in $\mathbf{A}$ and $\mathbf{B}$ with the length of the longest dependency chain of the concern, i.e., we assign each concern to its corresponding $\mathcal{L}_i$ where i denotes the length of the dependency chain. Recall that every concern can only have a single dependency on another concern (similar to single vs.

multiple inheritance in object-oriented programming). As a result we get labels $l(a_i)$ on all nodes a_i of the directed acyclic graph of dependency relations between concerns. We can further divide the graph into levels where each level consists of all nodes with the same label, i.e., level i contains all nodes of $\mathcal{L}_i$.

Assume for the discussion of composition of dependent contracts that $\mathbf{A} = (a_1, a_2, \ldots, a_n)$ and $\mathbf{B} = (b_1, b_2, \ldots, b_m)$ are contracts with an arbitrary number of orthogonal and dependent concerns. As a result of the composition of $\mathbf{A}$ and $\mathbf{B}$, we get a composite contract $\mathbf{C} = (c_1, c_2, \ldots, c_p)$ where p is the number of *all* types in $\mathbf{A} \cup \mathbf{B}$, i.e., $\mid \mathcal{T}(\mathbf{C}) \mid$.

We now define by complete induction how contract composition is done. We first show how the composition works for concerns in $\mathcal{L}_0$ and then describe how composition is done for $\mathcal{L}_i$ under the assumption that all concerns of $\mathcal{L}_j$ with $j < i$ were already composed.

For $\mathcal{L}_0(\mathbf{A})$ and $\mathcal{L}_0(\mathbf{B})$ we deal with orthogonal concerns only and thus can apply the rules from the previous section. We calculate $\mathcal{L}_0(\mathbf{C})$ as $\mathcal{L}_0(\mathbf{A}) \bigoplus \mathcal{L}_0(\mathbf{B})$.

For the concerns in $\mathcal{L}_i$ on the other hand, the concerns in $\mathcal{L}_{i-1}$ influence how the composition is performed. Basically, the composition operator for a concern c_b of type b ($\bigoplus_b$) which depends on another concern c_a of type a has to 'understand' what the composition operator for type a ($\bigoplus_a$) means. This requires a way to apply $\bigoplus_a$ to $\bigoplus_b$ to reformulate $\bigoplus_b$ according to the definitions in $\bigoplus_a$.

We define the *reformulated composition operator* $\bigodot_i$ for a concern of type i that depends on type $i-1$ as the result of applying $\bigodot_{i-1}$ to $\bigoplus_i$:

$$\bigodot_i = \bigodot_{i-1} \bigoplus_i = \bigodot_{i-2} \bigoplus_{i-1} \bigoplus_i = \ldots = \bigoplus_0 \ldots \bigoplus_{i-1} \bigoplus_i \tag{5.26}$$

Remark. Note that for orthogonal concerns of type i $\bigoplus_i = \bigodot_i$. For all other concerns, however, this is usually not the case.

Returning to the composition of $\mathcal{L}_i(\mathbf{A} \cup \mathbf{B})$, we define the composition $c_k \in \mathcal{L}_i(\mathbf{C})$ as follows: be $c_m = \mathbf{A}|m \bigodot_m \mathbf{B}|m \in \mathcal{L}_{i-1}(\mathbf{C})$ and $c_m \leftarrow c_k$. As in the orthogonal case, we have to distinguish three cases: the $type(c_k) = t_k$ exists only in $\mathbf{A}$, t_k exists only in $\mathbf{B}$, and t_k exists in both $\mathbf{A}$ and $\mathbf{B}$. We can thus define a $c_k \in \mathcal{L}_i(\mathbf{C})$ as:

$$\begin{aligned} c_k &= A|k & c_m \equiv \mathbf{A}|m \wedge \mathbf{B}|k = \{\} \\ c_k &= \bigodot_m A|k & c_m \not\equiv \mathbf{A}|m \wedge \mathbf{B}|k = \{\} \end{aligned} \tag{5.27}$$

$$\begin{aligned} c_k &= B|k & c_m \equiv \mathbf{B}|m \wedge \mathbf{A}|k = \{\} \\ c_k &= \bigodot_m B|k & c_m \not\equiv \mathbf{B}|m \wedge \mathbf{A}|k = \{\} \end{aligned} \tag{5.28}$$

$$c_k = A|k \bigodot_k B|k \qquad \mathbf{A}|k \neq \{\} \wedge \mathbf{B}|k \neq \{\} \tag{5.29}$$

This means that if there is neither a concern of type k in $\mathbf{B}$ nor a concern of type m in $\mathbf{B}$, we can simply take what is defined in $\mathbf{A}$. If no concern of type k is defined in $\mathbf{B}$ but a concern

of type m is defined in $\mathbf{B}$, we apply the reformulated composition operator of type m ($\bigodot_m$) to the definitions of type k in $\mathbf{A}$ ($\mathbf{A}|k$). The same applies vice verse for $\mathbf{B}$. If a concern of type k is present in both $\mathbf{A}$ and $\mathbf{B}$, we use the reformulated composition operator for type m ($\bigodot_m$) on $\bigoplus_k$ to obtain the reformulated composition operator for type k ($\bigodot_k$).

Remark. Note that we can reformulate both composition operators as well as concerns by applying composition operators of types they depend on as unary operators. If we use a composition operator as a binary operator, we actually compose two concerns of the same type to form a composite concern.

Example. Extending the previous example on orthogonal contracts, assume contracts $\mathbf{A} = (a_1, a_2)$ and $\mathbf{B} = (b_1, b_2)$ with $t_1 = type(a_1) = type(b_2)$, $a_1 \leftarrow a_2$, and $b_1 \leftarrow b_2$.

The contracts in this example cannot be composed because they are not compatible, i.e., a_1 and b_2 have the same type but do not have the same dependencies. Specifically, a_1 is orthogonal while b_2 depends on b_1.

Example. A slight modification of the previous example makes the contracts composable. Assume contracts $\mathbf{A} = (a_1, a_2)$ and $\mathbf{B} = (b_1, b_2)$ with $t_1 = type(a_1) = type(b_2)$, $t_2 = type(a_2)$, $t_3 = type(b_1)$, $a_1 \leftarrow a_2$, and $b_2 \leftarrow b_1$.

In this case, both a_1 and b_2 are orthogonal and the types of the concerns that depend on them are different which does not violate the composability condition.

Following the rules of contract composition as described above, we create and label the dependency graph and divide it into the following disjunct levels: $\mathcal{L}_0(\mathbf{A} \cup \mathbf{B}) = \{a_1, b_2\}$ and $\mathcal{L}_1(\mathbf{A} \cup \mathbf{B}) = \{a_2, b_1\}$. Starting with the orthogonal concerns, we would calculate $\mathcal{L}_0(\mathbf{C})$ as $c_1 = a_1 \bigoplus_{t_1} b_2$. For a_2 we have a dependency on a_1 but we do not have a concern of type t_2 in $\mathbf{B}$. Thus we apply the reformulated composition operator $\bigoplus_{t_1} = \bigodot_{t_1}$ to a_2: $c_2 = \bigodot_{t_1} a_2$. The same steps have to be done for b_1: $c_3 = \bigodot_{t_1} b_1$.

The above definitions and composition rules form the basis for contract composition as used in XGuide. It is already clear that the introduction of dependent concerns significantly increases the complexity of the overall composition since dependent concerns have to 'understand' the composition operators of the concerns they depend on. Fortunately, the most important contract concerns used in Web applications (the content, the layout and the application logic) do not require dependent concerns. Additional concerns such as access control or meta-data, however, do.

The next section demonstrates a concrete implementation of the formal contract model presented here. Contracts, concerns and composition operators are completely formulated in XML and the result of a composition operation again is an XML document.

5.3 XGUIDE CONTRACTS - XCONTRACTS

Since XGuide focuses on XML-based Web engineering and we aimed at expressing contracts declaratively, it stands to reason to use XML as the contract language. This choice is especially beneficial since some contract concerns can directly be expressed in an XML language. Using XML as contract language also provided us with a rich set of libraries and tools that could be used to create and manipulate contracts.

A first important property of a contract is that it must be uniquely identifiable. Thus all contracts have a unique identifier together with a version. The identifier is necessary to differentiate between the various contracts. The version identifier distinguishes multiple instances of the same contract. Multiple contract instances occur if contracts evolve over time to satisfy extended or changed requirements of pages. Nevertheless other pages referencing the same contract might not fulfill the update and still conform to the original version of the contract. As a result, both the contract identifier and the version identifier are necessary to unambiguously identify a contract.

The structure of an XContract wrapping the contract concerns is shown in Figure 5.1. The *xcontract* document element and all contract-related elements and attributes lie in the `http://www.infosys.tuwien.ac.at/xguide/contract` namespace. The contract and the version identifier are both required attributes on the document element.

```
<?xml version="1.0"?>
<xcontract xmlns="http://www.infosys.tuwien.ac.at/xguide/contract"
           id="SampleContract"
           version="0.9"
>

  <!-- concern definitions go here -->

</xcontract>
```

Figure 5.1: The structure of an XContract.

As we pointed out before, a contract basically is a collection of contract concerns. Each contract concern describes a certain characteristic of the Web component or page and may, or may not, depend on other concerns.

As shown in Figure 5.2, concerns are encapsulated by the *concern* element and again have an identifier attribute. Since we want to be able to use arbitrary XML languages to describe the content of a concern, we use the concept of XML namespaces to distinguish the various vocabularies. In Figure 5.2, the structure concern is represented by an XML schema (i.e., by elements from the `http://www.w3.org/2001/XMLSchema` namespace) and the interface concern is represented by our own interface definition language (i.e., by elements from the `http://www.infosys.tuwien.ac.at/xguide/concerns/interface` namespace).

Having introduced this contract structure, the extensibility requirement of contracts can easily be satisfied. To extend the contract, only a new *concern* element with content in the appropriate markup language has to be added. The only constraint is that the concern identifiers must remain

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract"
           id="SampleContract"
           version="0.9"
>
 <concern type="Structure">
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <!-- XML schema for component goes here -->

   </xsd:schema>
 </concern>
 <concern type="Interface">
  <idl:interface xmlns:idl="http://.../xguide/concerns/interface">
   <idl:in>
    <idl:param name="page_id" type="int" dimension="0" />
   </idl:in>
  </idl:interface>
 </concern>
 <concern type="SomethingElse">
  <ref src="external.xml" />
 </concern>
</xcontract>
```

Figure 5.2: An XContract with concerns from different namespaces.

unique. Another distinguishing criterion for concerns shown in Figure 5.2 is that concerns can be specified directly in the contract or merely referenced in the corresponding concern section. In the case of the *structure* and *interface* concerns, the concern specification is *inline*, i.e., directly provided in the contract using different namespaces. The *SomethingElse* concern, however, is *external*, i.e., only references an external definition stored in the `external.xml` file. The important advantage of external concern definitions over internal ones is that they can easily be reused in multiple contracts. Inline concerns, on the other hand, cannot be embedded in or referenced from other contracts and thus are only usable in one contract.

Before delving into the details of the structure and interface concerns, we first introduce the notion of an *implementation concern* versus a *contract concern*. Contract concerns are the various subsections of contracts that describe the various aspects of the specification. Implementation concerns, on the other hand, are the concerns used in concrete implementations of Web pages and components.

The interesting fact is that the mapping of contract concerns to implementation concerns is not necessarily a one-to-one mapping. The structure contract concern, for instance, is used as specification information for the content implementation concern (i.e., it defines the structure and data types of the content document) and the layout implementation concern (i.e., the development of XSLT stylesheets only depends on the structure information in the schema). Other contract concerns (e.g., the interface concern) directly map to implementation concerns (e.g., in a specific programming language).

As a result, contracts for the dominant three implementation concerns in Web engineering (content, layout and application logic) only require two contract concerns (structure and interface). The following subsections discuss both of these contract concerns in detail.

5.3.1 The Structure Contract Concern

The structure contract concern is the heart of an XContract. It relies on the W3C XML Schema recommendation [24, 143] and defines both the structural and the datatype constraints. Depending on the type of the concern (inline vs. external), the schema document is either directly embedded in or only referenced by the structural concern. We do not present more information about the XML Schema recommendation here but refer the interested reader to the excellent primer of the recommendation [53]. Figure 5.3 shows the (internal) structure concern snippet from a sample contract for a Web page.

5.3.2 The Interface Contract Concern

The interface contract concern is more complex to introduce because no XML dialect covering input- and output interfaces as needed in XGuide exists. As a consequence, we introduce our own markup language to describe the input and output requirements of pages and components. Recall that an XGuide component can only have a single input interface stating the information needed to create the component. In contrast, a component can have multiple output interfaces according to the number of embedded Web forms. Output interfaces consequently require an identifier attribute whereas the single input interface does not.

Both input and output interfaces share a common structure consisting of a set of parameters that have a name, a (data) type and a dimension. The dimension attribute specifies the whether a parameter is an array, and if so, what its dimension is. Accordingly a parameter of dimension 0 represents a single value, a parameter of dimension one a 1-dimensional array of the given type, and so on. Note that the notion of an interface in XGuide differs from interfaces as used in software engineering. XGuide interfaces only contain field members but no methods. The reason for this restriction is that always a single default method is (implicitly) used by clients of the interfaces. For input interfaces of pages and components, this is the *instantiate* operation; for output interfaces it is the *submit* operation. Supporting additional methods in the XGuide interfaces (e.g., display component, destroy page, etc.) only makes sense if the created system needs to be dynamically edited at runtime (see the Dexter hypertext model [75] for details on this idea); XGuide currently does not support dynamic editing of Web applications.

Figure 5.4 depicts the input and output requirements of a Web page that takes a string argument as input and provides two output interfaces (i.e., two separate Web forms). The *search* output interface provides a string argument (e.g., the search term). The *order* interface contains two integer arguments representing a product identifier and the quantity to be ordered. This constellation of interfaces could, for example, occur in a shopping cart contract that takes the user's name as input to display a customized welcome message and provides a product and quantity selection form together with a (separate) search form.

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract"
           id="SampleContract" version="0.9"
>
 <concern type="Structure">
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
              targetNamespace="http://xguide/webpage"
              xmlns="http://xguide/webpage"
   >

      <xs:element name="webpage">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="header" type="xs:string" />
            <xs:element ref="content" />
            <xs:element name="footer" type="xs:string" minOccurs="0" />
          </xs:sequence>
          <xs:attribute name="lang" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>

      <xs:element name="content" type="ContentType"></xs:element>

      <xs:complexType name="ContentType" mixed="true">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element name="b" type="xs:string" />
          <xs:element name="em" type="xs:string" />
        </xs:choice>
      </xs:complexType>

   </xs:schema>
 </concern>
 <!-- other concerns go here -->
</xcontract>
```

Figure 5.3: The structure concern of a sample XContract for a Web page.

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract"
           id="SampleContract" version="0.9"
>
 <concern type="Interface">
  <idl:interface xmlns:idl="http://.../xguide/concerns/interface">
   <idl:in>
    <idl:param name="user_name" type="string" dimension="0" />
   </idl:in>
   <idl:out name="search">
    <idl:param name="keyword" type="string" dimension="0" />
   </idl:out>
   <idl:out name="order">
    <idl:param name="product_id" type="int" />
    <idl:param name="quantity" type="int" />
   </idl:out>
  </idl:interface>
 </concern>
 <!-- other concerns go here -->
</xcontract>
```

Figure 5.4: The interface concern of a sample XContract for a shopping cart Web page.

The only elements in the `http://www.infosys.tuwien.ac.at/xguide/concerns/interface` namespace are the *interface*, *in*, *out* and *param* elements. The *in* and *out* elements encapsulate the input and output interfaces respectively. The *param* element represents a single argument with the specified name and data type in the respective interface. As explained above, the *dimension* attribute determines the dimension of array types. Note that the *dimension* attribute can be omitted to use the default dimension of zero.

The introduction of XContracts, contract concerns and the contract extensibility model is only a preparation for the discussion of how contracts can be composed. The formal model presented in Section 5.2 already defined composition operators for orthogonal and dependent concerns. The basic *structure* and *interface* concerns of XGuide contracts are both orthogonal. The next section extends our notion of XContracts with the definition of composition rules and operators for these concerns.

5.4 Contract Composition

If we think of contracts as XML documents as outlined in the previous section, composition means to process and merge the contract documents to create the resulting (composite) contract– again in XML form. Although their contracts syntactically do not differ, recall that we distinguish between page contracts and component contracts. Only the latter can be embedded into other components and pages; the former cannot be reused or further composed. Thus when we talk about contract composition we always mean the composition of a component contract with a page or another component contract.

Also note that composing two contracts does not create a third, completely new contract but embeds one contract into the other. The resulting hierarchy of composition (i.e., embedding) dependencies is natural for Web applications where reusable page fragments (e.g., the header or navigation bar) are embedded into pages.

Basically, two alternative ways of composing contracts are possible: *composition-by-copy* and *composition-by-reference*. In the first case, the referenced contract is merged with the referencing contract. Once the contracts are merged, the composition is transparent, i.e., no information on what contract was referenced and how the composition operation was performed exists. Using composition-by-reference, on the other hand, the contracts are only linked and the *composition operation* rather than the *referenced contract* is added to the referencing contract.

In the case of XContracts, composition-by-copy means to merge the XML fragments representing the corresponding contract concerns. The resulting XML fragment then replaces the original contract concern definition in the referencing contract. This has the advantage that changes to the referenced contract can never break the referencing contract. This fact, at the same time, is also a downside since updates to the referenced contract are not propagated and have to be integrated manually into a potentially large set of contracts. Even worse, if a contract embeds several component contracts by copy, it might not be possible to undo the composition operation to integrate a new version of one of the component contracts.

Alternatively, using composition-by-reference better facilitates reuse and change propagation. However, it results in a problem similar to the fragile base class problem [136] in object-oriented inheritance. Changes to the referenced contracts are immediately propagated to all referencing contracts, but might well require a subsequent update of the composition operator in the referencing contract. We meet this problem by supporting strongly typed contract references, i.e., both the contract's unique identifier and the full version number are included in the reference. Thus if a referenced component contract is upgraded, it gets a new version number and references to previous versions continue to work. Thus we do not support automatic change propagation to the referencing contracts but require an update to the contract reference to the new version of the embedded contract. The advantage of this approach is that no unexpected changes are propagated and the result of a contract update is more predictable. Updating a contract reference to a newer version of the contract is still easy since only the version information of the contract reference has to be updated.

In the discussion of the formal contract model we stated that contract composition means to iteratively compose all contract concerns. As a result, a contract composition operator consists of a set of concern composition operators for the concern types present in the contracts. Figure 5.5 demonstrates how the concern composition operators are added to the contract. A separate *compositionrefs* element contains all composition references. Composition references are iteratively integrated into the contract. Thus in the example shown in Figure 5.5, first *a.contract* is integrated and then *b.contract* is added to the result of the first composition. Each *reference* specifies a strong contract reference (i.e., the unique identifier and the version of the referenced contract) and provides composition operators for all concerns.

The subsequent sections present in detail the concern composition operators for the structure and the interface contract concerns. They also fill in the missing details of the XML representation of the operators as used in XContracts.

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract"
           id="SampleContract" version="0.9"
>
 <concern type="Structure">
    <!-- structure definition goes here -->
 </concern>
 <concern type="Interface">
    <!-- interface definition goes here -->
 </concern>
 <compositionrefs>
   <reference to="a.contract" version="1.2">
     <composition type="Structure">
       <!-- structure composition operator goes here -->
     </composition>
     <composition type="Interface">
       <!-- interface composition operator goes here -->
     </composition>
   </reference>
   <reference to="b.contract" version="0.9">
     <composition type="Structure">
       <!-- structure composition operator goes here -->
     </composition>
   </reference>
 </compositionrefs>
</xcontract>
```

Figure 5.5: The structure of an extended XContract with concern composition operators.

5.4.1 Composition of Structure Contract Concerns

Obviously, the concern composition operators themselves directly depend on the types of concerns to be composed. We already outlined at the beginning of the chapter that we use XML Schema definitions to define the structure of a component. As a consequence, we need to define operators for schema composition for the structure contract concern.

Our experience in the Web engineering domain and an analysis of existing Web sites led us to the assumption that complex composition operators for Web components are not necessary. Usually components are integrated at prominent places (e.g., only at the top-level of an XML DOM tree or the beginning of a given element). More sophisticated composition operators that could, for example, support integration of components somewhere in the middle of a sequence or collection of elements, are not used. This composition behavior is easy to understand given the fact that complex composition operations immensely increase the complexity of a page but do not add much additional value. This is especially true since re-arranging, modifying or sorting of elements is subsequently done by an XSL transformation rather than by embedding content at the right places.

As an outcome of this observation, we also keep the structure composition operation simple and only support operators that add the referenced structure concern at the beginning or end of an existing element definition. This is already more than is used in most of today's Web applications and still straight forward enough to not increase contract complexity too much. More powerful composition operators that fully exploit or even extend the composition mechanisms in the XML schema recommendation can easily be integrated. Until we experience a need for such powerful composition mechanisms, we only support the default composition operator that specifies the element and where (at the beginning vs. at the end) the referenced component should be added. Figure 5.6 shows a sample structure composition operator.

The contract in the example contains a simple XML schema for a Web page. As defined in the contract's structure concern, a Web page only consists of string content. We then specify a composition reference to the contract of a header component (*header.contract*). We further say that the component should be structurally added at the beginning (in the *position* attribute) of the *webpage* element. Assuming that the header component defines a *header* element of type *HeaderType*, the resulting structure concern is displayed in Figure 5.7. Since the composition is by reference and specified by the composition operator, this resulting concern representation is only used internally.

5.4.2 Composition of Interface Contract Concerns

With the composition operation of the structure concern in place, we now turn to the composition of interface concerns. Composing interfaces is a more complex task than composing schemas since we have to deal with name clashes, type conflicts and dimensional dependencies, i.e., potential adjustments of an argument's dimension. As a first step, we distinguish between composition of input interfaces and output interfaces.

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract"
           id="SampleContract" version="0.9"
>
 <concern type="Structure">
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
              targetNamespace="http://xguide/webpage"
              xmlns="http://xguide/webpage"
   >
      <xs:element name="webpage">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="content" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
   </xs:schema>
 </concern>
 <!-- other concerns go here -->
 <compositionrefs>
   <reference to="header.contract" version="1.2">
     <composition type="Structure">
       <operator elementName="webpage" position="beginning" />
     </composition>
   </reference>
   <!-- more references go here -->
 </compositionrefs>
</xcontract>
```

Figure 5.6: The structure concern composition operator of a sample Web page that references a component contract.

```
 <concern type="Structure">
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
              targetNamespace="http://xguide/webpage"
              xmlns="http://xguide/webpage"
   >
      <xs:element name="webpage">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="header" type="HeaderType" />
            <xs:element name="content" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <!-- more definitions go here -->
   </xs:schema>
 </concern>
```

Figure 5.7: The structure concern after the composition with the contract of the header component.

5.4.2.1 OUTPUT INTERFACES

Recall that output interfaces represent Web forms that are embedded in a page. In other words, an output interface contains an argument for every field, checkbox or option the user enters in a Web form. If the page contains multiple forms, multiple output interfaces are generated. Now imagine that you embed a component containing a Web form into an existing page. This simply means to add the whole form to the page; it does not make sense to combine the form with another, existing form. As a consequence, composition of forms is in most cases cumulative. This means that if we reference a component that contains a form, i.e., has an output interface, we simply add a new output interface to the referencing component. The only potential conflict with this strategy is that a form with the same name already exists in the referencing component which means that we have to rename one of the forms to resolve the name clash.

We consider a single special case when composing output interfaces. If we embed the same form multiple times in a page, we do not add a new output interface for each new instance of the form but unify them with an already existing or previously added one. A good example for this scenario is Google's search result page. It contains a search form both at the top and at the bottom of the result page. In XGuide, we could model the search form as a separate component that is included twice into the result page. The interface concern of the result page, however, would only contain a single output interface that is used by both forms and that submits the search information to the search engine.

5.4.2.2 INPUT INTERFACES

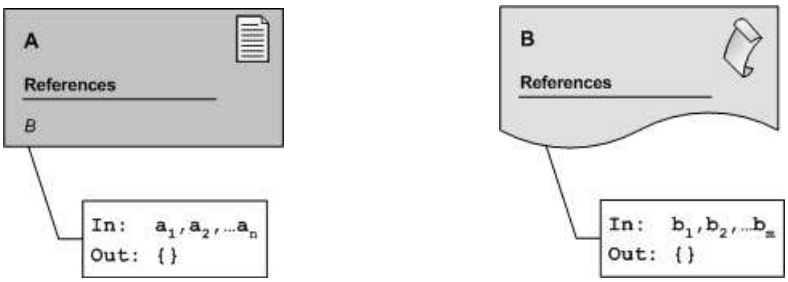
For input interfaces, the situation is more complex. The composition operation does not deal with the whole input interface but defines a composition operator for every parameter in the input interface of the referenced component. This is necessary since the resulting page can again have only a single input interface that has to also include the arguments required for the instantiation of the referenced component. This means that unlike output interfaces, input interfaces have to be merged and potential conflicts between interface arguments must be resolved.

For the purpose of this discussion, consider a page A that references a component B. The parameters of the input interfaces of A and B are denoted as $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_m)$ respectively. The XGuide component web for this scenario is shown in Figure 5.8.

When composing input interfaces, we distinguish the following four composition operations for composing a given parameter b_i with the input interface of A:

COMPOSITION-BY-ADDITION

Composition-by-Addition is the simplest case of composition where no semantic relationship exists between component B and page A. No semantic relationship in this context means that the information in component B does not depend on the information in the input interface of page A. Hence, each b_i is independent of any parameter a_j in the input interface of A. As a consequence, b_i is simply added to A's input interface resulting in the new input interface $(a_1, a_2, \ldots, a_n, b_i)$ for A.

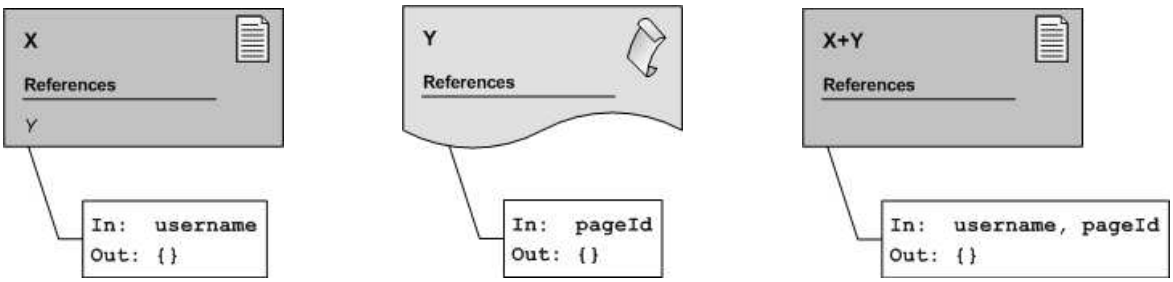


(a) Page A with input interface $a_1, a_2, \ldots, a_n$.

(b) Component B with input interface $b_1, a_2, \ldots, b_m$.

Figure 5.8: The component web for the sample scenario.

Consider a page X that displays the current user's name. Its only input requirement is the user name as shown in Figure 5.9(a).



(a) Page X taking a user name as input.

(b) The semantically independent component Y.

(c) The result of the composition operation.

Figure 5.9: Sample scenario for the composition-by-addition approach.

An example of a semantically independent component Y (with respect to page X) could be a menu bar that takes the current page identifier as input to highlight the currently viewed page or section (see Figure 5.9(b)). The input interface of component Y is clearly independent of the input interface of page X as it does not depend on the given user name. Consequently, the input interface of the composite page $X \bigoplus Y$ is a union of the input interfaces of X and Y, i.e., the user name and the page identifier. The result of the composition operation is shown in Figure 5.9(c).

In the concrete syntax of an XContract, composition-by-addition is denoted as shown in Figure 5.10 (only the interface concern is shown). For each parameter in the input interface of

the referenced contract a `<param-ref>` element is added that contains an operation definition (in the `<op>` element) to specify how the parameter should be composed. In the case of composition-by-addition, the *type* attribute simply states that the parameter should be added to the input interface of the referencing page or component.

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract" id="PageXContract" version="1.0">
 <concern type="Interface">
  <idl:interface xmlns:idl="http://.../xguide/concerns/interface">
   <idl:in>
    <idl:param name="username" type="string" dimension="0" />
   </idl:in>
  </idl:interface>
 </concern>
 <compositionrefs>
   <reference to="y.contract" version="1.0">
     <composition type="Interface">
      <param-ref name="page_id" type="int">
       <op type="add" />
      </param-ref>
     </composition>
   </reference>
 </compositionrefs>
</xcontract>
```

Figure 5.10: A partial XContract demonstrating the syntax of the composition-by-addition composition operator.

Another example for the composition-by-addition operation is a shopping cart application that uses a status component to display the user's current settings (e.g., payment and delivery options). The status component requires the user identification as input parameter *user_id* to identify the user. If we then compose the status component with an overview page that does not require any input parameters, composition-by-addition would add the user identification parameter *user_id* to the overview page's input interface.

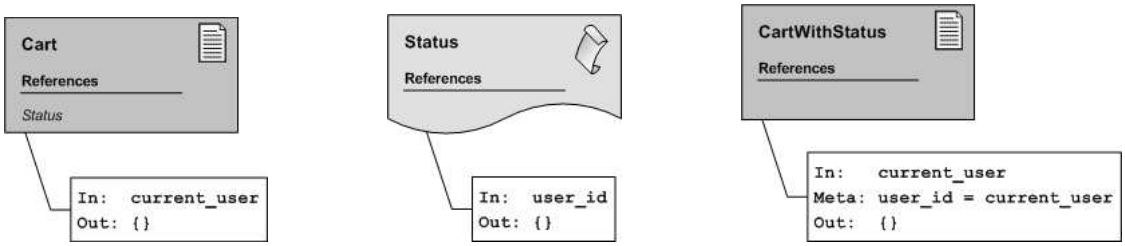
Note that composition-by-addition does not change the type or dimension of the referenced parameters. They are simply added to the existing input interface. To resolve potential naming conflicts between existing and added parameters with the same name, we support renaming of the added parameter. We could, for instance, rename the *user_id* parameter from the previous example to *status_user_id* in the composite to indicate that the parameter is required by the status component. Since no name clashes exist in this example, we can but by no means have to rename the parameter.

COMPOSITION-BY-UNIFICATION

A completely different situation arises if semantically dependent interfaces are composed. Composition-by-Unification deals with parameters b_i that are semantically dependent on a parameter a_j, i.e., represent the same information. In this case, the parameter b_i is unified with an

already existing parameter a_j in the input interface of A. Obviously, a_j and b_i must be of the same type to make this work. The resulting input interface is the unmodified input interface of A. Only the correspondence information between b_i and a_j needs to be preserved.

In a shopping cart scenario, consider a page $Cart$ that displays the contents of the user's shopping cart. It already defines an input interface with parameter *current_user* that is needed to retrieve the current user's shopping cart (Figure 5.11(a)). If we compose this page with the status component (Figure 5.11(b)), we can unify the status component's *user_id* input parameter with the existing *current_user* input parameter. The resulting component has the same input interface as the $Cart$ page–annotated with the correspondence of the *current_user* and *user_id* parameters (Figure 5.11(c)).



(a) The shopping cart page taking a user name as input.

(b) The status component requiring a *user_id*.

(c) The result of the composition operation.

Figure 5.11: Sample scenario for the composition-by-unification approach.

The syntactic representation of composition-by-unification in XContracts is shown in Figure 5.12. As in the composition-by-addition case, each parameter of the referenced input interface is represented by a `<param-ref>` element. The composition operator, however, differs. The *type* attribute of the operator indicates that the referenced parameter is unified with an existing one and the *with* attribute specifies the name of the corresponding parameter.

Note again that composition-by-unification only works for parameters of the same type and dimension. Simple transformation or cast operators could be imagined (e.g., unification of an integer with a long parameter) but are currently not viewed relevant in practice and do not conceptually contribute to the composition operation. Such special unification rules can be easily added by any implementation. Unlike composition-by-addition, the original input interface of the referencing page remains unchanged; only meta-information about the correspondence relationships is added.

COMPOSITION-BY-ADAPTATION

Composition-by-adaptation is an extended version of composition-by-addition. It supports modification of the referenced input parameters depending on the context where a component is

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract" id="PageXContract" version="1.0">
 <concern type="Interface">
  <idl:interface xmlns:idl="http://.../xguide/concerns/interface">
   <idl:in>
    <idl:param name="current_user" type="string" dimension="0" />
   </idl:in>
  </idl:interface>
 </concern>
 <compositionrefs>
   <reference to="status.contract" version="1.0">
     <composition type="Interface">
      <param-ref name="user_id" type="string">
       <op type="unify" with="current_user" />
      </param-ref>
     </composition>
   </reference>
 </compositionrefs>
</xcontract>
```

Figure 5.12: A partial XContract demonstrating the syntax of the composition-by-unification composition operator.

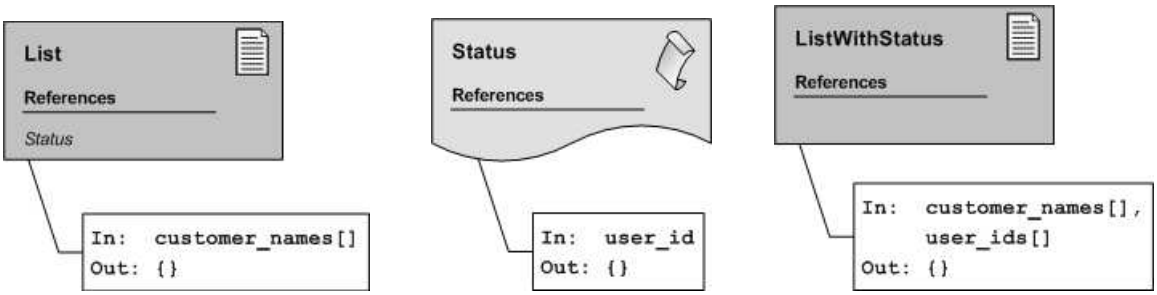
embedded. The typical application of composition-by-adaptation is to embed a component for each item in a given enumeration or iteration in a page.

Imagine a page in the shopping cart scenario that lists all customers with their status. Thus, the status component is not only embedded once, but for every customer in the list. Figure 5.13 demonstrates this situation. The $List$ page takes a list of user names as input parameter. It further references the status component that displays the status of a *single* user and takes the user identification as input (Figure 5.13(b)). In the composition process, we have to update the *user_id* parameter of the status component from a *single* user identification to a *list* of user identifications, i.e., increase the parameter's dimension by one. The updated parameter is then added to the input interface of the customer list page and results in the final input interface shown in Figure 5.13(c).

Thus a b_i in the input interface of the referenced component is not directly added to A, but modified (resulting in $b_{i'}$) and only then added to A's input interface forming the new input interface $(a_1, a_2, \ldots, a_n, b_{i'})$. The adaptation of an input parameter always results in a modified dimension attribute of the parameter. We do not support other adaptations such as type transformations.

The XML notation of the composition-by-adaptation operator is shown in Figure 5.14. The parameter reference of the *user_id* parameter specifies a single integer value (dimension is zero). In the composition operator the dimension is increased by one resulting in an integer array data type.

Frequently but not necessarily is the increase in the dimension of the parameter type an increase by one. If a component is embedded in a cascaded iteration (e.g., a list of customers that for each customer contains a list of accounts with different payment options), the dimension



(a) The $List$ page taking a list of user names as input.

(b) The status component requiring a *user_id*.

(c) The result of the composition operation.

Figure 5.13: Sample scenario for the composition-by-adaptation approach.

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract" id="CustomerList" version="1.0">
 <concern type="Interface">
  <idl:interface xmlns:idl="http://.../xguide/concerns/interface">
   <idl:in>
    <idl:param name="customer_names" type="string" dimension="1" />
   </idl:in>
  </idl:interface>
 </concern>
 <compositionrefs>
   <reference to="status.contract" version="1.0">
     <composition type="Interface">
      <param-ref name="user_id">
       <operator type="add" dimension="+1" />
      </param-ref>
     </composition>
   </reference>
 </compositionrefs>
</xcontract>
```

Figure 5.14: A partial XContract demonstrating the syntax of the composition-by-adaptation composition operator.

can increase by two. Similarly the increase is not always from dimension zero to dimension one. If the referenced component already takes a parameter of dimension one, for example, the dimension will be increased to two. An example of this scenario could be a list of customers that displays the list of items currently in each customer's shopping cart. The shopping cart component would take a parameter representing a list of articles (dimension is one). If we embed this component in a page with a list of customers, the parameter's dimension is changed to two to hold a list of articles for every customer.

A special case of composition-by-adaptation is a combined adaptation and unification. Consider again the example shown in Figure 5.14. If we change the *customer_names* input parameter into a *customer_ids* parameter, we first have to adapt the dimension of the status component's *user_id* by one (since it is embedded in a list) and then unify the modified parameter with the existing *customer_ids* input parameter (since the *customer_ids* parameter already contains the necessary user identification information).

COMPOSITION-BY-OMISSION

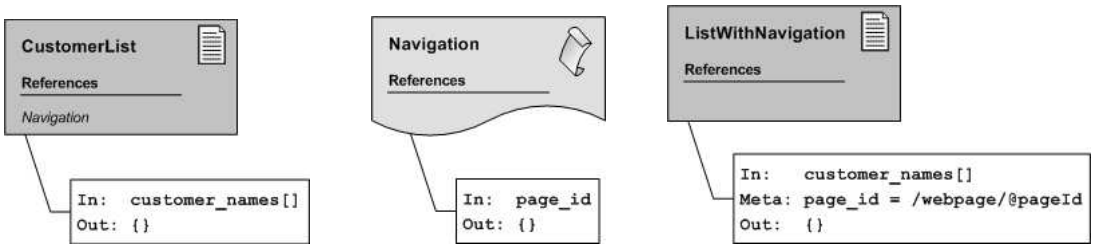
Finally, composition-by-omission describes the scenario when the input parameter b_i of a referenced component is not added at all to the input interface A of the referencing page. This situation occurs if the referencing page itself can provide the input parameter for the referenced component. This most frequently happens with internal parameters that are not visible to the user. The value for such a parameter is directly evaluated by an XPath expression in the referencing page. As a result, the input interface of A remains unchanged. The composition only adds the meta information how to retrieve the value for b_i to the interface concern.

Consider again the page displaying a list of customers as presented in previous sections. Instead of embedding a status component, we now reference a navigation component that is used to display a customized navigation bar depending on the currently viewed page (e.g., the currently viewed item in the navigation bar is highlighted). Figures 5.15(a) and 5.15(b) show the component web for this scenario. The $CustomerList$ page references the $Navigation$ component. The customer list again takes a list of customer names as input. The navigation component requires the page identifier (*page_id*) of the currently viewed page as input argument to highlight the appropriate navigation entry.

The page identifier of a page is usually encoded in the page itself, e.g., as an attribute of the document element. Figure 5.16 shows a snippet from a typical Web page with a page identifier. The page identifier is encoded as an attribute of the `<webpage>` document element.

To satisfy the input requirements of the navigation component, we don't have to add a new parameter to the input interface of the customer list page but simply specify how the component can access the required value. We use XPath expressions for this purpose. In the example in Figure 5.16, the expression would be `/webpage/@pageId`. Thus the additional input requirement of the referenced component is never visible externally but satisfied at composition time.

Figure 5.17 depicts the XML syntax of the composition-by-omission operator. The *type* attribute of the composition operator element indicates that composition-by-omission is used. The



(a) The $CustomerList$ page taking a list of user names as input.

(b) The navigation component requiring a *page_id*.

(c) The result of the composition operation.

Figure 5.15: Sample scenario for the composition-by-omission approach.

```
<?xml version="1.0"?>
<webpage pageId="customer_list">
 <heading>Our Customers</heading>
 <customers>
   <!-- customer information goes here -->
 </customers>
</webpage>
```

Figure 5.16: A snippet from a typical Web page offering a page identifier.

value attribute contains the XPath expression that specifies how the required value is retrieved in the context of the referencing page.

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract" id="CustomerList" version="1.0">
 <concern type="Interface">
  <!-- potential input interface goes here -->
 </concern>
 <compositionrefs>
   <reference to="navigation.contract" version="1.0">
     <composition type="Interface">
      <param-ref name="page_id" type="int" dimension="0">
       <op type="omit" value="/webpage/@pageId" />
      </param-ref>
     </composition>
   </reference>
 </compositionrefs>
</xcontract>
```

Figure 5.17: A partial XContract demonstrating the syntax of the composition-by-omission composition operator.

The discussion of the composition-by-omission composition operator concludes the presentation of input interface composition operators. With the concerns and composition operators discussed so far, the three dominant implementation concerns, i.e., the content, the graphical appearance and the application logic, are fully specified. Nevertheless, the XGuide contract approach is designed in a modular fashion to support plug-ins and thus can be easily extended with additional concerns. Such a concern consists of its representation in XML and the corresponding composition operators for concern composition. The extensibility mechanism can be used to extend contracts with new concerns such as meta-data or access control.

The structure and interface contract concerns are orthogonal to each other with respect to the definition given in Section 5.2. As such, their composition operators do not depend on each other. The remainder of this chapter discusses how concern composition works for dependent concerns and presents the effects on the respective composition operators.

5.4.3 Composition of Dependent Contract Concerns

A contract concern is called *dependent* if its definition depends on the definition of another contract concern. A typical example is the access control concern that determines which parts of a page are visible for a user. The visibility information in the access control concern is specified as XPath expressions. Obviously, these expressions depend on the schema (i.e., the structure contract concern) of the page. If the schema of the page changes, the access control information has to be updated, too.

In terms of the composition operators for dependent concerns, this means that such a composition operator also has to 'understand' the composition operator of the concern it depends on. In

the case of the access control concern, it must first apply the composition operator of the structure concern to derive the new schema information. Then the access control information of both the referencing page and the embedded component have to be updated to reflect the composed page structure.

Consider the example in Figure 5.18. It shows a simple contract for a navigation bar with an access control concern that grants access to all content in the component (i.e., the `<navigation>` document element and all its children are included in an `<allow>` rule).

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract" id="Navigation" version="1.0">
 <concern type="Structure">
   <xs:schema xmlns:xs="...">
      <xs:element name="navigation">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="naventry" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
   </xs:schema>
 </concern>
 <concern type="AccessControl">
   <ac:accesscontrol xmlns:ac="...">
     <ac:allow value="/navigation" />
   </ac:accesscontrol>
 </concern>
</xcontract>
```

Figure 5.18: A contract for a simple navigation bar with access control information.

Figure 5.19 shows a page contract that references the navigation component and specifies the necessary composition operators. It defines a `webpage` element that contains all the content (only indicated in the figure). In the access control concern, only the contents of the first child element of the `webpage` element (i.e., `/*[1]`) is made accessible. The dependency on the structure concern is made explicit in the definition of the access control concern. The *depends* attribute specifies this dependency.

The composition operator for the structure concern defines that the navigation information should be embedded at the beginning of the `webpage` element. The access control composition operator specifies that the access control information in the referenced component should be merged with the access control information of the page. An alternative option could be to ignore the component's access control information.

Though the *depends* attribute is the only explicit hint that the access control concern depends on the structure concern, behind the scenes a lot more has to be considered.

First, the access control information of the sample page in Figure 5.19 might not be valid after the integration of the navigation component since the structure information (i.e., the XML schema of the resulting page) has changed. In our example, the navigation bar is integrated at the

```
<?xml version="1.0"?>
<xcontract xmlns=".../xguide/contract" id="SamplePage" version="1.0">
 <concern type="Structure">
   <xs:schema xmlns:xs="...">
      <xs:element name="webpage">
        <xs:complexType>
          <!-- content definition goes here -->
        </xs:complexType>
      </xs:element>
   </xs:schema>
 </concern>
 <concern type="AccessControl" depends="Structure">
   <ac:accesscontrol xmlns:ac="...">
     <ac:allow value="/webpage/*[1]" />
   </ac:accesscontrol>
 </concern>
 <compositionrefs>
   <reference to="navigation.contract" version="1.0">
     <composition type="Structure">
       <operator elementName="webpage" position="beginning" />
     </composition>
     <composition type="AccessControl">
       <op type="merge" />
     </composition>
   </reference>
 </compositionrefs>
</xcontract>
```

Figure 5.19: A page contract embedding the navigation bar contract.

beginning of the `<webpage>` element. Thus the access control information that granted access to the first child element of the `<webpage>` element must be updated to now grant access to the second child element.

Second, the integration of the component's access control information (i.e., access to everything in the navigation bar) has to be transformed from `/navigation` to `/webpage/navigation` to reflect the embedding at the beginning of the `<webpage>` element.

As we mentioned in Section 5.2.1.2, composition of dependent concerns is considerably more complex than the orthogonal case. Although the *depends* attribute is the only visible sign of dependent concerns in a contract, the logic of concern composition is much more complex. Especially in the case of several, chained dependencies as presented in Section 5.2.1.2 in which case multiple, consecutive composition steps have to be performed.

The discussion of the composition of dependent concerns concludes this chapter on contracts, contract concerns and their composition. Having presented the formal contract model used in XGuide and the concepts of contract composition, the next chapter discusses how XGuide and support for contracts can be used in practice. It presents *XSuite*, an integrated development environment (IDE) for the XGuide methodology that supports the notion of contracts and contract-based development.

CHAPTER 6

XSUITE - AN INTEGRATED DEVELOPMENT ENVIRONMENT FOR XGUIDE

A complex system that works
is invariably found to have evolved
from a simple system that worked.

John Gall

However sophisticated and powerful any methodology is, strong tool support is a crucial requirement for its successful application in real-world projects [10]. *XSuite* is an integrated development environment (IDE) for Web projects following the XGuide process and the concepts presented in the previous chapters. Such an IDE must support the whole life-cycle of a Web project and should view the artifacts of the respective methodology as first class objects. As such, XSuite not only provides a project and resource management platform but deals with contracts, contract concerns, implementation of contracts and deployment strategies.

Within the scope of this thesis and the case study presented in the next chapter, we use the *MyXML publishing framework* as deployment platform [85, 92, 93]. MyXML itself is implemented in the Java programming language and provides good support for the integration of Java-based application logic of Web applications. For the XSuite IDE we also chose Java as an implementation technology for two reasons: first, we strive to achieve the same level of platform independence as the MyXML framework, i.e., only require a Java virtual machine on the target platform. Second, the Java-based Eclipse project [142] provides an extremely powerful and flexible framework for the development of customized IDEs. Despite our choice of Java as implementation language, it is important to note that the XGuide process by no means depends on the Java language or any Java-based technology. It could equally well be implemented on

Microsoft's .NET platform [114], e.g., as a Visual Studio plug-in, using Microsoft technologies such as Active Server Pages (ASP) as deployment platform.

This chapter starts with a brief introduction to the Eclipse project that highlights its great potential for the development of new IDEs. Next we present the concepts of the MyXML publishing framework that is used as deployment platform. The remainder of this chapter discusses the architecture of XSuite in the context of the Eclipse framework and shows how it reflects XGuide's extensibility concepts presented in Section 5.2. The architecture and design decisions presented in this chapter only represent one possible implementation of the XGuide process in a given system environment and using the Eclipse platform and MyXML framework. Though alternative implementations on other platforms and/or using other tools may choose to use different implementation approaches, they may benefit from the concrete realization considerations presented in the context of Eclipse and MyXML.

6.1 The Eclipse Project

The Eclipse Project [142] is a collaborative effort initiated by companies such as IBM, Rational Software, Red Hat, SuSe, TogetherSoft and others. Since November 2001 the consortium grew considerably and now contains about 20 members who stated their commitment and plan to release tools for the Eclipse platform. In the Eclipse project charter the mission statement contains the following sentence that summarizes what Eclipse is: *Eclipse is a kind of universal tool platform - an open extensible IDE for anything and yet nothing in particular.* In other words, the Eclipse platform is a generic environment to build IDEs and highly integrated tools and provides support for common constructs such as projects, resources, build processes, version control, etc. The platform's extensibility mechanism described later in this chapter is a core concept that 'teaches' the platform how to deal with such different resources as Java files, Web content, graphics, video or any other content types.

Figure 6.1 shows the graphical appearance of a generic instance of the Eclipse platform without any additional tools or plug-ins.

The *navigator* (in the upper left corner) is the central resource management component of the Eclipse platform. It lists all available *projects* and for each project a hierarchy of resources that it contains. Eclipse projects are file-oriented, i.e., a project has a base URL in the file system and project resources are mapped to files in the project directory or one of its sub-directories. *Editors* are components to modify resources. Depending on the resource type (e.g., the file extension or content type), different editors such as text editors, color choosers, or graph drawing environments are used. Figure 6.1 shows the built-in editor with the Eclipse welcome message. The content of the editor is an Eclipse system file containing the message that is interpreted and rendered by the editor. In contrast to editors, *views* only display the contents of a resource. More specifically, they provide a special view on the contents of a resource. Thus, multiple different views of the same content can be used to highlight different properties of the content. The *outline view* in Figure 6.1, for instance, could be used to display the overall structure of the welcome message (e.g., contain the sections and subsections and provide links to directly jump to the selected section).

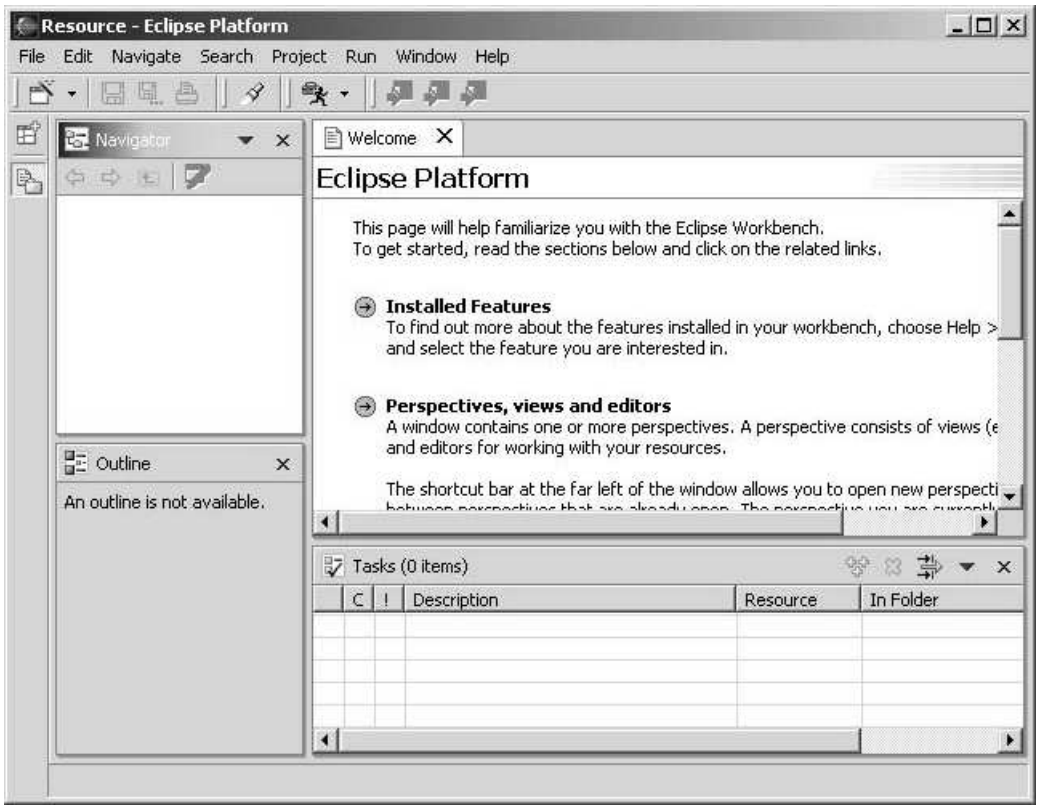


Figure 6.1: A generic instance of the Eclipse platform.

Further the platform provides mechanisms to annotate resources (called *markers*) that can be used to highlight syntax problems, compiler errors, missing information, or anything else. Annotations are displayed in the *task pane* shown at the bottom of Figure 6.1. Project *build processes* define what it means to build a project (e.g., compile source files, apply XSL transformations, deploy Web content, etc.). An incremental project builder keeps track of the changes since the last full build and can build only the changed resources. Additional built-in functionality of the platform includes support for version control systems and a flexible help system.

Obviously this section cannot be a complete introduction or tutorial to the Eclipse platform. It should only give you an impression of how powerful the platform concept is and what the main components are. Many other powerful features dealing with performance considerations, internationalization, deployment, updates, etc. are beyond the scope of this overview. All this information is available via the project Web site at `www.eclipse.org`.

Apart from the core platform, the Eclipse project also includes two sub-projects to support the development of Eclipse-based tools. The first is a showcase for the power, flexibility and extensibility of the Eclipse concept. The Java development tooling (JDT) is a full-fledged Java IDE including syntax highlighting, an incremental compiler, and many other features to ease the software development process. The second sub-project is the plug-in development environment (PDE) that supports the development of Eclipse extensions.

Figure 6.2 depicts the Eclipse Java IDE and the sources for the XSuite tools and plug-ins. The navigator contains the hierarchical source view corresponding to their internal organization in Java packages. An editor component displays the main XSuite plug-in source file with an intentional programming error (the instance variable *resolvedInstallDir* of type URL is assigned the integer value two). As a result, a marker is shown at the beginning of this line, the task pane contains the compile error message and the file's respective graphical appearance in the navigator and the editor's title bar display an error marker.

The basic concept underlying the Eclipse platform is its extensibility via plug-ins. The Eclipse Java IDE is built exclusively using this plug-in mechanism to implement all its functionality and extend the generic platform. The next section investigates the Eclipse extensibility mechanism in detail and gives several examples of plug-ins, i.e., platform extensions.

6.1.1 The Eclipse Extensibility Mechanism

We already used the notion of Eclipse *plug-ins* in the previous section. A plug-in is the smallest unit of functionality and deployment in Eclipse. A plug-in can contain as much functionality as an HTML editor or as little functionality as an action to save a resource. The size of a plug-in depends on its internal architecture, e.g., whether parts of the functionality are intended for reuse or whether other plug-ins should be allowed to extend and refine the plug-in's functionality or not.

The plug-in architecture of the Eclipse platform introduces the notion of *extensions* and *extension points*. An extension point is a well-defined interface that a plug-in or the platform provides that can be extended by other plug-ins. An extension point can be regarded as a hook for other plug-ins to contribute new functionality. The Eclipse platform defines a large set of extension

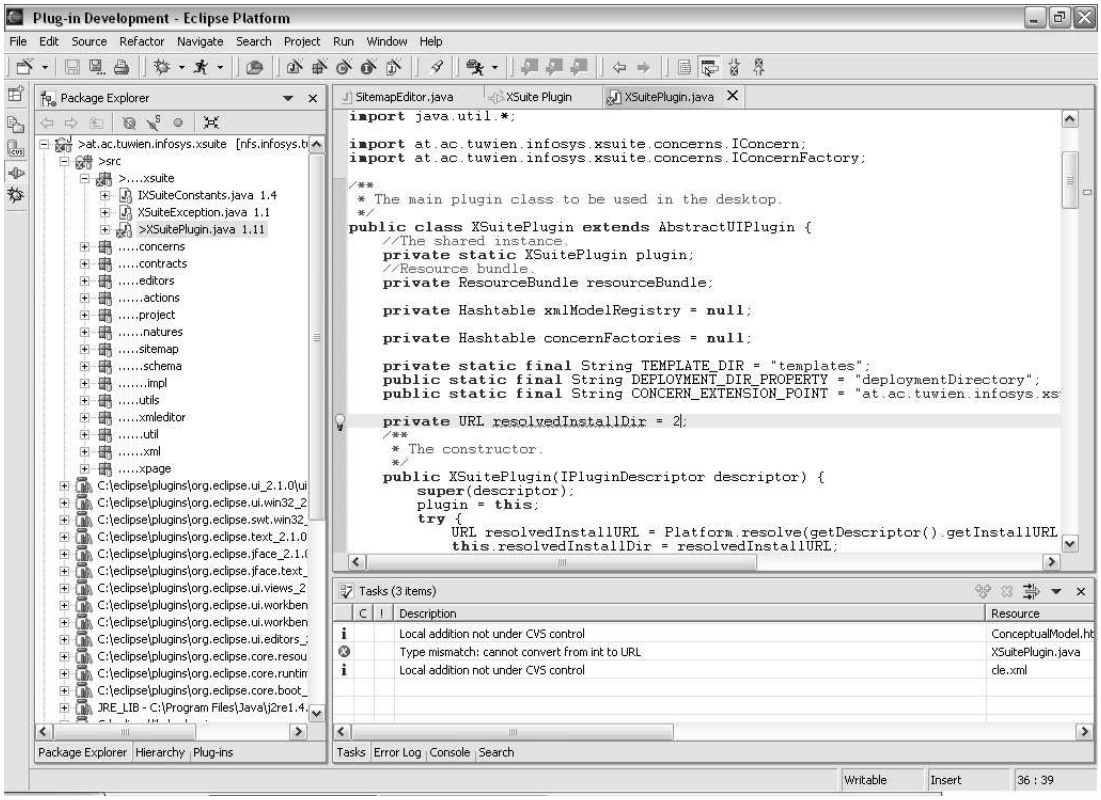


Figure 6.2: The Eclipse Java IDE with the XSuite sources.

points that allow plug-ins to extend and customize almost all aspects of the IDE. Examples include extension points for new project builders, new resource markers, new editors and views, new (popup) menus, new toolbars and toolbar buttons, and many more.

The hooks defined by the extension points are used by other plug-ins that contribute their extensions, i.e., refined or extended implementations of the extension point's interface. Even tools such as the XSuite development tool come in the form of a plug-in that contributes an extension of the *applications* extension point. These plug-ins are loaded when the platform is run.

Figure 6.3 shows the plug-in structure of all Eclipse applications. A small runtime library provides the required plug-in services that the rest of the platform uses. All application plug-ins completely and solely rely on the extension points offered by the runtime and the platform. They in turn offer new extension points to yet another plug-ins that contribute their functionality.

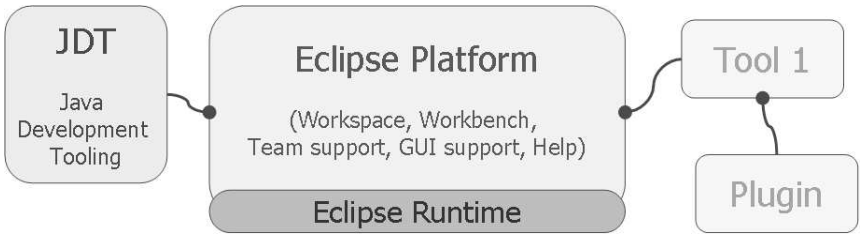


Figure 6.3: The plug-in architecture of the Eclipse platform.

In the context of the XSuite application, the XSuite plug-in contributes to the application extension point and is loaded at startup time. It further defines a concern extension point that supports plugging in new contract concerns at any time. Thus each contract concern is encapsulated in a separate plug-in that contributes to this extension point.

The set of all extension points and all extensions defines a dependency graph between plug-ins. At platform startup, this dependency graph is constructed by analyzing the dependencies (i.e., extensions and extension points) of all available plug-ins. When an application plug-in is loaded, all dependent plug-ins are loaded recursively to complement the application's functionality. Obviously, this process would result in an enormous amount of plug-ins that must be loaded as soon as the platform is started. To avoid this memory and performance bottleneck, each plug-in has an associated *manifest* that contains meta-information about the plug-in. The manifest is an XML document called *plugin.xml* and includes the dependency relationships of the plug-in. As a result, only the manifest information needs to be loaded to create the dependency graph of extensions and extension points. The actual code libraries are only loaded on demand, i.e., as soon as the user initiates an action that requires code from the respective plug-in.

Summarizing, an Eclipse plug-in contributes to one or more extension points, optionally declares new extension points, depends on a set of other plug-ins, and contains Java code libraries and other resources. The plug-in's detailed meta information (contributions, new extension points, dependencies, etc.) is spelled out in the manifest. A snippet of the manifest file for the XSuite application plug-in is shown in Figure 6.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
   id="at.ac.tuwien.infosys.xsuite"
   name="XSuite Plug-In" version="1.0.0" provider-name="Clemens Kerer"
   class="XSuitePlug-In">

   <runtime>
      <library name="xsuite.jar">
         <export name="*"/>
      </library>
   </runtime>

   <requires>
      <import plugin="org.eclipse.ui" version="2.1.0"/>
      <!-- list of other required plug-ins -->
   </requires>

   <extension point="org.eclipse.ui.editors">
      <editor default="true" name="XSuite Sitemap Editor"
            icon="icons/img1.gif" extensions="xmap"
            class="SitemapEditor"
            id="at.ac.tuwien.infosys.xsuite.sitemap.SitemapEditor">
      </editor>
   </extension>

   <extension point="org.eclipse.ui.propertyPages">
      <page objectClass="org.eclipse.core.resources.IProject"
            name="XSuite Project Property Page"
            class="XSuiteProjectPropertyPage"
            id="at.ac.tuwien.infosys.xsuite.projectprops">
      </page>
   </extension>

   <!-- more extensions here -->

</plugin>
```

Figure 6.4: A snippet of the *plugin.xml* manifest file for the XSuite application plug-in.

Every plug-in manifest starts with the plug-in identification consisting of an identifier, a human readable name, a version, an optional provider name and the class implementing the plug-in. The `<runtime>` element contains the location of the plug-in's code and what parts of it should be exported. In the sample manifest, the library *xsuite.jar* contains the code and everything (indicated by the star in the `<export>` element) is exported. The `<requires>` section of the manifest states all dependencies on other plug-ins (including the version of the plug-in). Finally, a list of extensions, i.e., contributions to extension points, is given. The snippet in Figure 6.4 only lists two. First an extension to the *editors* extension point is specified. It defines a customized editor for XGuide sitemaps with the extension *xmap*. The second extension contributes a new property page to the platforms collection of property pages. The property page offers a set of options to configure the behavior of the XSuite application. Other extensions defined in the XSuite plug-in manifest (not shown in the figure) include customized project definitions for Web projects, more editors and several wizards to create new resources such as contracts, concerns or XPages.

After this brief introduction to the Eclipse platform, its potential and the basic plug-in extension mechanism, we now introduce the deployment platform used in the XSuite reference implementation and the case study: the MyXML publishing framework.

6.2 The MyXML Web Publishing Framework

The MyXML Web Publishing Framework consists of a language and a compiler that supports the creation of XML-based Web applications while keeping the content, the layout information and the application logic separate. MyXML is an implementation language (in the terminology of the XGuide approach) that we developed together with our colleagues Engin Kirda and Roman Kurmanowytsch. As in the case of the Eclipse framework, we only give a brief introduction to the potential of the MyXML framework. Details, examples and our experiences in the deployment of MyXML-based Web applications can be found in [85, 86, 91, 93–95].

The World Wide Web Consortium's eXtensible Markup Language (XML) [31] along with the eXtensible Style Sheet (XSL) [3] technology aim at solving the layout and content separation problem. Ultimately, complete layout independence can be achieved by the use of XML and XSL. Although the layout and content separation problem has been attacked intensively (e.g., by standards such as XML, XSL, Cascading Style Sheets, etc.), the problem of separating the application logic from the layout and content in dynamic Web applications has not received much attention yet. Most popular Web technologies (such as PHP, JavaScript, Active Server Pages (ASP) and Java Server Pages (JSP)) are XML unaware and do not exploit its capabilities. These tools and technologies lack support for the creation and maintenance of layout-independent *dynamic* Web content.

MyXML is an XML/XSL-based template engine that supports a strict separation of layout, content and application logic. The content and its structure are defined in well-formed XML documents, the layout information is given as an XSL stylesheet and the application logic is defined separately in an arbitrary programming language. The template functionality of the

MyXML engine is exploited by using special MyXML elements in the input XML document. These tags (i.e., elements) are defined in the MyXML template language that is based on the MyXML namespace definition (see [84] for details). The layout stylesheets that can be applied to a MyXML document are arbitrary XSL transformations.

The functionality of the MyXML template engine is based on the MyXML process which defines the actions to be taken depending on the type of the MyXML input document. Figure 6.5 shows the MyXML process.

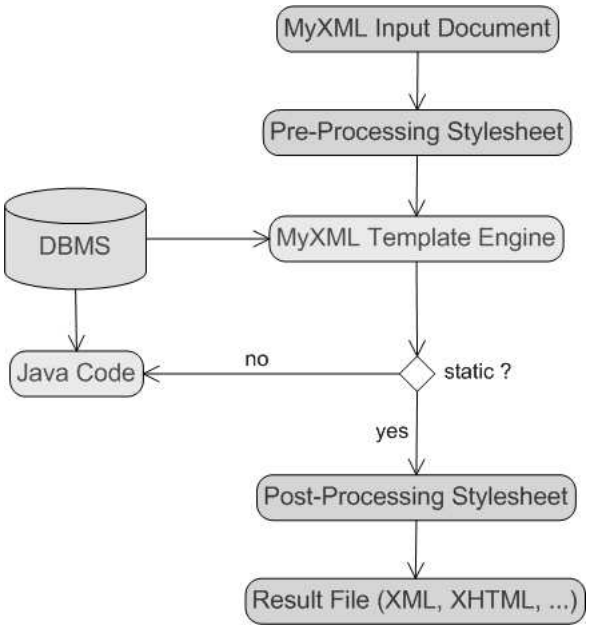


Figure 6.5: The MyXML process.

The process starts with a MyXML input document. Any well-formed XML document which may contain elements of the MyXML template language can be used as input document. In the next step a pre-processing XSL stylesheet can be applied to add layout information to the document. Additionally, the XSL stylesheet can be used to add static information to a document (e.g., header and footer) or to restructure the input document. After having passed this second step of the MyXML process, the template engine processes the modified input file.

The MyXML template engine distinguishes between two kinds of input documents: static and dynamic documents. A MyXML document is considered *static* if all MyXML elements can be resolved at processing time. A *dynamic* MyXML document, on the other hand, contains at least one dynamic MyXML element such as a reference to a CGI parameter or a dynamic database query which can only be evaluated at runtime.

If the MyXML engine detects a static input document, it processes the MyXML elements that it contains. Optionally, it applies a post-processing XSL stylesheet before creating the result file which usually, but not necessarily, is an HTML or an XML file. Such a post-processing stylesheet could be used to add additional layout information based on the result of a database

query (e.g., alternate the background color of a database-backed HTML table for every second row).

If a dynamic input document is passed to the MyXML template engine, source code has to be generated which handles all dynamic aspects defined in the input document. Arbitrary programming languages can be supported by the MyXML engine since a special code generator interface represents the link between the MyXML template engine and the application logic. By implementing this interface for a given programming language, support for that language can be added to the MyXML engine. The reference implementation supports generation of Java code which can easily be used by servlets or other programs encapsulating the application logic of a Web application.

Figure 6.6 shows a sample MyXML content file. It uses the `<myxml:sql>` and `<myxml:cgi>` elements to model the user input (i.e., the selected event) by means of a CGI parameter and a database query depending on the user's choice. In addition, we provide the contents of the *title*, *date* and *description* fields in the result set using the `<myxml:dbitem>` element. These values are formatted further by the XSL pre-processing stylesheet and then processed by the MyXML engine that retrieves the actual value of the CGI parameter, executes the database query and returns the result document.

```
<?xml version="1.0" encoding="US-ASCII"?>
<!DOCTYPE event_search>
<selected_event xmlns:myxml="http://www.infosys.tuwien.ac.at/ns/myxml">
    <myxml:sql>
      <myxml:dbcommand>
        SELECT title, date, description FROM VIF_EVENTS
        WHERE id = <myxml:cgi>id</myxml:cgi>;
      </myxml:dbcommand>
      <event>
        <title><myxml:dbitem>title</myxml:dbitem></title>
        <date><myxml:dbitem>date</myxml:dbitem></date>
        <description>
          <myxml:dbitem>description</myxml:dbitem>
        </description>
      </event>
    </myxml:sql>
</selected_event>
```

Figure 6.6: A sample MyXML content page querying a database to display an event with a given identifier.

The MyXML template language has several other elements besides `<myxml:sql>` and `<myxml:cgi>`. The `<myxml:loop>` and `<myxml:multiple>` elements allow the engine to repeatedly process parts of a document (e.g., for generating the list of items a user has stored in a shopping cart). The `<myxml:single>` element represents a user-defined variable, whose value is determined at runtime (e.g., the name of the user currently logged in). The `<myxml:attribute>` element can be used to dynamically set the attribute of another element (e.g., the *src* attribute of an HTML `<IMG>` or the *href* attribute of an HTML link). A detailed

discussion of all these elements (and several more) as well as their attributes can be found on the MyXML homepage at `www.infosys.tuwien.ac.at/myxml/`.

Although MyXML-based Web solutions usually include more files and have a higher complexity than traditional HTML-based solutions, it adds a great amount of flexibility, reusability and maintainability to the site. Using the strict separation of layout, content and application logic makes it easy to change or reuse any of the three parts independently of the others. All that is needed after an update or modification of any part of the Web site is a regeneration of the affected pages using the MyXML template engine.

There are many template-based products and tools for Web development in the market. Most of these tools are HTML oriented and do not support a clean separation of content, formatting information and application logic. These tools do not satisfy the requirement of an XML-based implementation technology and are consequently excluded from being an XGuide implementation technology.

Apart from MyXML, the Apache Cocoon project [107] is an example of an alternative XML-based implementation technology that can be used with XGuide. It offers support for the clean separation of layout and content and to a lesser degree of the application logic. In contrast to the MyXML approach that tries to do as much processing as possible at compile time, Cocoon performs all processing steps at runtime. As a consequence, it uses a sophisticated caching mechanism and defines a processing pipeline that starts with the XML content assembly, followed by a sequence of (XSL) transformations to bring the content in the desired output structure or markup language and a final serialization step that encodes the result document and returns it to the client.

For this thesis we prefer MyXML as an implementation technology over Cocoon since it has a more flexible concept of separating the application logic from the content and better supports Web components. As mentioned above, Cocoon could also be used for the implementation phase but would require a much more sophisticated build process since Web components are not supported.

After this brief introduction to Eclipse and MyXML that are the core technologies for the implementation of the XSuite IDE, we now present selected details of the implementation supporting the XGuide development process.

6.3 XSUITE CONCEPTUAL MODELING

To optimally support the XGuide process with software tools, we must not only provide tools for single tasks or steps in the process but need an environment that supports the full life-cycle of a Web application as defined in the XGuide process shown in Figure 4.1. The conceptual modeling part of the process includes the requirements analysis and *design-in-the-large* activities that result in an XGuide sitemap. We already introduced the syntactic notation for the requirements diagram and the sitemap in Chapter 4. Creation and editing of the diagrams is currently not fully integrated into the Eclipse IDE but supported by an external modeling tool: Microsoft Visio (version 2002). Although Eclipse provides a graph-drawing plug-in that could also be used for the generation of the respective diagrams, extending and customizing Visio reduced the overall

programming effort and lets users benefit from Visio's usability features. Figure 6.7 depicts the XGuide environment and the shapes available in Visio.

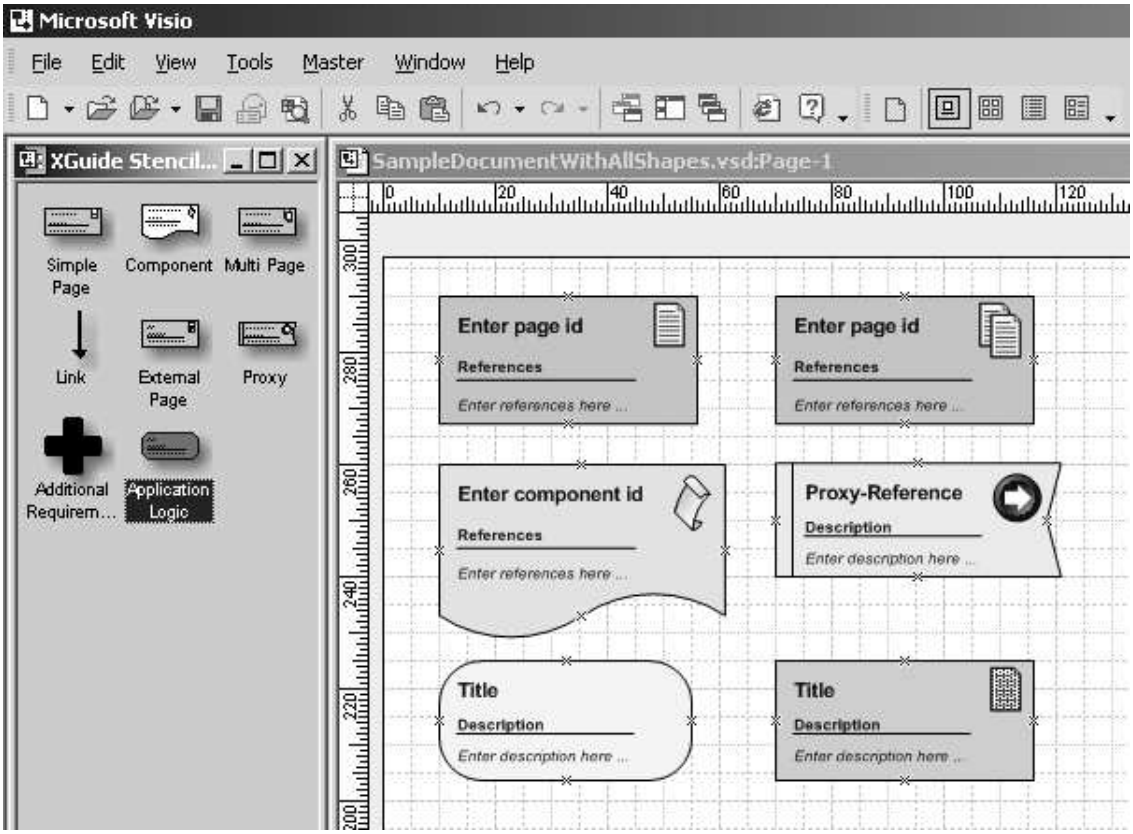


Figure 6.7: The extended Visio workspace for XGuide development.

Visio supports a concept called *stencils*. A stencil is a collection of drawing artifacts, connectors and shapes that topically belong together. For instance, Visio provides built-in stencils for UML modeling, network diagrams, database design and workflow charts. For modeling XGuide requirements diagrams and sitemaps, we created an additional stencil with the shapes presented in Chapter 4 and shown again in the figure. A stencil contains so-called *master shapes* that act as templates and can be dragged into a document. In the document, a copy of the master shape is created that can be further adapted. Figure 6.7 demonstrates a sample document with the shapes as they appear in a document after being inserted from the stencil. The default captions and text information is subsequently replaced by the user.

By extending the Visio environment for XGuide diagram modeling rather than developing a proprietary editor, we also inherit Visio's comfortable editing functionality: shapes grow dynamically in height and width depending on the amount of text they contain, connection points support persistent linking of shapes even if they are moved in the document or rearranged, Visio's routing algorithm can be used to optimally lay out the shapes on the pages, input and output interfaces can be edited using popup dialogs (an example is shown in Figure 4.10), etc. The final

sitemap is then exported as Visio XML drawing and transformed into the sitemap structure as presented in Figure 4.15 using an XSLT stylesheet that strips away all Visio and editing related information and restructures the remaining information appropriately.

6.4 XSUITE ECLIPSE IDE

As mentioned above, the XSuite IDE is an application plug-in for the Eclipse platform. This section discusses the architecture of the XSuite IDE and presents some of the key decisions for the implementation. Since the actual IDE implementation not only depends on the Eclipse platform but also on third party libraries, we encapsulated all required libraries in separate plug-ins and installed them in the platform. Only then can the XSuite IDE be plugged into the (modified) platform using the plug-in extension mechanism. Figure 6.8 shows this correlation in a layered diagram.

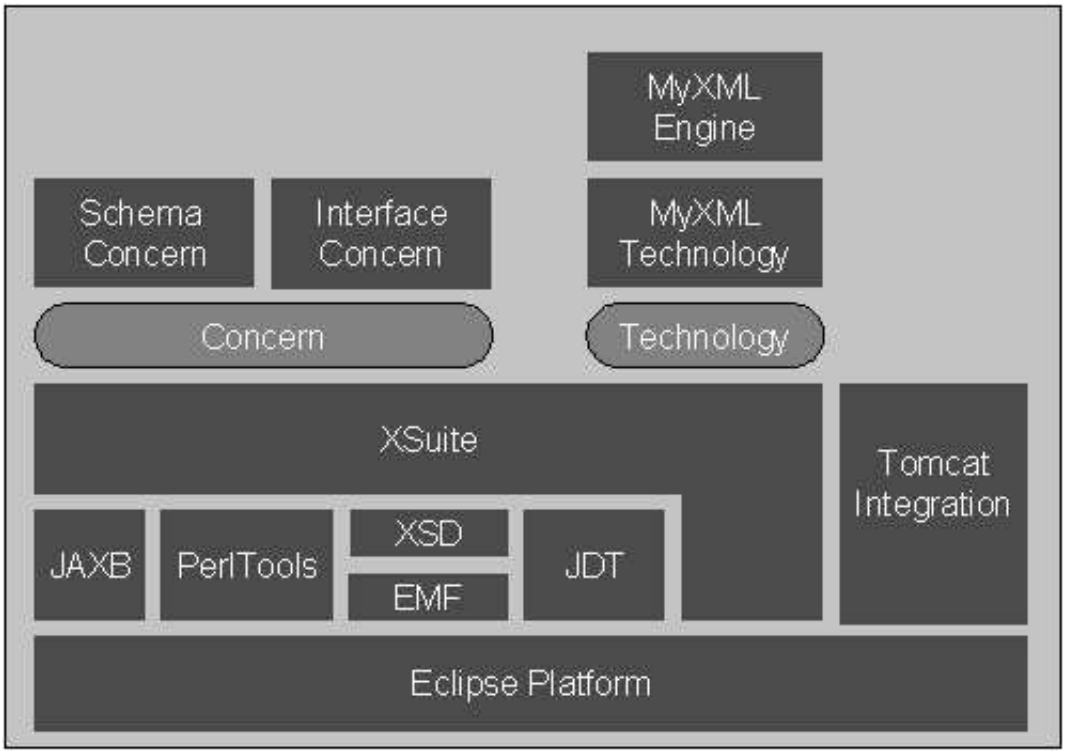


Figure 6.8: The dependencies of the plug-ins constituting the XSuite IDE.

The Eclipse runtime constitutes the bottom layer. The plug-ins required by the XSuite IDE are JAXB (Java XML Data Binding), PerlTools (regular expressions), EMF (Eclipse Modeling Framework), XSD (XML Schema Support), and JDT (Java Development Tooling). They all provide functionality that is directly used by the XSuite plug-in. The plug-in in turn defines two core extension points: the *Concern* extension point and the *Technology* extension point. The

former supports plugging in new contract concerns, the latter supports different implementation technologies. Independent of the XSuite IDE, the Tomcat plug-in integrates the servlet container into the Eclipse platform.

Figure 6.8 explicitly shows only the two XSuite specific extension points for concerns and technologies. Effectively, many more (platform) extension points are involved in building the functionality of the IDE. Table 6.1 shows an overview of the extensions contributed by the XSuite plug-in (it does not list the extensions provided by required plug-ins, concern plug-ins, technology plug-ins and the Tomcat plug-in).

To complement the discussion of plug-in mechanism used by the Eclipse platform, the following subsections present the XSuite-specific extension points for concerns and technologies in some detail. They also introduce the interfaces that stand behind the extension points and explains the rationale for their design.

6.4.1 The Concern Extension Point

The concern extension point defines a hook that supports plugging in new contract concerns (in addition to the default structure and interface concerns). Such concerns could capture the navigation requirements, access control information, or meta data for a given page or component. To integrate a new concern with the XSuite IDE, the concern has to interact with the IDE in many situations. The plug-in interface specifying all such interactions is shown in Figure 6.9.

```
public interface IConcern {
   public String getTypeId();

   // representation of concerns
   public Element asXML() throws XSuiteException;
   public boolean hasDataModelSupport();
   public Object getDataModel() throws XSuiteException;

   // composition of concerns
   public void createConcernCompositionUIFor(Listener validationListener,
          Composite c, IConcern concernToCompose) throws XSuiteException;
   public String isConcernCompositionInfoValid();
   public IConcernComposition getConcernCompositionInfo();
   public void composeWithConcern(Node concernNode, IConcern otherConcern,
          IConcernComposition compositionInfo) throws XSuiteException;
}
```

Figure 6.9: The Java interface for contract concerns.

The `getTypeId()` method simply returns the unique identifier of this concern used to group and compose concerns of the same type. The remaining methods in the plug-in interface can be divided into two groups: methods dealing with the representation of concerns and methods for the composition of concerns.

The native representation of a concern is XML. The `asXML()` method provides access to the document object model (DOM) of the concern. In some situations, however, more sophisticated

Table 6.1: XSuite extensions contributed to the Eclipse platform.

Extension Point	**XSuite Extension**	**Description**
`ui.editors`	`XMLEditor`	Contributes an XML editor that supports syntax highlighting and auto-completion for XML tags.
	`SitemapEditor`	Contributes a visual editor for XGuide sitemaps.
	`MultipageEditor`	Contributes an editor that splits XML files into several parts and lets the user edit the parts in a tabbed pane.
`ui.documentProviders`	`XMLDocumentProvider`	Contributes the document provider for the XML editor.
`core.resources.natures`	`XSuiteNature`	Defines the characteristics of XGuide projects; the nature is automatically added to new projects.
`ui.newWizards`	`ProjectCreationWizard`	Contributes a wizard for the creation of XGuide projects.
	`XPageCreationWizard`	Contributes the wizard for the creation of new XPages (based on a contract).
	`ContractCreationWizard`	Contributes a wizard to create new contracts.
	`FileCreationWizard`	Contributes a wizard to create new implementation files.
`ui.importWizards`	`SitemapImportWizard`	Contributes a wizard to import Visio sitemaps into the IDE.
`ui.propertyPages`	`ProjectPropertyPage`	Contributes a property page for XGuide projects.
`ui.actionSets`	`ContractActions`	Contributes the set of actions (i.e., menu items, toolbar buttons, etc.) used in the XSuite IDE.

or comfortable data models for concerns exist. The structure concern, for instance, uses a schema data model to represent an XML schema rather than the DOM tree of the schema. All concerns must support the XML representation of the concern, customized data model support is optional. The `hasDataModelSupport()` method returns whether a concern has a separate data model. If so, the `getDataModel()` method returns it.

The other methods in the concern interface describe the interaction for contract, i.e., concern composition. First the `createConcernCompositionUIFor()` method is a callback for the corresponding composition wizard and adds a wizard page with all user interface elements needed to gather the composition information for this concern. For the structure concern this is the position where the new schema should be embedded. For the interface concern, the composition information includes operators for all parameters in all input and output interfaces. The `isConcernCompositionInfoValid()` and `getConcernCompositionInfo()` methods are used to validate that all required information was entered and to return an object representing the composition information. Finally, the `composeWithConcern()` method composes the concern with the concern passed as method argument using the information in the composition information argument.

6.4.2 The Technology Extension Point

The second extension point defined by the XSuite plug-in specifies the interface to the implementation and deployment technology used in the project. Per default, the MyXML technology is used for this purpose, but other technologies could replace it by implementing this extension point. Figure 6.10 depicts the callback methods for new technology plug-ins.

```
public interface IImplementationTechnology {
   public String getName();

   // contribute pages to project creation wizard
   public IProjectCreationContribution getProjectCreationContribution();

   // contribute pages to XPage creation wizard
   public IXPageCreationContribution getXPageCreationContribution();

   // create a template for the given implementation concern
   public IFolder getFolderForImplType(IFolder implementationFolder,
          String type);
   public void createTemplateFor(String implConcern, Element parentNode,
          IConcern contractConcern, IFile target) throws XSuiteException;

   // build, compile, deploy the project
   public String getProjectBuilderName();
   public IncrementalProjectBuilder getProjectBuilder();
}
```

Figure 6.10: The plug-in interface for new implementation technologies.

Apart from the `getName()` method that returns the name of the implementation technology, the interface has methods for the creation of new implementation and XPage files, the building of the project and the gathering of configuration information at project creation time.

The `getProjectCreationContribution()` method provides optional wizard pages for the project creation wizard. These pages gather configuration information for the implementation technology. In the case of MyXML, for instance, we ask for the default deployment directory, the default output type, etc. Similarly, the `getXPageCreationContribution()` method contributes pages to the XPage creation wizard. Again, the pages collect implementation specific information such as the output name, the processing type or the processing scope (in the case of MyXML).

The `getFolderForImplType()` and `createTemplateFor()` methods are used to create a new implementation template for the specified concern and retrieve the folder where such concern implementations are stored. Using the MyXML technology, the method would generate a MyXML document for the content implementation concern, an XSL stylesheet for the layout concern and a Java interface and factory class for the application logic concern.

The final group of methods deals with the building of projects, folders and files. Depending on the implementation technology, building a project has a different meaning. In MyXML it means to process all XPages with the MyXML engine, compile the generated Java sources and copy all static and dynamic resources to the deployment directory. The `getProjectBuilderName()` method returns the name of the customized project builder that is subsequently used to locate and initialized the plug-in containing the builder. The `getProjectBuilder()` method then provides access to the builder implementation.

The presented interfaces for contract concerns and implementation technologies form the backbone of XSuite's extensibility mechanism that in turn relies on the Eclipse extensibility mechanism via plug-ins. Figure 6.11 depicts a slightly modified UML package diagram that includes plug-in boundaries in addition to the actual software packages. It further only shows dependency but does not model communication relationships.

The total number of classes contributing to the functionality of the XSuite IDE (including the MyXML engine but excluding the Tomcat integration and the required libraries such as JAXB, EMF, XSD, etc.) is 355 (containing a little over 38 000 lines of source code). A detailed discussion of the concrete implementation of the full XSuite functionality is not the purpose of this work. Thus, the remainder of this chapter focuses on three interesting implementation decisions that influenced the development of the XSuite IDE: the separation of implementation concerns with MyXML, the application of JAXB (Java XML Data Binding) and the contract cache.

6.5 Separation of Concerns with MyXML

The MyXML introduction in Section 6.2 explains that MyXML separates the content, the layout and the application logic. The separation of the content from the layout is done using XML and XSLT. For dynamic pages, however, the separation of the application logic from the content and the layout is more interesting. In this case, MyXML first generates an interface that describes

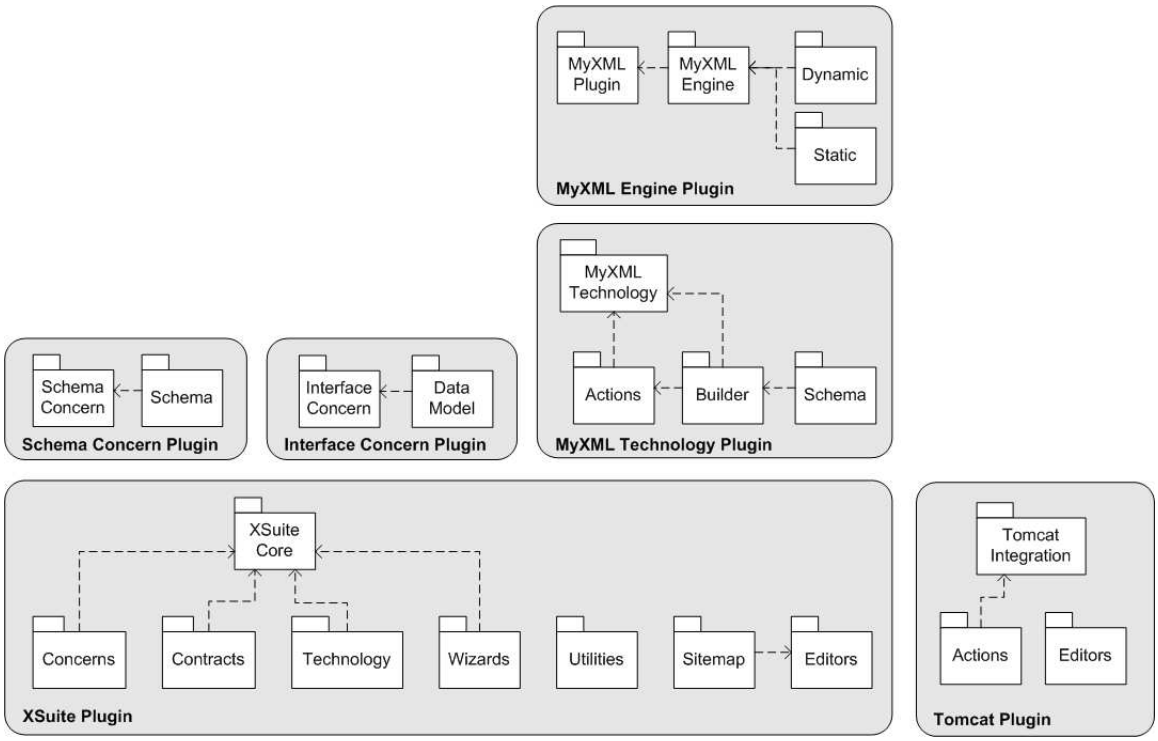


Figure 6.11: The modified UML package diagram for the XSuite IDE.

the input/output behavior of the page. It then generates the class encapsulating the content and the layout for the page and ensures that this class implements the generated interface.

Figure 6.12 shows the MyXML generated interface for a page `SamplePage` that takes a string `title` as input parameter and provides an output interface for the string array `value1` and the integer `value2`.

```
public interface ISamplePage {

  public void setInput(String title);
  public void print(PrintWriter pw) throws Exception;

  public interface IOutput {
    public String[] getValue1();
    public Integer getValue2();
  }
}
```

Figure 6.12: The MyXML generated interface for a sample page with input and output interfaces.

For the purpose of parallel development of Web applications, however, it must be possible for the programmer to implement the application logic *without* the concrete MyXML generated class with the content and layout (i.e., the output class).

As a consequence, XSuite uses the factory pattern [61] for the creation of output classes. When the Web application is deployed, the factory will create an instance of the MyXML generated output class and return it to the caller. During the implementation phase, however, the XSuite IDE generates the interface and a dummy implementation of the interface from the contract. The factory is then configured to return an instance of the dummy implementation which lets the programmer work independently of the content manager and the graphics designer.

In practice, two XSL transformations are used to create the interface and the factory with the dummy implementation from the contract. The contract already contains all information (i.e., the page's input and the output interfaces) required for this processing. This code generating transformation also takes the type and the dimension of the parameter into accounts.

For non-string types, the transformation generates appropriate conversion operations since Web forms only deal with strings. Thus an integer parameter in an input interface has to be converted to a string; in the same way an integer output parameter requires a conversion from the string value provided by the Web form to an integer object.

Parameter dimensions greater than zero further complicate the code generation process since we have to convert arrays to and from Web form values. Web forms don't have the notion of array parameters but require the mapping of an array to a sequence of single value fields. As a solution we use the name of the array as a base name of the form field and append the value's array index to the base name for the final name of the field. When submitting a Web form, we scan all parameters and reconstruct an array from the submitted values using the 'base name plus index' naming convention.

To generate a dummy implementation of the output interface that the factory can return immediately even if the final output class does not yet exist, the transformation code creates a class that encapsulates a simple HTML page with the following properties:

- For each parameter in the input interface specification, the page contains a line that states the name, the type and the value of the parameter.
- For each output interface, the page contains a separate Web form. Again the name and type of the expected value is displayed. For array values a sequence of five consecutive input fields is used.

Using this approach, the programmer can implement the application logic without any dependencies on the content or the layout. She can even test the interaction among multiple servlets by using the generated dummy classes that implement the same interface as the final output class will.

6.6 JAXB - Java XML Data Binding

When working with XML documents in a software application, it is a common task to parse an XML document and create a Java data structure from it rather than working directly with the XML (DOM) tree. With the Java XML Data Binding (JAXB) specification [135], Sun Microsystems provides a schema-based approach to ease the parsing, creation and validation of XML documents. Starting from an XML schema, the schema compiler creates a set of Java interfaces and classes that are able to represent all documents that comply with the schema.

In the application, marshaller and unmarshaller classes provide operations to read and validate XML documents and to access their data model representation. The actual mapping of schema elements to data model elements can further be adapted by so-called customized bindings that specify how a schema element should be translated into a data model entity.

In addition to reading of documents, also the creation of schema compliant documents and their serialization to an output stream is supported in JAXB. In XSuite we use JAXB to work with configuration files, the MyXML project file, interface definitions and sitemap models.

6.7 The Contract Cache

In Chapter 5 we discussed in detail contracts and contract composition. We also stated that contract composition is performed 'by reference' in XSuite which means to only embed a pointer to the referenced contract and the required composition information as opposed to the actual contents of the contract.

As a result, contracts in XSuite have a `<compositionreferences>` section that contains all such composition pointers. In some situations, however, we need the expanded version

of the contract, i.e., the contract after the composition operation was performed and the referenced contract was integrated. One such situation is the composition of a contract with another contract. We have to first expand all existing composition references since the new composition information must be collected relatively to expanded version of the contract.

Consider that we need to specify the composition information for a structure contract concern. The composition information consists of an element name and the position within the element where the new contract is to be embedded. Obviously we must first expand the current contract to retrieve the list of available elements rather than using only the elements in the current contract's schema and ignoring any elements from referenced contracts.

Contract composition requires to iteratively compose all contract concerns of all referenced contracts. As a result, the composition operation is extremely expensive. Because of the event model of Eclipse wizards, access to the expanded version of a contract is needed frequently. To avoid the performance penalty of continuously expanding contracts, we introduced the so-called *contract cache* that stores the expanded version of a contract and invalidates the cache entries if the contract is modified.

6.8 Generating Canonical XML from an XML Schema

In the XPage creation wizard, the final task is to generate implementation templates for all implementation concerns in the page. For the content this means to generate an XML document that reflects the structure and data type information of the page's contract. In other words, we needed to generate an XML document from a given XML schema.

Since an XML schema represents a class of documents rather than a single instance, the generation of a template file cannot be deterministic. Instead, it has to be decided how to react on `<xsd:choice>`, `<xsd:sequence>` or `<xsd:all>` elements and what to do with facets such as the minimum and maximum number of element occurrences. For the use in XSuite, we defined a simple algorithm to derive an instance document from a schema. We apply the following rules:

- For every element with a simple type, create a string value stating the type's name.
- For every element using `<xsd:choice>`, continue with the first element in the choice enumeration.
- For every element using `<xsd:all>` or `<xsd:sequence>`, process the elements in the order they appear in the schema.
- For every `minOccurs` facet that is zero, add a comment stating that the following content is optional.
- For every `maxOccurs` facet that is unbounded, add a comment stating that the following content can be repeated infinitely.

- If `minOccurs` and `maxOccurs` are within a given lower and upper bound, generate `minOccurs` number of such elements and add a comment with the `maxOccurs` upper bound.
- If `minOccurs` and `maxOccurs` are not within the given bounds, insert a fixed number of elements and add a comment stating the `minOccurs` and `maxOccurs` values.
- For attributes, always insert the attribute and add a comment if the attribute is optional.

We introduce the term *canonical XML* for an XML document that is derived from a given schema using a set of agreed upon rules. Some requirements for these rules are that they should be able to create customized default values for simple type elements and attributes that take the base type as well as existing facets into account. For choices, a comment should outline the other possible content models for a given type. The handling of `minOccurs` and `maxOccurs` facets should be parameterized to support more flexible document generation.

It is clear that such rules can be formulated, it remains an interesting exercise, however, to think about reasonable assumptions in these rules and to implement a full-fledged generator for canonical XML from a given schema.

This chapter started with an introduction to the Eclipse and MyXML technologies. It then presented the extensibility mechanism and the architecture of the XSuite IDE plug-in and discussed some implementation decisions. What is still missing, is the functional description of the XSuite IDE, i.e., how the XGuide process is mapped onto activities in the development environment and what wizards, dialogs, editors, project builders, deployment strategies and consistency checks the tool provides.

To better illustrate XSuite's functionality we postpone the functional description to the next chapter where we introduce the Vienna International Festival case study. On the basis of this case study, we will demonstrate how the XGuide process is supported by XSuite and what functionality the IDE offers to the developer.

CHAPTER 7

THE VIENNA INTERNATIONAL FESTIVAL (VIF) CASE STUDY

Looking at the proliferation of personal web pages on the net,
it looks like very soon everyone on earth will have 15 Megabytes of fame.

MG Siriam

In this chapter, we demonstrate how the XGuide process and XSuite IDE were deployed for a first real-world case study: the VIF 2003 Web application.

The Vienna International Festival is a major cultural event in Vienna. This annual festival usually lasts six to eight weeks over a period in May and June. The festivities take place in various theater locations and concert halls and consist of operas, plays, concerts, musicals and exhibitions. Often, famous international directors, performers and ensembles are guests. The VIF attracts many visitors from around the globe. Most of the international visitors, however, come from neighboring countries such as Germany, Italy and Switzerland. As a consequence, the content in the Web application must be fully bilingual (German and English).

Since its first presence on the Web in 1995, the VIF has been changing its look-and-feel every year according the that year's promotion theme of the festival. The services the site provides also vary annually. Information such as event locations, current programme, an archive of earlier performances, news updates and press reports are traditionally provided to visitors. Additional features such as feedback facilities, discussion forums, and hosting of smaller festivals and series depend on user feedback, Web server log statistics and the theme of the festival. Further, one of the key services of the application is the online ticket ordering. Users can choose and buy tickets online using a shopping-basket application. In total, the whole application consists of hundreds of static and dynamic pages and attracted more than 70 000 (different) visitors in two months (resulting in about 6 million hits and more than 22 gigabytes of network traffic).

Until the year 1999, we had been using our HTML-based technologies for building and managing the VIF Web presence. These template-based tools enabled us to achieve some flexibility, but we were not able to have a strict separation of the content and the layout, let alone the application logic. Starting in 2000, we implemented the application using XML, XSL and our MyXML template engine technology [85, 92, 93]. Our aim at that time was to achieve a high level of flexibility in order to decrease the necessary effort in integrating new layout and service requirements.

With the MyXML implementation technology and the work introduced in this thesis, i.e., the XGuide process and the supporting tools, we now cover – in addition to the implementation level – also the conceptual and methodological level. The remainder of this chapter follows the XGuide process to discuss the design and implementation of the case study. It starts with the presentation of the requirements for the VIF 2003 Web application and consecutively focuses on selected parts of the application for an in-depth discussion to show our experience with the use of XGuide.

7.1 Analysis of VIF Requirements

The analysis of the requirements for the VIF 2003 Web application were driven by the VIF managers. The four main areas of concern were the programme information, the ticket ordering system, the VIF archive and the dissemination of general information about the festival (e.g., contact information, special offers, cooperations, payment options, etc.). Since the whole application is bilingual, the customer should be able to switch languages at any point in time, i.e., every German page (be it static or dynamic) has an English counterpart it is linked to. Additionally, a browseable Web gallery was envisioned that contains images and textual information on selected events. The possibility to integrate future news releases, press reviews, critique, highlights and interviews concluded the list of main requirements for the VIF 2003.

In the context of the VIF project, we identified the following stakeholders that need to interact to get the project implemented:

- **VIF Managers.** The VIF managers are the owner of the Web site and are concerned about meeting deadlines and getting functionality implemented. The pay for the work done by the other internal and external stakeholders.

- **Content Managers.** The content managers are VIF staff and thus internal stakeholders. Their only concern is the correctness and freshness of the information available via the Web application.

- **Graphics Designers.** The design of the visual representation of the content was outsourced to an external design company whose only task is to produce the (HTML) page templates.

- **Programmers.** In the VIF scenario, we take the role of programmers who implement the application logic and integrate it with the graphical layout and the content. In addition we perform maintenance/evolution tasks such as content updates or extensions of the application's functionality (e.g., special offers if you buy more than 5 tickets).

- **System Administrator.** The system administrator role is responsible for running the infrastructure (e.g., hardware, network, Web server, etc.) and provide secure, fast and reliable access to the services. We assumed this role in cooperation with the VIF's technician.

- **Customers.** The customers are the users of the VIF Web application who want to get information about the festival and order tickets.

In the following we provide detailed information on the programme and ticket ordering requirements. We use this central and most complex part of the application for our case study description in the remainder of this chapter.

The programme overview page is the central source of information for the client regarding events, locations and dates. It contains a list of all events, shortcuts to the ticket ordering system for each event, indicates whether events are still available and lets the customer browse events by location, date or keyword. A link for each event brings up the respective event details page that contains detailed event information (e.g., title, author, actors, location, dates, short description of the content, an image, etc.). Event details pages further support navigating the set of all events or a specific search result with links to the previous and next event in the set.

The ticket ordering process covers all aspects of selecting, reserving, ordering and paying for tickets and manages a shopping cart that the client can modify at all times. Furthermore, (host-based) third-party services running on a mainframe computer need to be integrated into the workflow to perform online credit card validation and to interface with the ticket issuing system used by the rest of the festival's ticket offices (e.g., to globally reserve tickets).

The ordering process is initiated from the programme overview page or from the event details page. First, the list of available performances of the selected event is shown. When the customer has chosen the desired date, a list of available ticket categories and the number of free seats in each category are displayed. She now selects the number and category of all tickets she wishes to buy. In the next step, the current contents of the customer's shopping cart is shown, including the newly added tickets. This list already contains the exact row and seat numbers of the tickets and allows the customer to cancel any number of tickets. At this time, she can either choose to go back to the programme to order more tickets or to finish the transaction. In the latter case, a page requests the customer's personal and payment information and lets the customer commit the order. When the order was successfully processed, she obtains a confirmation page on the screen and an email with the same confirmation information (including the ordered tickets, the grand total and the reservation number).

Following the XGuide process, we modeled these requirements in a requirements diagram using only simple and multi pages. Figure 7.1 shows the requirements diagram of the programme and shopping cart sections of the application.

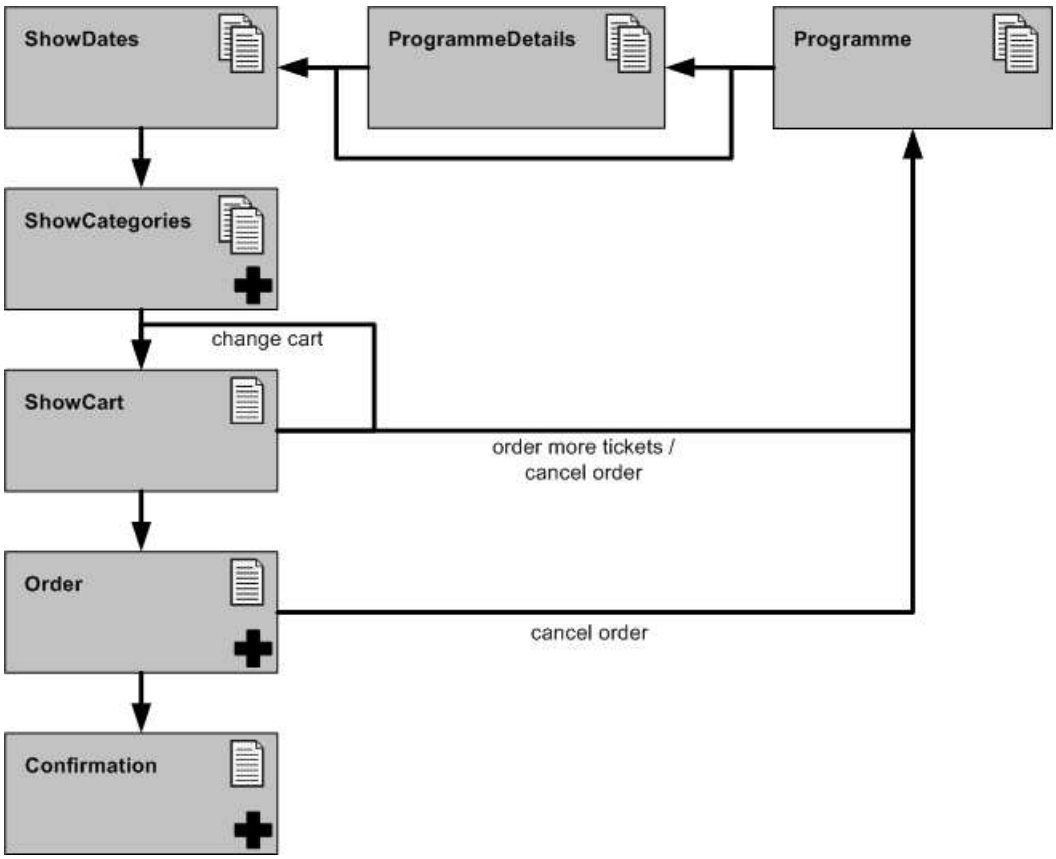


Figure 7.1: The requirements diagram for the programme and ticket ordering sections of the VIF 2003 application.

The programme overview page *Programme* is the starting point for any event information or ticket ordering process. It is modeled as a multi page since it is parameterized with the criteria the customer selected (e.g., date range, location, type of event, etc.). After the client selected an event, the corresponding event details are displayed in the *ProgrammeDetails* multi page that is parameterized with the identifier of the selected event.

From both the programme main and the event details pages, the customer can start the online ticket ordering process. In a first step, all dates for the selected event are shown (*ShowDates*). Having selected a particular date, the various ticket categories and the number of available tickets per category are displayed (*ShowCategories*). The customer can then select the number of tickets per category she wants to put into the shopping cart. As a result of putting tickets in the shopping cart, the contents of the cart is shown in the *ShowCart* page. Besides the possibility to cancel the ordering process, this page provides options to go back to the programme to add more tickets for other events, to remove selected tickets (*change cart*) from the current shopping cart, and to proceed to the *Order* page to finalize the ordering process.

The final page in the online ordering process requires the customer to enter her personal information (e.g., name, address, email, phone number, etc.) and the preferred payment and delivery options. Once all information is provided, the order is processed on the back-end system including credit card validation and online reservation of the selected tickets in the ticket issuing system. Eventually, the customer receives a confirmation page (*Confirmation*) containing details on her order (e.g., the exact seat and row numbers, reservation id, grand total of all tickets, selected delivery option, etc.).

Also note the 'additional requirements' indicators on the *ShowCategories*, *Order* and *Confirmation* pages. The respective requirements cards state that

- the number of available tickets per category in the *ShowCategories* page must be retrieved from the third-party ticketing system,
- finishing the order in the *Order* page means to first validate the credit card information (again using a third-party service) and then book the tickets in the back-end ticketing system,
- in addition to the *Confirmation* page, the customer must get a confirmation email containing the same information as the confirmation page. Further the same email is for redundancy reasons sent to the VIF ticketing office (to cross-check in the ticketing system).

7.1.1 Discussion

First it needs to be emphasized that the requirements diagram explained above is the result of an iterative discussion process. We started out with a much simpler version of the diagram. When discussing the workflow, however, additional requirements were added and the diagram evolved. Examples of such changes include:

- removing single tickets from the shopping cart's *ShowCart* page (as opposed to canceling the whole shopping cart or all tickets of a selected event),

- adding shortcut links to the *Programme* overview page to directly start the ticket ordering process (as opposed to requiring the customer to first navigate to the event details page),
- showing the exact number of available tickets per category in the *ShowCategories* page (as opposed to merely indicating whether tickets in the respective category are available or not).

Another requirement we were not able to easily capture in the requirements diagram pertains to the bilingual nature of the Web application. As a consequence, every page exists in both languages and switching from one language to the other should be seamlessly implemented throughout the application. Theoretically this would mean to duplicate the requirements diagram and for every page to add a navigational dependency to its counterpart in the other language. This would render the diagram almost unreadable. With respect to the requirements diagram, we decided to consider the diagram to be language-independent (i.e., not to model the switching of languages at this level); instead we added this requirement as a separate section to the requirements document.

An important aspect with respect to the following feasibility decision are represented by the non-functional requirements. Apart from the technical requirements of integrating legacy or third-party systems, two other requirements had significant impact on the project: the short development time and the expected change rate of the application.

In terms of development time, a hard deadline for the Web application was determined by the date of the public press conference in which the programme of the next year's VIF is presented. The press conference for the VIF 2003 programme was scheduled for December 13, 2002. In the project planning meetings it became clear that the final layout and most of the content will only be available in late November. This left us with about 2-3 weeks to integrate the content, the layout and the back-end application logic.

The expected evolution rate of the application was the other significant concern. On the one hand, the amount of information to deal with continues to grow as press releases, interviews with artists and reviews of performances become available. On the other hand, our experience with Web projects and the VIF in particular shows that managers and Web site owners usually do not think of all requirements at the beginning of the project but tend to come up with new requirements later in the project. Examples are given in Section 7.6 detailing on maintenance and evolution activities.

7.2 The Feasibility Decision

Since the VIF case study is a relatively small Web project, assessing the feasibility of the project was relatively easy. Following the checklist presented in the XGuide process, we evaluated the project in terms of money, time, people, dependencies, quality of service, know-how and technology.

Because the implementation of the case study was set up as a cooperation with the VIF, money was not the primary concern. Also the people involved in the project and their responsibilities and roles (content management, project management, graphics design, implementation) has become clear over the years. In terms of dependencies on third-party products and services only the credit card validation service and the ticket issuing system were identified. The latter is a host-based system using terminal emulations to gather user input. As a consequence, an adapter service to translate between the Web-based shopping application and the back-end ticketing system had to be developed.

Regarding the know-how of the people involved in the project, the situation was more problematic. The people at the VIF office are mostly users of the Web but did not have much experience developing Web-based applications. As a result, we planned for increased communication to discuss what is possible and reasonable to do on the Web. Also in terms of hardware infrastructure (server machines, service maintenance, network security, etc.) the VIF could not provide the necessary know-how as outlined when discussing the technology aspect below. The graphics design was done by an external company whose expertise is in creating HTML-based layouts using WYSIWYG tools such as Dreamweaver and scripting using JavaScript and Macromedia Flash. In the case of the VIF case study, we considered the know-how of the developers as being no problem since we have enough experience developing, deploying and maintaining Web applications. We also developed the technologies used during the implementation phase (e.g., MyXML, XSuite) ensuring sufficient familiarity with the tools and technologies.

The project setup as a cooperation with the VIF gave us the possibility to decide on the technological aspect of the project. For the VIF 2003 we use an Apache Web server running on Linux as base platform. The application development is done using XML, XSL, MyXML and Java servlets. The back-end data store is realized as MySQL database. Taking this setting into consideration, the technology aspect of the feasibility decision brought up two potential problems: first, the VIF did not have experience in setting up or maintaining a Linux-based Web server; second, the graphics designers did not have knowledge of XSL but only HTML.

The solution to the first problem was that we took over the responsibility for configuring, securing, running and maintaining the Web server machine and services. Since it was not an option for the graphics designers to provide the layout as XSLT templates, we planned for a 'preprocessing' task to analyze and transcode the delivered HTML mockups into XML and XSLT definitions.

Also what kind of quality of service is required for the VIF Web application was an interesting question. Since the Web presence of the VIF is important especially for foreign customers, availability was the main concern. To cover the possibility of hardware defects that cannot be fixed by replacing the hardware, we have a cold stand-by solution, i.e., copies of all required software packages to be able to setup an alternative Web server in a couple of hours. Obviously, also all the data is backed up. The security of the system is ensured by a restrictively configured firewall that only accepts HTTP(S) requests and monitoring of the Web log files. Only for the network connection to the Internet, the VIF fully rely on their provider which constitutes a single point of failure.

The scalability of the system is not a major concern since the Web server load still by far does not tap the full potential of the server and the lifetime of the Web application will end

with the final events in June (i.e., be less than a year). The only critical point with respect to scalability is the communication with the adapter service for the ticketing system since only a limited number of connections are accepted by this service. On certain days (e.g., the first day when the ticketing service is available, the day of the opening ceremony, etc.) we expect the amount of concurrent ticket orderings to exceed this limit. Regarding the performance of the application, the situation is similar. The performance of the Web and database server are more than sufficient, the communication with the adapter service (which in turn connects to the ticketing system), however, causes some delays. Since the adapter service is provided by an external party, we cannot directly influence its properties.

Regarding the project duration and implementation time, however, the situation was more difficult. We already mentioned that the available development time was extremely short and it was not clear at the beginning whether we could finish the implementation in the given time frame. As a result, we agreed to implement all functionality except the ticket ordering before the press conference and, if necessary, provide the ticket ordering functionality at a later date. As it turned out, the integrated support for separation of concerns let us finish everything in time (since we could implement all the functionality independently of the actual data and graphical appearance). Still the online ticket ordering could only start a month after the press conference since the VIF could not provide the necessary ticket information.

7.3 Designing the VIF Web application

The design of the VIF case study builds on the requirements diagram created in the first step. The design in-the-large activity refines the requirements diagram to include Web components, application logic processes and the necessary input- and output interfaces. Consecutively, design in-the-small translates the high-level design into concrete contracts for all pages and components. Further it specifies the composition operators to embed components into pages.

7.3.1 Design In-the-Large

In the requirements diagram, the envisioned Web application is viewed from the client's perspective. Design in-the-large means to further analyze the requirements diagram to identify commonalities across pages (and separate them out into Web components) and to precisely specify all interfaces (i.e., between pages, components and application logic processes).

Figure 7.2 depicts the snippet from the diagram showing the programme overview, search and details pages. The *Programme* page references three Web components: the *Header*, *Programme-Search* and *ProgViewDefault* components. The header component is embedded in all pages to display a common header section and navigation structure for all pages. The programme search component represents a search facility that lets the client search for events depending on the selected date, location and event category. Finally, the (default) view components displays the list of events that match the search criteria.

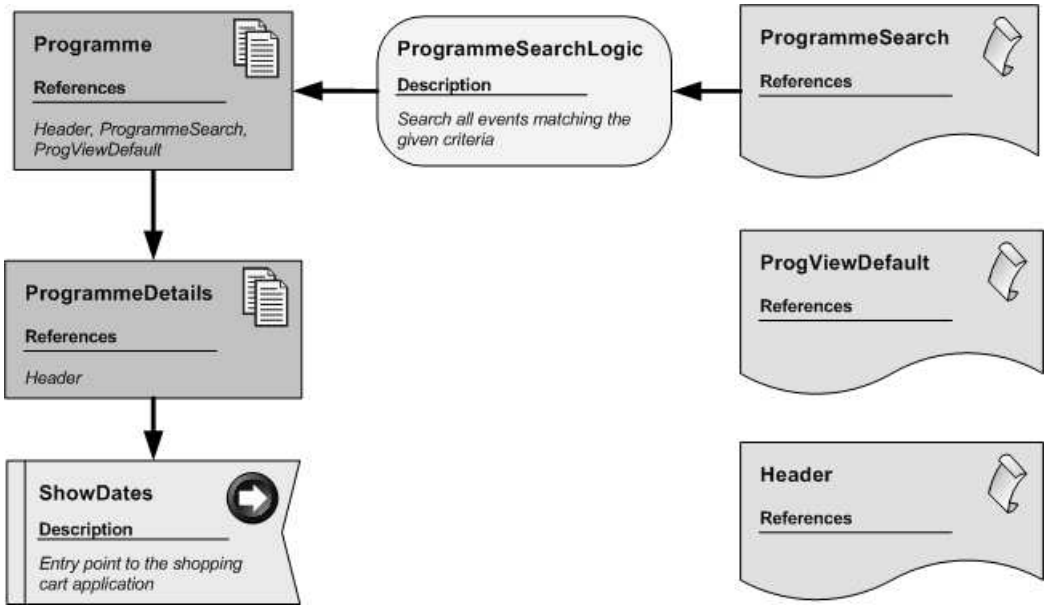


Figure 7.2: The snippet of the design diagram responsible for the programme overview, search and details pages.

Further the search component is linked to an application logic process that performs the actual search against the back-end database containing all events and displays the search result in yet another page of type *Programme*. This recursive dependency is also the reason why the programme page is modeled as multi page as opposed to a simple page—each search query produces a new instance of the page that only differs in the list of events to be displayed. When initially navigating to the programme page, no restrictions on the search criteria are specified resulting in a list of all events in the view component.

From the list of events in the view component, the client can select an event to view its details. The details page reuses the header component and displays all information available for the selected event. The *ProgrammeDetails* page again is a multi page since its page specification is parameterized with the identifier of an event and reused for all events. From the programme details page, also the ticket ordering process starts. If the client chooses to order tickets for the selected event, she is first presented with a list of all dates when the event is performed. In the design diagram, we modeled the ticket ordering process on a separate page (shown in Figure 7.3). As a result, we used a proxy component in Figure 7.2 to indicate the fact that the *ShowDates* page is specified somewhere else (i.e., on a different page) in the diagram.

Since the input and output interfaces are properties of the components and pages set via a property dialog, they are not shown in Figures 7.2 and 7.3. In Figure 7.2, for instance, the *ProgrammeSearch* component has an output interface that specifies the selected search criteria, i.e., the date range, the location and the event category. This output interface is matched by a corresponding input interface of the *ProgrammeSearchLogic* application process that uses the provided values to compile the list of events that match the search criteria. Consequently, the

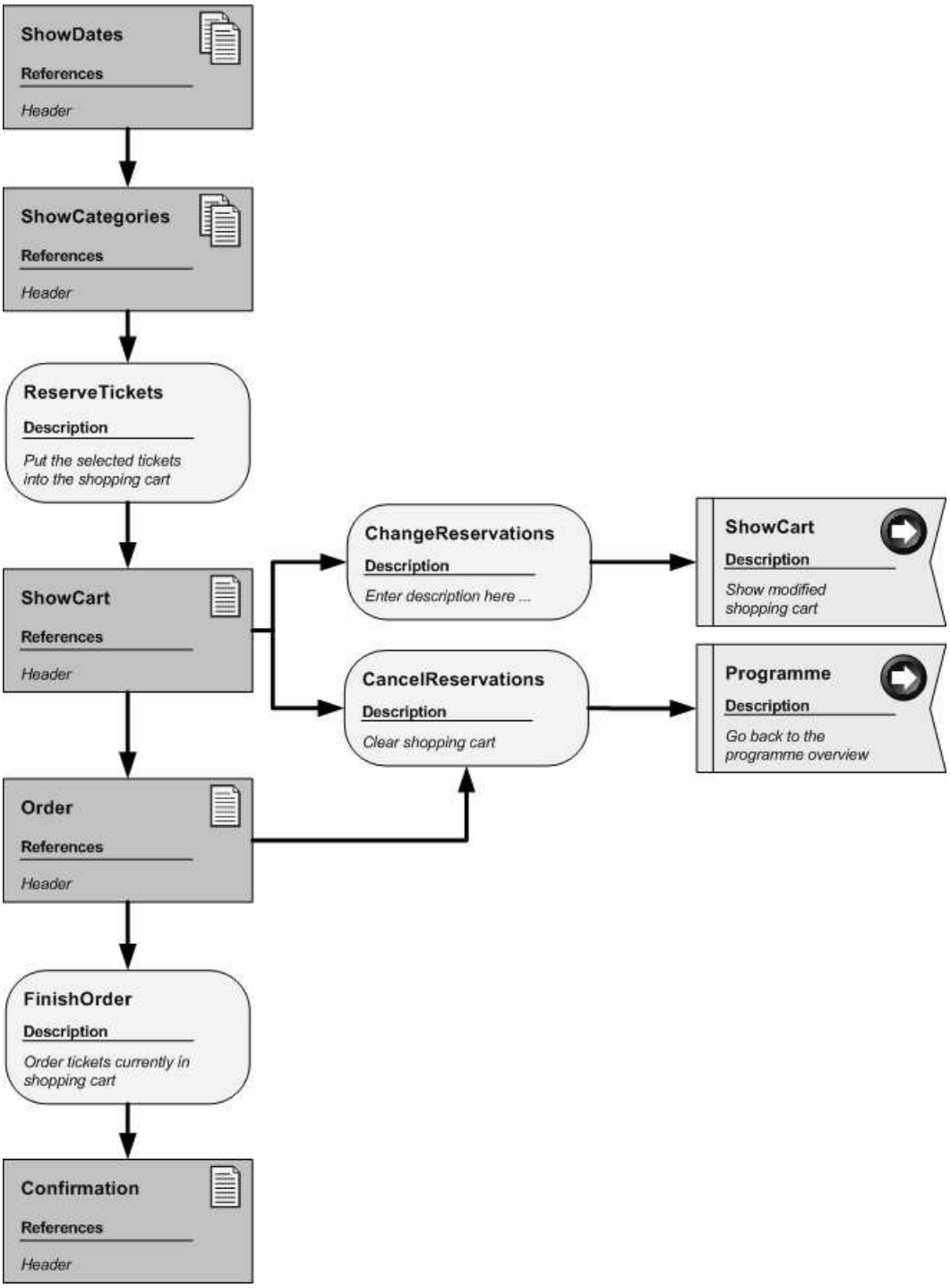


Figure 7.3: The page from the design diagram that specifies the ticket ordering process.

process defines an output interface that contains a list of events. This list is used as input interface in the *Programme* page and displayed as search result. The duality of input and output interfaces continues for all pages.

To finish the discussion of the design in-the-large activities in our case study, we briefly walk through the diagram for the ticket ordering process. The process starts with the aforementioned *ShowDates* page that displays all dates when the selected event is performed. When the client has chosen a date, the available ticket categories (i.e., price categories depending on the event location) are displayed together with the number of tickets available in each category in the *Show-Categories* page. The client then selects the number of tickets she wants to order (per category) and puts them into the shopping cart. On the server-side, this triggers the (internal) reservation of the tickets (*ReserveTickets*) to avoid placing the same tickets into multiple shopping carts at the same time. The *ShowCart* page displays the current content of the shopping cart and provides buttons to change or cancel the order. In the former case, the reservation is changed on the server and the modified shopping cart displayed anew. In the latter case, the reservation is canceled and the programme overview is shown. If the client decides to finish the order, she is asked to enter the delivery and payment information on the *Order* page. Again she can cancel to reservation and return to the programme overview; or she chooses to finish the order,i.e., affirmatively booking the tickets on the server and, in the case of a credit card transaction, validating and charging her credit card. Eventually, a confirmation page with all ticket details is shown. Note that all pages embed the header component to ensure a consistent appearance and navigation experience.

7.3.2 DESIGN IN-THE-SMALL

So far, all modeling was done in the Visio modeling environment. Starting with design in-the-small, visual modeling is replaced by the XSuite IDE. To make the Visio sitemap usable in XSuite, it is first translated into an XML representation. The parts of the VIF sitemap that model the programme pages are shown in Figure 7.4.

You can see that only the essential information such as the identifiers, the names, input and output interfaces, composition dependencies and link information are extracted from the Visio diagram.

The XSuite IDE provides a customized editor for sitemaps and supports creation of contracts for all the pages, components, and application logic processes in the sitemap. To create a contract for a page or component, the developer first chooses among the available contract concerns. All pages and components must have a *Structure* concern that defines their structure and data model. Dynamic pages (and components) might in addition use the *Interface* concern to define their input/output behavior. Application logic processes, on the other hand, only use the interface concern since they do not have content or a visual representation themselves but only consume and provide information for pages and components.

The contract creation wizard integrated into the XSuite IDE supports the developer in creating contracts. Figure 7.5 shows the first page of this wizard. It lets the user select the appropriate concerns and defines a location in the project where to store the contract. Eventually, it creates

```
<?xml version="1.0" encoding="UTF-8"?>
<ConceptualModel xmlns="http://www.infosys.tuwien.ac.at/xguide/sitemap"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://.../xguide/sitemap ConceptualModel.xsd">
  <Pages>
      <SimplePage><!-- simple page definitions go here --></SimplePage>
      <MultiPages>
          <MultiPage id="88" name="ProgrammeDetails">
              <Interface>
                  <Input><Param name="eventId" type="String"/></Input>
              </Interface>
              <LinkInformation>
                  <Target id="1" type="proxyLink"/>
              </LinkInformation>
              <References><Ref id="14"/></References>
          </MultiPage>
          <MultiPage id="149" name="Programme">
              <LinkInformation>
                  <Target id="88" type="proxyLink"/>
              </LinkInformation>
              <References>
                  <Ref id="14"/>
                  <Ref id="108"/>
                  <Ref id="114"/>
              </References>
          </MultiPage>
        <!-- more multi pages here -->
      </MultiPages>
  </Pages>
  <Components>
      <Component id="14" name="Header">
          <Interface>
              <Input><Param name="pageId" type="String"/></Input>
          </Interface>
      </Component>
      <Component id="108" name="ProgrammeSearch">
          <Interface>
              <Output name="output">
                  <Param name="date_from" type="String"/>
                  <Param name="date_to" type="String"/>
                  <Param name="keyword" type="String"/>
                  <!-- more parameters here -->
              </Output>
          </Interface>
          <LinkInformation>
              <Target id="95" type="directLink"/>
          </LinkInformation>
      </Component>
      <Component id="114" name="ProgViewDefault"/>
  </Components>
  <AppLogic>
      <Process id="95" name="ProgrammeSearchLogic">
        <!-- more process-related information here -->
      </Process>
      <!-- more application logic processes  here -->
  </AppLogic>
</ConceptualModel>
```

Figure 7.4: A fragment from the XML representation of the VIF sitemap.

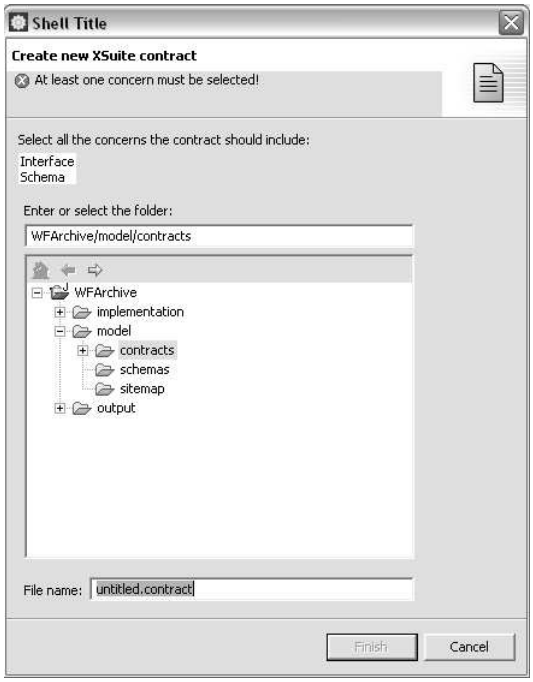


Figure 7.5: The first page of the contract creation wizard of the XSuite IDE.

a contract template from the user input and the information in the sitemap. Such a contract template is shown in Figure 7.6.

With the contract template at hand, the designers fill the contract with the corresponding information (e.g., interface specification and XML schema definition). Eventually, when all contracts are defined, the contracts can be composed according to the definitions in the sitemap. Another wizard supports the designer in doing so. First the contract that should be composed with the selected contract is chosen. Then the composition operators for all concerns in the contract are specified. If all required information is available, the contracts are composed, i.e., the appropriate component reference is inserted into the `<compositionreferences>` section of the contract. Figure 7.7 shows an input page of the contract composition wizard that requests information on how to compose the interface concerns. As discussed in Section 5.4.2, the user can choose among composition by addition, by unification, by adaptation, and by omission. Corresponding user interface controls are provided on the wizard page.

The effect of the composition operation on the XML representation of the contract (i.e., the updated `<compositionreferences>` section) is shown in Figure 7.8. A new composition reference was added that uniquely identifies the embedded contract via its name and version. Further a custom composition operator is specified for each contract concern that contains the information on how to integrate the referenced concern.

When all contracts are fully specified and all composition references satisfied, the design phase is finished. The set of contracts (including the composition information) acts as a specification for the developers. They encapsulate all information necessary to develop the various

```
<xcontract version="1.0" xmlns="http://.../xsuite/contract"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://.../xsuite/contract contract.xsd">

  <concern type="Interface" id="interfaces">
    <idl:interface xmlns:idl="http://.../xguide/concerns/interface">
      <idl:in />
      <idl:out name="output">
        <idl:param name="date_from" type="String" dimension="0" />
        <idl:param name="date_to" type="String" dimension="0" />
        <idl:param name="keyword" type="String" dimension="0" />
        <idl:param name="location" type="String" dimension="1" />
        <idl:param name="category" type="String" dimension="1" />
        <idl:param name="search" type="String" dimension="0" />
      </idl:out>
    </idl:interface>
  </concern>

  <concern type="Structure"
           docElement="ENTER_DOCELEMENT_HERE"
           id="structure">
    <xs:schema
      targetNamespace="ENTER_YOUR_TARGETNAMESPACE"
      xmlns:xx="ENTER_YOUR_TARGETNAMESPACE"
      elementFormDefault="qualified"
      xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <!-- add your schema definition here -->

    </xs:schema>
  </concern>

  <compositionreferences>
  </compositionreferences>
</xcontract>
```

Figure 7.6: The contract template generated for the programme search component with specifications for the interface and the structure concerns.

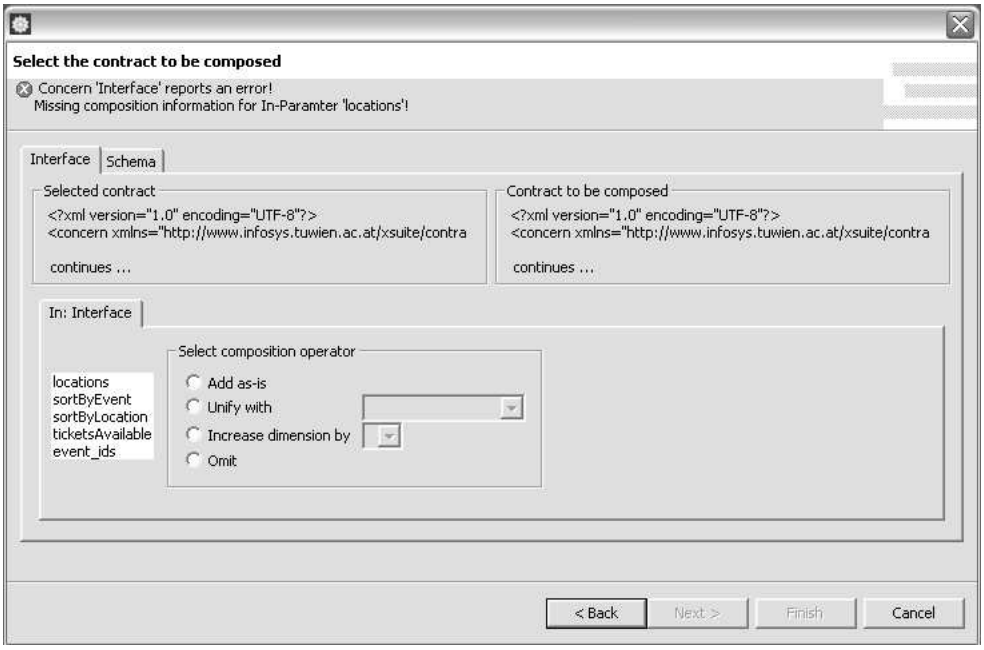


Figure 7.7: A page of the contract composition wizard requesting composition information for the interface concerns.

implementation concerns concurrently and independently of each other.

7.3.3 Discussion

The design phase depends to large parts on the expertise and experience of the designers. In the programme search example we outlined how a page can be factored into components that can subsequently be reused across multiple pages. As a result of the concrete design presented before, the initial programme overview page is merely a special case of search result, i.e., the list of events that match no specific criteria. While this design has advantages in terms of reuse and implementation effort, it makes it more difficult (if not impossible) to update the *Programme* overview page (e.g., with special offers or recent news) without also affecting the search result pages. Thus if changes to the programme overview page can be anticipated that should not be reflected on the search result pages, a better design would be to single out the overview page and model it separately (again reusing the header and search components, though).

Another aspect to keep in mind is the complexity of the model. The more components are defined, the more complex and incomprehensible the model gets. Depending on the expected change- or evolution-rate of the application, this might make sense. A mere over-engineering of the design diagram, however, (e.g., splitting all pages into components even if the components cannot be reused) only adds to the complexity of the diagram but not any value.

A good example for this kind of trade-off was the decision to model the list of events returned by a search query as a separate component (called *ProgViewDefault* in Figure 7.2). In

```
<xcontract>
  <!-- concerns go here -->

  <compositionreferences>
    <reference version="1.0" with="programmeview.contract">
      <composition type="Interface">
        <in>
          <param-ref name="locations">
            <operator type="as-is"/>
          </param-ref>
          <param-ref name="sortByEvent">
            <operator type="as-is"/>
          </param-ref>
          <param-ref name="sortByLocation">
            <operator type="as-is"/>
          </param-ref>
          <param-ref name="ticketsAvailable">
            <operator type="as-is"/>
          </param-ref>
          <param-ref name="event_ids">
            <operator type="unify" value="events"/>
          </param-ref>
        </in>
      </composition>
      <composition type="Structure">
        <operator elementName="programmeview" position="beginning"/>
      </composition>
    </reference>
  </compositionreferences>
</xcontract>
```

Figure 7.8: The updated composition reference section in the contract that contains all composition operators.

the requirements phase, we already discussed offering alternative views of the search result (e.g., sorted by event location, by date, etc.). Thus we expected other components to be used together with the existing default view. If this probable extension of the programme page would not have been obvious from the beginning, it would not have made sense to isolate the list of events in a separate component. Instead, it would have been part of the *Programme* page itself.

7.4 Concurrent Implementation based on Contracts

Once all component and page contracts exist and the composition information specifies how contracts are composed, the concurrent implementation phase starts. As explained before, the information in the contracts is everything needed to create the various parts of the implementation independently of others. Consider the contract of a dynamic page. It contains a structure and an interface concern. The structure concern (i.e., XML schema definition) defines the structure and data model. The interface concern (i.e., interface definition) specifies the input/output behavior. Thus the content manager uses the structure part of the contract to create the content documents, the graphics designer also uses the structure concern (i.e., XML schema) to create appropriate XSLT templates, and the programmer solely relies on the interface definition when implementing the application logic.

To initially create an implementation of a contract, the XSuite IDE again provides a supporting wizard. The wizard lets you select the contract to be used and lets you enter the name of the resulting XPage (i.e., implementation of a contract). Since an XPage only contains pointers to the actual implementation files, we can easily support implementation reuse by selecting existing implementation files. If no such implementation exists, the wizard creates a new implementation skeleton based on the information in the contract. Figure 7.9 shows the second page of the XPage creation wizard that provides a separate tab for every implementation concern. It further lets the developer decide on a per concern basis whether a new implementation template should be created or an existing implementation reused.

If the developer tells the wizard to create a new implementation skeleton, it analyzes the contract to create a preliminary content (XML) document, layout (XSLT) stylesheet and (Java) application logic. For this purpose, the wizard processes the structure part of the contract and creates a commented XML document conforming to the schema. Similarly, a schema-compliant XSLT stylesheet is generated. For the application logic, the situation is a little more complicated. First, a Java interface is generated based on the interface definition information in the contract. In this process, the input interface is translated into a dedicated method signature and each output interface is encapsulated in a separate sub-interface. While this interface would be sufficient to develop and compile the application logic, the programmer would not have a possibility to test the application logic until the final content and layout are available. Instead we create a factory class that returns a dummy implementation of the application logic interface until the final content and layout becomes available.

Consider the example of a *generic message* contract that defines a page to display a simple text message. The page takes a *header* and a *text* parameter as input. It does not provide output

Figure 7.9: The XPage creation wizard provides separate tabs for all concerns and the options to reuse existing implementation files.

interfaces. If the developer uses the page creation wizard to create a new implementation of this contract, the wizard creates an XPage as shown in Figure 7.10. The XPage indicates the identifier and the version of the contract it uses and contains references to implementations for the content, layout and application logic. For space reasons, we do not show the skeletons for the content and layout files here. The interface and a fragment of the created factory is shown in Figure 7.11. All this source code is directly generated from the contract using two XSLT stylesheets.

Based on the created implementation skeleton, the developers use XSuite to finish the implementation. For XML and XSLT documents, customized editors with syntax highlighting and auto-completion support the developer. For the application logic, a complete Java IDE is integrated into the XSuite environment (including editor, compiler, debugger, etc.). When all implementation activities are finished, only the creation of the page template in the `newInstance()` method of Figure 7.11 has to be changed to use the final content and layout. Then the component or page can be tested and deployed.

To demonstrate that the implementation tasks are independent of each other, we implemented the case study starting with the application logic and–in the first step–completely ignoring the content and layout concerns. With the created dummy pages, we successfully implemented the full functionality of the case study (shopping cart, programme search, archive search, etc.). Only then did we add the actual content and layout information (again independently of each other). In this step we could show that a concurrent implementation phase based exclusively on contracts is possible. The experiences we made in the various implementation tasks are collected in the discussion section below.

```
<xpage xmlns="http://www.infosys.tuwien.ac.at/xguide/xpage"
       contract="model\contracts\generic\genericMessage.contract"
       contractversion="1.0">
  <concern type="Content">
    <ref target="implementation\content\generic\genericMessage.myxml"/>
  </concern>
  <concern type="Interface">
    <ref target="implementation\logic\generated\IGenericMessage.java"/>
  </concern>
  <concern type="Layout">
    <ref target="implementation\layout\genericMessage.xsl"/>
  </concern>
</xpage>
```

Figure 7.10: The generated XPage acting as a container for references to the actual concern implementations.

After the implementation phase, the VIF case study consisted of 18 contracts that were used in a total of 25 XPages and components. 25 content files, 13 stylesheets and 118 Java classes were involved in the implementation of the pages and application logic. 93 of the total 118 Java classes were automatically generated by the XSuite IDE and the MyXML engine; the remaining 25 classes encapsulated the 'real' application logic. The number of actual HTML pages delivered to users is further increased by the fact that many pages are parameterized multi pages. For example, the event details page is parameterized with the event identifier and is used to create about 50 different HTML pages; similarly the archive details page is reused for hundreds of articles in the archive.

7.4.1 Discussion

In implementing the VIF case study, we could show how the content managers, graphics designers and programmers can work independently of each other. In practice this means that the content managers need a simple stylesheet to validate the entered content. The graphics designer work with dummy content that lets them test various formatting alternatives. The programmer, finally, uses the automatically created dummy pages mentioned before to test the input/output behavior of the application logic. In this setting, XSuite's version control support proved to be extremely useful. The integrated CVS module allowed the developers to check in their contribution to the project while the temporary files (dummy content, simple layout definitions, etc.) stayed local. As a result, the shared repository always reflected the current status of the project even if the developers used task-specific environment settings.

The development of the application logic took advantage of the integrated Java IDE. Especially the incremental compiler and debugger helped to ensure contract compliance since all sources got immediately compiled against the interfaces created from the contract. Thus the application logic–if compiled successfully–automatically is guaranteed to fulfill its contract. Also for the other concerns, the integration of the independently developed concern implementations worked without problems as long as they remained contract compliant.

```
public interface IGenericMessage {
  public void setInput(String header, String text);
  public void print(PrintWriter pw) throws Exception;
}

public class GenericMessageFactory {
  private static GenericMessageFactory factory = null;

  public static GenericMessageFactory getInstance() {
    if (factory == null) {
      factory = new GenericMessageFactory();
    }
    return factory;
  }

  public IGenericMessage newInstance() {
    return new GenericMessageDummyImpl();
  }

  public class GenericMessageDummyImpl implements IGenericMessage {

    private String header = null;
    private String text = null;
    private boolean initialized = false;

    public GenericMessageDummyImpl() {
      this.initialized = false;
    }

    public void setInput(String header, String text) {
      this.header = header;
      this.text = text;
      this.initialized = true;
    }

    public void print(PrintWriter pw) throws Exception {
      if (!initialized) {
        throw new Exception("Output class was not initialized!");
      }
      pw.println("<HTML>");
      pw.println("  <HEAD>");
      pw.println("    <TITLE>Auto-generated page from XSuite</TITLE>");
      pw.println("  </HEAD>");
      pw.println("  <BODY>");
      pw.println("    <H1>Auto-generated page for IGenericMessage</H1>");
      pw.println("    <H2>Input Parameters</H2>");
      pw.println("    <P>header (string) = " + this.header + "</P>");
      pw.println("    <P>text (string) = " + this.text + "</P>");
      pw.println("  </BODY>");
      pw.println("</HTML>");
    }
  }
}
```

Figure 7.11: The generated interface and factory of the programme search component.

Eventually it remains to be noted that in some occasions the created contracts turned out to be not complete, i.e., do not correctly specify the dependencies and interfaces between the implementation concerns. This is a result of a not careful enough design phase and, once more, shows how important it is to invest enough time and thoughts in the design. One example of an incomplete specification in the design phase is related to hidden fields in Web forms. Consider the programme search component: the *from*, *to*, *location*, *category* and *keyword* arguments are obvious since they represent form fields in which the user can enter or select values. What is easy to miss, is the existence of hidden fields in Web forms (e.g., to indicate whether the form is shown the first time, what language the form is shown in, etc.). In the design phase, such information (though important to the application logic) is easily overlooked. This is especially the case if the design is driven by design and layout mock-ups and no cross-check with the requirements of the application logic is performed. Another example of an incomplete contract turned up in the implementation of the *ShowCart* page. This time the reason was that the page uses several commit buttons in the same form (i.e., cancel order, change cart, finish order) and thus the application logic has to react differently depending on which button was pressed. The necessary information (i.e., the name of the selected button) is crucial to the application logic but was not part of our initial contract.

Besides the conclusion that we did not invest enough in the design of the application, the set of incomplete contracts gave us the possibility to try out real-world evolution scenarios as presented in Section 7.6.

7.5 Testing and Deploying the VIF case study

Because of the limited time, testing of the VIF case study before the initial roll-out started with the most critical parts of the application, i.e., the programme search and ticket ordering system. Everything else was then validated according to its importance and remaining time before the initial deployment.

Testing of the programme search and ticket ordering system basically meant to ensure that the system reacts as expected (i.e., on the client as well as the server side) for arbitrary user input. Therefore we (manually) created a set of test cases to cover the most common user scenarios, several faulty inputs and corner cases. Since the input/output behavior and the state on the server could be tested using the dummy pages and the application logic, the testing process started together with the implementation of the application logic. When the final content existed and the layout was integrated, the set of tests was executed once more against the final Web application to ensure that the integration of the content and layout did not tamper the system's behavior.

We already mentioned at the beginning of this chapter, that the design of the graphical appearance of the Web application was outsourced to an external company that produced HTML mock-ups rather than XSLT stylesheets. As a result, testing of the layout concern meant to ensure that the created XSLT stylesheets (visually) produce the same output as the original (and accepted) HTML mock-ups. Similarly, the content was re-checked in terms of correctness of information and typing errors. Eventually, a link checking tool would ensure that no broken links exist in the application.

To re-run the application logic tests and perform the content and layout checks having integrated all concerns, we needed to deploy the final application to a test environment. To deploy the case study, we needed to invoke the MyXML template engine for all XPages, compile the generated and manually coded Java sources and copy the final runtime files to a Java servlet-enabled runtime environment. For this purpose, we used XSuite's integrated Jakarta Tomcat [140] servlet engine that supports running the Web application under development directly from within the IDE. Figure 7.12 shows the Tomcat property page to manage the necessary configuration information. Everything else is done transparently for the user, i.e., the Web application description files (`web.xml`) and the server configuration file (`server.xml`) are maintained automatically. Supported by Tomcat's code hot-swapping ability, changes to the application logic even become immediately visible through the Web interface and no longer require time-consuming restarts of the servlet container.

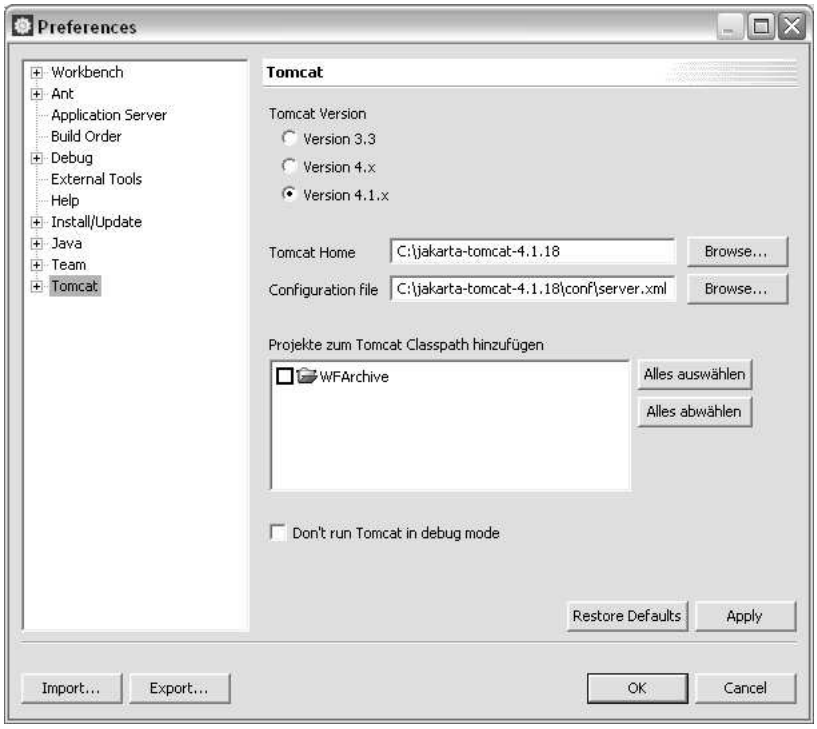


Figure 7.12: The property page of the integrated Tomcat servlet engine.

The final deployment step is to copy the Web application to the production server. Based on so-called server profiles that encapsulate all server-specific information (e.g., document root, database driver, database connect string, etc.), it suffices to copy all implementation files to the production machine and re-build the application using the corresponding server profile.

Figure 7.13 shows the final programme overview page after integrating the implementations for the content concern, the layout concern and the application logic. The common header component resides at the top of the page, the programme search component is displayed in the vertical bar on the left, and the list of events in the main area of the page.

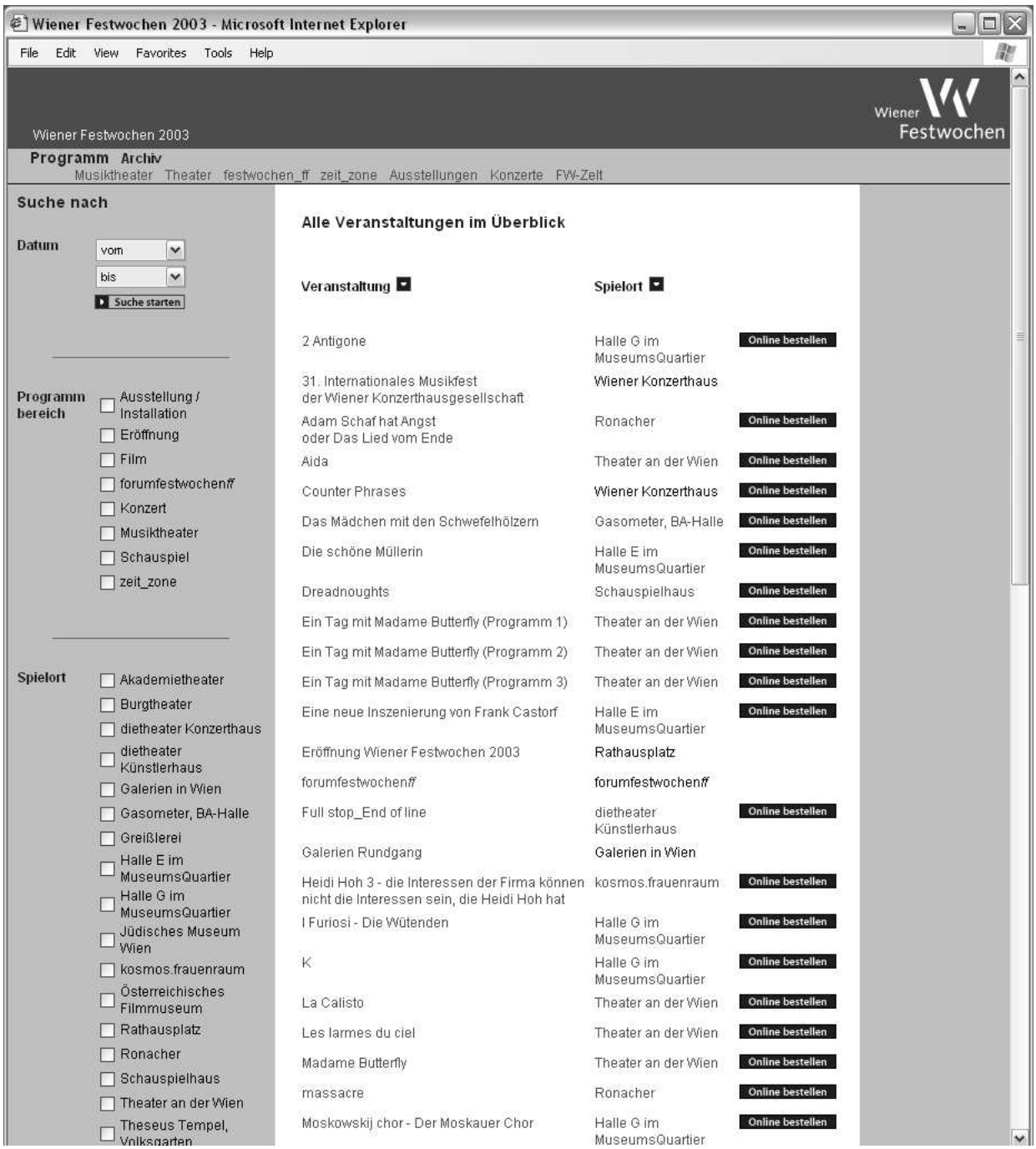


Figure 7.13: The final programme overview page of the VIF case study.

7.5.1 Discussion

The main problem when testing the VIF case study was that there is little or no work on how to exhaustively test a Web application. Even in the software engineering domain, though much research on software testing exists, testing of non-trivial software products is a difficult task– not to speak of proving the correctness of applications. Since the application logic of Web applications in most cases is non-trivial and needs to deal with transactional processes (i.e., manage user sessions), the work on software testing can be directly applied to the application logic.

For the other concerns of a Web application (e.g., content, layout, etc.) and especially their integration, it is not so clear what testing means. Furthermore, hypertext-related testing (e.g., to avoid the broken link problem or to make sure all pages are reachable) and user testing (e.g., to ensure user acceptance and usability) is important. It is also unclear to what extent the testing of Web applications can be automated since the test driver has to have a way to evaluate the server's reply. Since such replies change depending on the state of the server, the input device used, the graphical layout deployed and maybe even environmental variables (e.g., the time of the day), this is an extremely difficult task.

Also the lack of appropriate metrics to evaluate how well an application is tested (such as block coverage or arc coverage in software engineering) makes it difficult to argue about the quality of a Web application. Existing metrics mainly focus on the complexity of Web applications [45, 109].

In terms of regression testing, it is often impossible to derive from a changed resource what other resources are affected by the modification. Because of the strict separation of all concerns in XGuide, we can do a better job by maintaining dependency graphs that track all uses of any given resource. For every content or layout file, for instance, we can derive in what XPages it is used (i.e., what pages need to be rebuilt). For every XPage, we can find the contract it uses to ensure that the change does not violate the contract. Using this mechanism, we can build a list of pages that are affected by any given modification and thus need to be re-built, re-tested and re-deployed.

The deployment of the Web application itself, be it into the test environment or to the production environment of the VIF application, did not cause much problems. The concept of server profiles was expressive enough to encapsulate all machine-related information and reduce the deployment activities to copying all required files and rebuilding the application with the new profile.

7.6 Maintenance and Evolution of the VIF 2003 Web application

Once the Web application is deployed, it enters the maintenance phase. Recall that we distinguish maintenance and evolution depending on whether a modification remains within a single concern (and leaves the contract unchanged) or is cross-concern (and requires an update to the contract).

In the VIF case study, we applied maintenance in the application logic to fix programming bugs and extensively in the content concern to add new or update existing content information. Maintenance in the layout concern, i.e., changes to the graphical appearance, did not occur.

Maintaining the application logic is directly supported within the XSuite IDE by editing the source code and recompiling it. The newly generated Java class files are then automatically copied to the test environment where the code hot-swapping feature of Tomcat uses them to replace existing, now out-dated versions. As a result, changes and bug fixes in the source code are instantaneously active in the test environment once the compilation process succeeds.

Content maintenance is more complicated. The content of most Web applications is stored either in static XML/XHTML files (for static pages) or in a database (for dynamic pages). This is also the case in the VIF example. Thus content maintenance has to deal with changes to static content files and updates to database content.

In the case of static files, XSuite offers the same editing capabilities as for their creation, i.e., syntax-highlighting and auto-completing text editors. A modification of a static file is not immediately visible in the test environment since it requires a rebuild of all XPages that use the modified file. Only then, do the changes become visible.

Database content is a little different in that it does not directly affect the XGuide process. Instead it is used by the generated dynamic pages. As such, an update of database content does not require an update of any XGuide related resource. To address the recurring requirement of update interfaces for database content, we created a MyXML-based tool called *WebCUS* (Web Content Management System) [87, 95] that can be used to provide generic, customizable Web interfaces to arbitrary relational databases (as long as a JDBC connection to the database can be established). WebCUS takes the entity relationship model of the data source to manage (in XML format) and a set of access control definitions ("who is allowed to do what on which table") as input and generates a full XML/XSL-based Web application to maintain the content in the database. The functionality of WebCUS includes entering, updating, and deleting information and managing one-to-one and one-to-many relationships. Since WebCUS uses XSLT for its formatting, it can be easily adapted to fit any given corporate identity or graphical layout.

Summarizing, the maintenance activities of the case study worked well and did not cause problems. All concerns could be maintained (i.e., updated, tested and deployed) separately and integrated smoothly with existing concern implementations.

Apart from the maintenance tasks, also several evolution steps were performed. This started with the contracts that later on turned out to be incomplete (e.g., because some hidden form fields were not modeled in the interface). In general, such problems could be solved quickly by updating the contract and the corresponding XPage implementation files. To avoid inconsistencies, it is important to update the version numbers of the contract and the referencing implementations.

Another example of evolution from the VIF case study was that the ticket ordering process should not only support regular tickets but also deal with reduced tickets for children and special offers involving multiple events and/or tickets (e.g., you a get a 30 percent price reduction if you buy tickets for all events in a series). Such extended functionality was not originally considered important and needed to be integrated later on; it is a typical example of an evolution task since it obviously is cross-concern: it requires an updated content to describe special offers, updated

layout to visually indicate them and updated application logic to correctly process them (e.g., correctly calculate the grand total of an order in the presence of a special offer).

In a first step, we considered what changes to the contracts were needed. Obviously the structure concern had to reflect the additional textual information describing (potential) special offers. Further, the interface to the application logic had to be updated to include a parameter that, at runtime, would contain the actual description of the special offer (depending on the selected event and the current contents of the shopping cart). Once the updated version of the contract existed, the implementation again was done in parallel. The provision of the textual description of special offers was straightforward. Also the visual rendering (i.e., updates to the layout) were easy to do since the special offer description should simply appear in red color below the existing ticket description. Only the modifications to the application logic that needed to check for potential special offers, create the offer descriptions, and correctly calculate the overall price required significant effort. In the end, however, the support of special offers was surprisingly easy to implement and integrate.

7.6.1 Discussion

Basically, the maintenance and evolution tasks in the case study worked without problems and confirmed the advantages of contracts for maintenance as postulated in Chapter 4. Further the separation of maintenance and evolution activities better structures the post-deployment phase that is–though often costly and the longest phase–traditionally badly neglected.

For the purpose of maintenance and evolution, the XSuite IDE is not fully satisfactory. First, contract updates do not automatically result in an updated contract version. Second, modified contracts do not yet invalidate all dependent XPages and implementation files. And third, the use of CVS as version control system does not ideally support using multiple versions of the same resource.

Whenever a contract is modified, its version should be increased. There is currently no enforcing mechanism for increasing versions built into XSuite. Such a mechanism should be able to deal with the situation that the updated contract is currently not used in the project and should provide the ability to intentionally suppress the automatic update of the contract versions (e.g., to correct a typo in a contract). Once a contract is updated, all pages that use the contract should be invalidated. This forces the developers to revisit the affected pages to make sure they still comply with the contract. Such a feature becomes critical to keep track of contract compliance when more than a few versions of a contract are used. Eclipse's concepts of incremental project builders and resource change listeners offer the required functionality for its implementation. Finally, the use of CVS as version control system is not perfect to concurrently use multiple versions of the same resource in one project. CVS was designed to keep track of multiple versions of a file but usually does not support concurrent (even read-only) access to multiple versions easily.

In the VIF case study, these issues were not a real problem since we never used multiple contract versions at the same time but updated all components and pages to use the latest contract version. With the maintenance and evolution phase the VIF application enters the final stage

of its lifecycle in which it keeps returning either to the implementation phase (in the case of maintenance) or to the design phase (in the case of evolution). This loopback iteration continues for the remaining lifetime of the application, in the case of the VIF until the end of the festival in June.

In this chapter we outlined the implementation work for the VIF case study. It by far does not cover all details of the full implementation and can only show how the XSuite IDE supports the XGuide process. The discussions at the end of each section presented the strengths and open issues in the current prototype implementation of the IDE. In the final chapters we present further evaluation details of our approach and present advanced topics and open issues that need to be dealt with in future research.

CHAPTER 8

EVALUATING THE XGUIDE WEB DEVELOPMENT METHOD

If you use the original World Wide Web program,
you never see a URL or have to deal with HTML.
That was a surprise to me -
that people were prepared to painstakingly write HTML.

Tim Berners-Lee

This chapter discusses the usefulness and applicability of the XGuide development process. Evaluating a development methodology is a difficult task, all the more so if the method is new and not much experience with it exists. This chapter looks at the suggested process from various angles to present differences to and advantages over other approaches. It summarizes the experiences from the implementation of the case study and presents the most critical factors that influence the successful use of the method.

A good method to evaluate any new product or process is to calculate its rating depending on an agreed upon metrics. In computer science, many benchmarks for processors, graphics chips, hard disk, memory chips, etc. exist and help to evaluate new products. In software engineering metrics are used to evaluate the complexity of software artifacts and to estimate the length and cost of software projects. On the Web, however, such estimates or metrics are rare. Only recently a survey of Web metrics [45] shows that most existing approaches focus on aspects such as the quality or complexity of a Web site. Early work on estimating the development effort of a Web site is presented in [109] where the authors limit themselves to static Web pages and argue that merely estimating the development effort but neglecting the design and especially maintenance phases can only be a first step. As such no useful metrics to compare the XGuide process with other approaches (e.g., by evaluating them using the same Web application) exists to date.

Lacking objective metrics, another possible evaluation technique could be to run controlled experiments to evaluate how well various Web development approaches can deal with given scenarios. We apply this approach to the evaluation of the maintenance phase of XGuide. The application of this technique to the whole process is not possible since the early phases of a Web project heavily depend on the knowledge and experience of the individuals involved. Thus it is close to impossible to create a controlled and continuous environment for whole process experiments.

One possible measure to make up for the different backgrounds, knowledge and experiences of the individuals involved in the development process is to do as many experiments as possible with as homogeneous development teams as possible. We considered setting up student projects to gather feedback from as many students as possible implementing the same example project. The problem with this approach was that the prerequisites to successfully deploy the XGuide method are too high for students. They needed to know the basic XML technologies (XML, XML schema, XSLT, etc.), to understand the XGuide development process, to learn the MyXML implementation technology and to familiarize themselves with the XSuite IDE. This is beyond what a single course can teach within reasonable time.

A similar approach of evaluating a software product on as large a sample as possible is pursued by open source products. Such software is downloaded from people around the world with possibly widely varying capabilities. The main evaluation criterion in such a setting is the amount of downloads or, even better, installations of the software. Again this approach is not directly applicable to the evaluation of a development method. Successfully using a new method requires much more than downloading and installing a software package. The understanding and correct use of the development process, design models and implementation technologies is of utmost importance.

The remaining option to evaluate a development method is to implement a case study. It is obvious that the experiences gathered from a single case study are not necessarily representative for all possible kinds of Web applications but they provide a first indication of the usefulness of a method. Having presented the VIF case study in the previous chapter, the remainder of this section summarizes our experiences, outlines the prerequisites to successfully deploy XGuide and discusses in what kind of projects XGuide can best realize its potential.

8.1 Experiences from the VIF Case Study

We already gave details on the deployment of XGuide for the VIF case study in the previous chapter. Here we summarize our experiences with the central ideas in XGuide, i.e., the model driven, visual design phase, the notion of contracts as specifications for Web pages and components, the parallel implementation based on contracts, pluggable implementation technologies, and the XSuite IDE.

The visual modeling of a requirements diagram in Visio and its refinement into a high-level design diagram worked without problems for the case study. Though we introduced proxies as a way to make the diagram more readable and support splitting of diagrams into multiple pages,

for large projects these diagrams might still be overly complex to analyze and read. Especially the 'References' section of pages that lists the identifiers of embedded components is a problem. We intentionally did not add arrows or connectors to indicate such a relationship to not pollute the diagram with lines. From the list of component names, however, it is not trivial to find the referenced components in the diagram and for any given component it is hard to figure out all the places where it is used.

The introduction of contracts as a specification for pages and components forces the designers to exactly specify all the properties of the artifact. Though this is a desirable effect, it also leads to multiple evolution iterations if the designers are not experienced or not enough time is spent in the design phase. The only other problem with the definition of contracts is that the exact schema specification of the structure and the data types of all elements of a page is time consuming and requires detailed knowledge of the XML schema language. This limitation will vanish with the integration of an XML schema editor.

The contract composition operators needed in the case study could easily be covered with the operators provided for the structure and interface concerns. More sophisticated and complex operators do not seem to be a major requirement for Web applications.

An unsatisfactory experience was testing the Web application. As explained in the previous chapter we did not fully test the application; on the one hand due to lacking time, on the other hand due to a generally missing understanding of what it means to test a Web application. Standardized and well-defined testing processes for Web applications (including hypertext-related testing, software testing, acceptance testing, etc.) remain an open issue and call for future research initiatives.

We further found the visual XSuite environment and the integrated Java IDE and Tomcat servlet engine to be really helpful. This is especially important since the contract compliance of the developed application logic is automatically checked with every compilation process. We never experienced any compatibility problems at runtime (e.g., missing or superfluous parameters, etc.) as well-known from previous Web projects with CGI-based technologies. We can only emphasize the claim of [10] that tool support is critical for any new development process.

8.2 XGUIDE FOR DEVELOPMENT

XGuide's main goal in the development phase is to reduce the overall implementation time needed. Since we need to invest extra time for the definition and creation of contracts, we need to make up for it somewhere else. We thus pose the question whether XGuide really saves time–and if so–how much and in what scenarios. We do not include the time spent on requirements gathering and other analysis tasks here. Nor do we include the effort that goes into maintaining the deployed Web application; we discuss the maintenance separately in the next section. Finally, we do not include the experience and knowledge of the developers into the following considerations but assume that in both cases the developers are equipped with all the knowledge (process and technology) they need.

For the following discussion we define the *implementation delivery time* as the time period from starting the implementation phase to the initial deployment of the Web application. The *total implementation effort* of a Web project, on the other hand, is calculated from all man-hours needed to implement the Web application. Thus if the implementation phase consists of three parallel tasks that each requires 2 man-days, the delivery time would by 2 days (since all tasks are executed concurrently). The total effort, however, would be 6 man-days.

Considering traditional (i.e., sequential) Web development processes, the total delivery time is calculated from the sum of the times required to provide the layout, the content, the application logic and any other concerns. In XGuide, on the other hand, the delivery time is reduced–because of the parallel implementation phase–to the longest time required to implement any of the involved concerns. Since the design activities (e.g., the creation of contracts) is a prerequisite for the parallel implementation of concerns and does not exist in traditional processes, we also have to include it into the calculation.

The equation below shows the relationships of all the time factors involved where t_i is the time needed to implement concern i and $t_{contract}$ is the time needed to design and create the contracts.

$$\sum_{i=0}^{n} t_i \quad \gg \quad t_{contract} + max(t_1, t_2, \ldots t_n) \tag{8.1}$$

In other words, the benefit of XGuide in terms of delivery time depends on the amount of time spent for design (i.e., contract creation) and the relative differences of the t_i. If $max(t_1, t_2, \ldots t_n)$ is almost the same as $\sum t_i$ (i.e., the time to implement one concern is much greater than the times needed for all other concerns), XGuide might not be a good choice since it requires the additional $t_{contract}$. If, however, the t_i are all of similar size, a significant (delivery) time improvement can be achieved–even with the additional cost of $t_{contract}$. This is clearly expressed in the transformed formula that eliminates t_j which is the longest time for implementing a concern.

$$\sum_{i \neq j} t_i \quad \gg \quad t_{contract} \qquad where \quad t_j = max(t_1, t_2, \ldots t_n) \tag{8.2}$$

Note, however, that while XGuide's parallel implementation phase shortens the delivery time, it usually adds to the total effort since contract creation is not considered in traditional approaches. In the long run, i.e., also including maintenance and evolution activities, the benefits of using contracts should compensate for the initially increased total effort.

For the VIF 2003 case study we estimate that the creation of the content in the correct format took about 2 weeks (assuming the content is available), the design of the graphical appearance and formatting stylesheets approximately 3 weeks and the implementation of the application logic about 6 weeks (2 weeks to implement the basic functionality such as searching; another 4 weeks for the implementation of the shopping cart functionality). For the contracts we assess one to two weeks. Thus a sequential implementation phase would have taken 11 weeks (of delivery time as well as total effort). Using contracts, we could have reduced the delivery time to 7-8 weeks while increasing the total effort by the 1-2 weeks spent for contract creation. It must be

noted, however, that we explicitly did not include education effort here which–especially for the first couple of XGuide projects–can significantly contribute to the overall project duration.

Though it is great to claim a net reduction of delivery time from 11 to 7 weeks (about 36 percent), the above formula is not quite realistic. We experienced a relatively extreme case in the VIF case study but in many projects the implementation activities are not executed strictly one after the other. Often partial deliveries (e.g., graphical design templates for a couple of pages) allow for some degree of parallelism. As a result, the above formula only represents the optimal upper bound of saved time.

Further the VIF case study is not the optimal example since the content creation and layout definition activities took significantly less time than the six weeks of application logic programming. Had all tasks used the same amount of six weeks, the reduction of development time would have been from 18 to 7 (or 61 percent). It remains open, however, whether the development times of the concerns are similar or unbalanced in the 'average' Web development project. A study of many Web projects could create at least a profile for such an average project that could be used to calculate the average time benefit using XGuide.

After all these thoughts on saving time during the implementation phase, it has to be emphasized that the parallel implementation phase is just one advantage of the XGuide method. Additional benefits stem from a better communication means, explicit documentation (through contracts and design diagrams) and advantages in maintenance and evolution. The next section details on maintenance and evolution scenarios and outlines XGuide's advantages in this field.

8.3 XGUIDE FOR MAINTENANCE AND EVOLUTION

In order to evaluate XGuide with respect to maintenance and evolution, we introduce a set of scenarios of varying size, effect on the application, and complexity that we believe are typical for many Web applications. We then use different implementation approaches to highlight how each of them would manage the given scenarios. We do not pick concrete implementation technologies for this discussion but describe the properties of a set of technologies to cover multiple approaches that share similar characteristics. We distinguish three classes of technologies:

1. **Technologies that do not explicitly separate concerns** *(Class 1)*. In this group, dynamic functionality is often directly embedded into HTML pages and interpreted by a server-side module. Scripting languages such as PHP, ASP or JSP (without additional design patterns such as model-view-controller) belong in this group.

2. **Technologies that separate the application logic from content/layout templates** *(Class 2)*. This is traditionally the field of template engines. Content/layout templates are created in HTML and enriched with placeholders for dynamically generated content that is inserted at runtime. HTML++ [11], Webmacro [51] and scripting approaches that apply the model-view-controller pattern are examples for this category.

3. **Technologies that strictly separate content, layout and application logic** *(Class 3)*. Recent technologies such as Cocoon [107] or MyXML [85] try to enforce the principle of separation of concerns. They provide the most structured approach and best support reuse of artifacts. XGuide differs from other approaches in that it supports composition of pages from components and uses contracts to model dependencies between concerns.

In the following we discuss content updates, layout changes and application logic bug fixes before attacking more complex tasks such as restructuring pages, adding pages or modifying the navigation structure. For each scenario we discuss how well approaches from the above categories are suited to implement the required changes.

8.3.1 Content Updates

The most primitive content update is to simply change a typo on a single page. Independent of the approach you have to first identify the file that contains the content to be updated. This might be easier for class 1 approaches than for classes 2 and 3 since the latter involve more files that are further not directly mapped to the final structure of the Web application. The concrete update activity, however, is easiest for class 3 where the developer only has to deal with the content. In class 2, content and layout information must be dealt with. Class 1 combines all concerns into a single file and thus undesired side effects of the change can be introduced easily.

A content change across all pages of a Web site (e.g., updating an contact email address that appears in the footer of all pages), is more complicated. In principle, such an update requires an update of all pages (class 1), templates (class 2) or content files (class 3). Good support for this kind of change can only be achieved if component or fragments are supported that can be reused across all pages (e.g., HTML++, MyXML). Then only a single component needs to be updated. Page fragments are only supported in few technologies in classes 2 and 3.

8.3.2 Layout Updates

For any layout-related tasks, we do not consider the use of cascading stylesheets (CSS) [150] since they are applicable to all approaches and would not help in distinguishing them. In principle, however, we strongly encourage the use of CSS. A simple layout update could include the change of the background color or the appearance of links on all pages. Such a modification requires an update of all pages or layout templates–again with the problem of implicit modifications of the intermixed content (and potential functional definitions) in classes 1 and 2. Most class 3 approaches use XSLT to specify the graphical appearance. XSLT's concept of importing other style definitions (i.e., componentization) makes such a change almost trivial.

The situation is different if the task is to highlight a single sentence on a (single) given page. Here the replication of layout information in all pages (class 1) is an advantage since it is easy to incorporate the desired change. With approaches in classes 2 and 3 it is much harder to deal with such 'exceptions'. In this case, a new template or stylesheet has to be created which is clearly unacceptable for many such exceptions.

More serious updates include the rearrangement of information on a page and a complete change of a page's visual appearance. For class 1 approaches, this is almost impossible to implement with respect to the tightly integrated application logic and content. Class 2 approaches need to adjust all templates (and the contained content information) to the new style. In class 3, the (reusable) XSLT stylesheets are the only place where such structure and formatting information is stored. Thus only the affected XSLT stylesheets need to be updated.

8.3.3 Application Logic Updates

Updates to the application logic (e.g., bug fixes) are mainly a problem in class 1 approaches where the functionality is mixed with content and layout definitions. In these cases, unintentional modifications to the content and/or layout are easy to introduce. Since class 2 and 3 approaches strictly separate the application logic from the other concerns, functional updates are straightforward (as long as they do not affect the interfaces to the other concerns, e.g., introduce a new parameter that needs to be passed to the content template).

8.3.4 Page-Related Updates

Sometimes, larger evolution scenarios such as adding of a new page need to be implemented. Class 1 approaches require the developer to start from scratch, i.e., implement the content, the layout and the application logic of the page anew. In class 2, layout/content templates can easily be reused to create a new page as long as a suitable template already existed. Otherwise, a new template must be created. Class 3 approaches offer the most flexible alternative. Their strict separation of concerns supports reuse of arbitrary existing concern implementations (e.g., the layout or the content) and minimizes the amount of new implementation work.

Another complex evolution scenario is to introduce a new header, footer or sidebar on all pages of a Web application. With class 1 approaches this is almost impossible. It requires to edit all pages and clone the component to be added for every page. Support for this kind of change in classes 2 and 3 depends on the ability to define components and reuse implementation artifacts across pages. If, for example, a template approach supports page fragments, it is easy to implement a header fragment. Then the fragment has to be embedded into all existing templates. Without such fragment support, also class 2 and 3 do not provide significant advantages over class 1 since again the desired fragment has to be embedded into all templates or implementation files.

8.3.5 Navigation Updates

When talking about navigation updates, we distinguish two types of links: *content links* and *structure links*. Content links are embedded into the content and denote content-related resources. Structure links, on the other hand, define the navigation structure of the application and are not bound to the content.

Inserting a new content link into a page is similar to the update of the content itself. New information is hence added to the content (see above). In class 1 and 2 approaches, not only the content for the link but also its appearance need to be defined. Class 3 normally just requires extending the existing content with the new link definition. The visual appearance does not need to be specified explicitly but is reused.

Adding a link to the navigation structure of a Web application, requires to update the navigation bar on all pages of the application. In class 1, this once more requires an update of all pages. For class 2 and 3 it again depends on whether components or page fragments are supported. If fragment support is provided, the navigation structure can be modeled as a single component that is embedded into all pages. In this case, adding or modifying the navigation structure is easy.

8.3.6 New Output Formats

A final remark on maintenance refers to the ability to present content in an output format other than HTML. In a previous VIF case study, we showed how to implement access to the Web application using a cell phone and the Wireless Markup Language (WML) [148]. In a different project, we also demonstrated how to generate PDF documents that completely reflect the content of a Web site. Class 3 approaches that cleanly separate all concerns are the only candidates that can provide such a service. As soon as any layout information is encoded together with the content, support for multiple output formats is almost impossible.

Abstracting from the above scenarios, we can state that the more an approach separates concerns and supports components, page fragments, and implementation reuse, the better it is suited to deal with varying kinds of maintenance requirements. The two exceptions to this rule are the increased complexity having to maintain many more files and the non-existing support for ‘exceptions’, i.e., page or layout properties that apply to only a single page.

Similar to other class 3 approaches, XGuide exploits its strict separation of concerns to structure maintenance activities. However, it has the additional advantage of using contracts that document any changes and updates to the system, support a concurrent maintenance phase, and indicate whether other concerns are involved in a given task (evolution) or not (maintenance). Further it supports a flexible notion of Web components that support reuse not only of implementation artifacts but of whole page fragments. As a result, scenarios such as modifying the navigation structure or adding new footer component can easily be realized.

The drawback of using XGuide in the maintenance phase clearly is the overhead of maintaining and validating the contracts. This overhead, however, is not wasted but directly contributes to maintaining a well-documented and well-structured Web application.

To summarize the above sections on evaluating the XGuide process, we present a condensed view of our experiences. First, the basic idea of using contracts as design artifacts of Web applications that enable a parallel implementation phase worked to our (almost) complete satisfaction.

The prerequisites to using this approach, however, are considerable (at least at the time of this writing): developers have to be fluent in several XML languages, graphics designers are expected to model the layout in XSLT/CSS, content managers are required to create XML content, etc. To date, this is not the situation in the majority of Web projects; we hope this will change in the near future.

The requirements and design diagrams used in XGuide provide a good communication means with all the roles involved in the Web project. This is a considerable advantage over previous approaches, especially since the diagrams are directly refined into the final contracts. We showed that the implementation of contracts can be done in parallel utilizing the MyXML technology and that the strict separation of concerns facilitates a maximum of reuse potential. The support for Web components and contract composition plays a central role in reusing not only implementation concerns but whole page fragments. This is also important during maintenance and evolution.

The overall time savings from the parallel implementation phase is reduced by the additional cost of creating contracts. Though we believe based on the case study that XGuide saves a remarkable amount of time, many more experiments and Web projects are necessary to confirm this preliminary claim. The XSuite IDE as supporting software tool is important to the success of the XGuide method. It was a great help in the implementation of the case study, most remarkably the combination of a Java and Web development environment and the well integrated deployment environment represented by the Tomcat servlet engine.

CHAPTER 9

CONCLUSION AND FUTURE WORK

As an adolescent I aspired to lasting fame,
I craved factual certainty,
and I thirsted for a meaningful vision of human life -
so I became a scientist.
This is like becoming an archbishop so you can meet girls.

M. Cartmill

In this dissertation we presented *XGuide*, a novel development methodology for Web projects based on XML technology. With the progression of Web sites (i.e., pure information dissemination) into Web applications (i.e., business-critical, highly complex software applications), new requirements on technologies and development methods were introduced. Web technologies such as XML, XSL and XML Schema are the answer to new demands such as customizable content languages, reusable formatting instructions and strongly-typed documents. Though several related approaches exist (see Chapter 3), they do not fully utilize the potential offered by these technologies.

XGuide fully leverages XML technologies and provides full life-cycle support for Web projects–from the analysis to the maintenance stage. It proposes a novel notation for the design of Web applications that acts as a communication means with customers and gradually gets refined into a high-level specification of the Web application. Starting from this specification, we address another key issue in many Web projects: time-to-market. In Web projects, a short development time is often crucial to their success. XGuide accommodates this requirement by supporting a fully parallel implementation phase based on the concept of contracts. A contract strictly separates the various concerns involved in the implementation of a Web page (e.g., content, layout and application logic) and describes all dependencies among them. The contract of a page is all a developer (e.g., content manager, graphics designer, programmer, etc.) needs

to implement her aspect of the page. At the end, all implementations are tested for contract compliance and combined to form the final Web page.

The reuse potential found in a Web application is impressive: formatting rules are applied to all pages, page fragments such as headers or footers are reused across many pages, the same content is presented differently depending on the current context of the user, and so on. We introduce the notion of a Web component to represent a page fragment that can be reused in multiple pages. Just as Web pages, Web components are specified in contracts. Components are then embedded into other components or pages by composing their contracts (i.e., specifications) to form the specification of the composite page. Apart from the specification reuse in the shape of contracts, a separate page description file supports implementation reuse for all concerns. Thus the content, application logic or graphical appearance can be reused separately for arbitrary components and pages.

We implemented the *XSuite* integrated development environment to support Web development following the XGuide process. XSuite uses a model-driven approach and is based on the generic Eclipse [142] framework that offers a flexible extension mechanism to customize the environment. The XSuite IDE supports design and implementation of Web applications, provides wizards to guide the developers through contract creation, composition and implementation activities, supplies code generators, and stages a full Java development environment. XSuite further has direct support for versioning systems in order to support distributed teams and keep track of progress. The integrated Web server and servlet container using the Tomcat engine are transparently included into XSuite and facilitate immediate testing of the application.

To demonstrate its practicality, we used the XGuide method in the Vienna International Festival (VIF) case study to implement the programme, programme search, archive, and shopping cart functionality. The results of this first case study are promising in terms of clear specifications derived from visual models and based on contracts, parallel and independent implementation of all concerns, and contract-based round-trip engineering in the maintenance phase.

9.1 Analysis of this Dissertation

The first observation addresses the fundamental concept of separation of concerns. Although a broad agreement exists on the importance of this concept in the Web engineering domain, only few approaches provide continuous support for it. On the conceptual level, the actual contents, the formatting and the navigation information are frequently captured in separate concerns. XGuide takes this concept one step further in multiple respects. First, it does not operate on a fixed number of conceptual concerns but introduces an open concern model that supports dynamic adding of new concerns (e.g., application logic, meta-data, access control, etc.). Second, the dependencies of the concerns are made explicit through the contracts and the corresponding composition operators. Third, the separation of concerns is not only supported on the conceptual level but continues on the implementation level supporting concurrent development.

Several recent developments in Web engineering (e.g., template engines, separation of formatting information via XSLT and CSS) focus on realizing the reuse potential commonly found

in Web applications (e.g., content reuse, style reuse, alternative visual representations, etc.). Because of the strict separation of all concerns in XGuide, this form of reuse is fully supported. The concept of Web components further supports an orthogonal form of reuse by realizing that pages can be broken down into smaller parts, so-called page fragments. These fragments (e.g., headers, navigation structures, footers, etc.) are autonomous entities on their own and can be reused across an arbitrary number of pages. Although page fragments are implicitly used in other approaches, the novel concept of contracts in XGuide explicitly states their requirements, dependencies and external interfaces. Contract composition then explicitly defines how components can be assembled to larger components and full Web pages.

The concept of contracts is crucial to the XGuide approach. Though contracts proved to be effective in making design information more explicit and enabling parallel development, they also introduce additional complexity to the development process. Apart from learning the syntax of contracts, it often requires a major rethinking to adopt contract-based development as it requires an extended design phase, strict tests for contract-compliance, and the comprehension of contract composition rules and operators, contract versioning and update propagation. Even in software engineering, more sophisticated contracts than pure interfaces are usually avoided because of their complexity. In XGuide, the one-dimensional contracts known from software engineering are extended to multi-dimensional contracts that cover several concerns and their dependencies.

One major contribution of the XGuide process and the XSuite IDE is their openness with respect to new concerns such as meta-data or access control. We are not aware of any other approach that is able to comprise arbitrary concerns and specify their interworking with other concerns. The XGuide contract and contract composition calculus forms the basis for a semantically rich concern handling.

Another focus of this dissertation is the full life-cycle support of Web application development. Though its importance was initially pointed out by [65], few approaches respect this finding. The XGuide process starts with a requirements diagram that gets refined towards an implementation and—after the initial deployment of the Web application—introduces the dedicated maintenance and evolution phases to cover modifications and extensions of the application. XGuide's model-driven nature thereby assures that the model remains consistent with the contracts and the implementation.

Building on the survey presented in [10], we provide a prototype of a visual development environment (XSuite) supporting all phases of the XGuide process. Despite its prototype status, the IDE proved its usefulness in the implementation of the VIF case study where non-trivial contracts had to be designed, composed and implemented. Even with this preliminary experience, we concur with [10] that any reasonably complex development method will not succeed without proper tool support.

Finally, we emphasize the importance of an emerging role in the Web development process: the Web architect. Following the XGuide process, the development effort is distributed across different organizational units and roles. While this decentralization and strict separation enables concurrent development, the global view of what is going on and how it relates to other tasks in the project can easily get lost. One responsibility of the Web architect is to bear in mind the

'big picture' of all activities and to communicate it to the other roles. The architect's view of the project is initially created by guiding the development process from the business requirements to a set of well-engineered specifications (i.e., contracts in the XGuide sitemap) for components and pages. With the notion of contracts to capture the architectural and design decisions, XGuide explicitly supports this phase of the development process. With the specifications, the programmers, content managers and graphics designers can start their work. In the implementation phase, the Web architect has a mentoring role for the implementation teams to clarify the specifications and ensure that all developers maintain a consistent view on the project.

Deciding on the architecture and the design of a Web application is an important and difficult task and the resulting decisions may have far-reaching consequences. A good example is the decomposition of pages into components. While components enable a flexible architecture and can be reused, they also increase the complexity of the overall page. Failure of using components, on the other hand, results in implementing the same functionality multiple times and entails consistency problems. The advantages and disadvantages of such design decisions need to be carefully balanced to avoid undesired implications. Furthermore, a Web architect not only needs profound knowledge of conceptual modeling and system decomposition but also has to know how to choose the proper implementation technologies. This is crucial to optimally map the design to a concrete implementation, satisfy performance, scalability, or interoperability requirements and support evolution of the application. Thus a Web architect must understand the benefits and limitations of existing technologies on the conceptual level and in combination with other technologies.

9.2 Ongoing and Future Research

When a thing is done, it's done.
Don't look back.
Look forward to your next objective.

George C. Marshall

The work presented in this dissertation is the foundation for several ongoing and future extensions of the XGuide process and the XSuite IDE. One area of investigation deals with dependent concerns. Dependent concerns cannot exist on their own but directly depend on another concern. An example of a dependent concern is access control to the Web application's content that makes heavy use of XPath to denote what parts of a document should be presented to the user and what parts should remain hidden. This concern obviously is dependent on the structure concern since the XPath expressions assume the document structure defined in the structure concern.

As a consequence, the access control concern requires special attention in the context of contract composition. Basically we have to trace all modifications to the structure concern (resulting from composition operations) and adapt the access control information and composition

operators accordingly. So far it is unclear to what degree such an adaptation, if at all, can be performed automatically. Next, the effects on the implementation of the layout need to be researched. The formatting stylesheets were developed assuming a certain structure and might not work unchanged with the modified document. Imagine a user that has no access right at all; thus all content will be removed from a page. The stylesheet needs to be prepared for this situation. As a result we plan to contractually state what parts of the document might be removed because of access control limitations.

All concerns discussed in this thesis define aspects of Web components or pages. It requires a different kind of concerns, however, to characterize relationships among whole pages rather than within pages and components. We call concerns that capture dependencies among pages *site-level concerns* as opposed to the page-level concerns introduced before. A typical example for a site-level concern is the navigation concern. It contains links among pages and specifies the possible navigation paths through the Web application.

In the case of the navigation concern, the contract could include the links that must appear on all pages that reference the contract (e.g., navigation bars or hierarchical navigation menus). Our preliminary thinking is that the set of links to be included on a given page should not only be pre-determined by explicit specifications in the contract but could also be created at runtime via link queries. Link queries support the generation of link collections based on a query that operates on meta information defined in the meta data concern. A link query could, for instance, provide links to all pages that have the category 'sport' attached in the meta information. For structural links (e.g., the 'next' and 'previous' links in a slideshow), the contract also should provide a mechanism to specify what structural links a page must contain to satisfy the contract.

While the development of the access control and navigation concerns is ongoing work, the ideas on integrating device independence and meta data are longer-term. We plan to support publishing the same Web application on multiple output platforms based on the concepts of *page splitting* and *process partitioning* [90]. The goal is to support new target environments (e.g., WML on a WAP-enabled mobile phone) by only providing a set of customized stylesheets for the respective platform. The other concerns should be modified as little as possible.

Meta data aspects of Web applications are the focus of several initiatives in the Web community, most notably the W3C's semantic Web initiative [145]. Since meta data was designed from the beginning to work on top of existing Web content (i.e., with little or no interference), it is an ideal candidate for a new contract concern. Unlike device independence, the integration of the meta data concern is less a conceptual or implementation rather than a practical issue of how to provide and use such meta information.

Apart from adding new concerns to the model, several suggestions for extending the interface between the application logic and the content/layout information exist. Currently, we support displaying the information as the only operation. In some situations, other operations such as creating, updating or deleting information might also be useful. This would, of course, require a specialization of the component concept and requires us to support different operations on different kinds of components. One example could be to define a *data component* similar to or wrapping an Enterprise Java Bean (EJB) entity bean [116]. For such components, new operations such as add or delete could make sense.

But the XGuide process is not the only source for future improvement. Using the XSuite IDE for the implementation of the VIF case study revealed several interesting features that could be integrated. These features include enhanced usability, improved versioning, and better support for round-trip engineering (i.e., updates to the model should propagate to the implementation). As presented in Chapter 6, we designed the XSuite IDE to provide a plug-in interface for arbitrary implementation technologies. So far only the MyXML technology implements this interface and is thus usable in the tool suite. It is an interesting project to create a wrapper for the Apache Cocoon technology [107] that also implements the technology interface as an alternative to MyXML. The major problem for this undertaking seems to be the missing support for components and page fragments in the Cocoon platform. In the worst case, the deployment process for the Cocoon platform would have to directly integrate the components into all referencing pages.

Finally, more case studies following the XGuide process and using the XSuite IDE are necessary to strengthen the evaluation results and the experiences from the VIF case study. Well-defined student projects seem to be the most likely environment to get more experience with the XGuide process. Groups of students could implement the same Web project and report on their experiences. Such a setting would also enable the comparison of teams using the XGuide process and teams implementing in any other technology. This might also let us reason about the education overhead involved with initially deploying the XGuide process.

BIBLIOGRAPHY

[1] International Standards Organization. Standard Generalized Markup Language, 1985.

[2] About The World Wide Web. The World Wide Web Consortium. `http://www.w3.org/WWW`, 1992.

[3] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. Extensible Stylesheet Language (XSL) 1.0. Technical report, World Wide Web Consortium, Oct 2001.

[4] G. Arango. Domain analysis: From Art Form to Engineering Discipline. In *Proceedings of the 5th Internation Workshop on Software Specification and Design*, pages 152–159. ACM Press, 1989.

[5] Luciano Baresi, Franca Garzotto, and Paolo Paolini. From Web Sites to Web Applications: New Issues for Conceptual Modeling. In Stephen W. Liddle, Heinrich C. Mayr, and Bernhard Thalheim, editors, *ER 2000 Workshops on Conceptual Modeling Approaches for E-Business and the Web, volume 1921 of Lecture Notes in Computer Science*. Springer-Verlag, October 2000.

[6] Michael Barnett, Egon Boerger, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Using Abstract State Machines at Microsoft: A Case Study. In *Proceedings of the Applications of Software Measurement Conference, San Jose, CA, USA*, Mar 2000. `http://research.microsoft.com/~gurevich/Opera/145.ps`.

[7] Michael Barnett and Wolfram Schulte. Spying on Components: A Runtime Verification Technique. In *Proceedings of Workshop on Specification and Verification of Component-Based Systems at OOPSLA 2001*, 2001.

[8] Michael Barnett and Wolfram Schulte. The ABCs of Specification: AsmL, Behavior and Components. *Informatica*, 25(4), Nov 2001. `http://research.microsoft.com/foundations/comps.pdf`.

[9] Michael Barnett and Wolfram Schulte. Contracts, Components, and their Runtime Verification on the .NET Platform. Technical report, Microsoft Research, Foundations of

Software Engineering Group, Apr 2002. `http://research.microsoft.com/research/pubs/view.aspx?tr_id=555`.

[10] Chris Barry and Michael Lang. A Survey of Multimedia and Web Development Techniques and Methodology Usage. *IEEE Multimedia*, 8(2):52–60, April-June 2001.

[11] Robert Barta. What the heck is HTML++?,TUV-1841-95-06, Distributed Systems Group, Technical University of Vienna. Technical report, 1995.

[12] 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.

[13] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Declarative Information Extraction, Web Crawling and Recursive Wrapping with Lixto. In T. Eiter, W. Faber, and M. Trusczynski, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, Vienna, Austria*, volume 2173 of *Lecture Notes in Computer Science*, pages 21–42. Springer Verlag, Sep 2001.

[14] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual Web Information Extraction with Lixto. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *Proceedings of the 27th International Conference on Very Large Databases (VLDB), September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, Sep 2001.

[15] Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, 1984.

[16] F. Bell and B. J. Oates. *A framework for method integration*. Information Systems Methodologies. British Computer Society, 1994.

[17] 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.

[18] Tim Berners-Lee. The World Wide Web - Past, Present and Future. *Journal of Digital Information*, 1(1), Jul 1996. http://jodi.ecs.soton.ac.uk/Articles/v01/i01/BernersLee/.

[19] Tim Berners-Lee. Tim Berners-Lee Homepage. `http://www.w3.org/People/Berners-Lee/`, 2002.

[20] Tim Berners-Lee, R. Fielding, and H. Frystyk. Request for Comments 1945: Hypertext Transfer Protocol – HTTP/1.0. Technical report, MIT/LCS and UC Irvine, May 1996.

[21] Tim Berners-Lee, L. Masinter, and M. McCahill. Request for Comments 1738: Uniform Resource Locator (URL). Technical report, CERN and Xerox Corporation and University of Minnesota, Dec 1994.

[22] Antoine Beugnard, Jean-Marc Jzquel, Noel Plouzeau, and Damien Watkins. Making component contract aware. *IEEE Computer*, 32(7):38–45, Jul 1999.

[23] 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)*, pages 328–333. IEEE Comput. Soc. Press., Los Alamitos, CA, USA, 1996.

[24] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes. Technical report, World Wide Web Consortium, May 2001.

[25] B. Boehm. Software engineering economics. *IEEE Software Engineering*, 10(1):4–21, Jan. 1984.

[26] B. W. Boehm. Software risk management. Sep 1989.

[27] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley, 1999.

[28] Götz Botterweck. Einsatz von XML und WAP zur Realisierung strukturierter, ubiquärer Informationsdienste. Master's thesis, University of Koblenz, Germany, 2000.

[29] I. K. Bray. *An Introduction to Requirements Engineering*. Addison-Wesley, Pearson Education Limited edition, 2002.

[30] Tim Bray, Charles Frankston, and Ashok Malhotra. Document Content Description for XML: W3C Note 31 July 1998. `http://www.w3.org/TR/NOTE-dcd/`, Jul 1998.

[31] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). Technical report, World Wide Web Consortium, Oct 2000.

[32] R. Cailliau. A Little History of the World Wide Web. `http://www.w3.org/History.html`, 1995.

[33] Joao M. B. Cavalcanti and David Robertson. Synthesis of Web Sites from High Level Descriptions. In *9th WWW., 3rd Web Engineering Workshop, Amsterdam, Netherlands*, May 2000.

[34] 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*, pages 137–157. Elsevier Science B.V, May 2000.

[35] 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.

[36] F. Coda, C. Ghezzi, G. Vigna, and F. Garzotto. Towards a Software Engineering Approach to Web Site Development. In *Proc. 9th IEEE Int. Workshop on Software Specification and Design (IWSSD)*, pages 8–17, Japan, 1998.

[37] Jim Conallen. Modeling Web Application Architectures with UML. *Communications of the ACM*, October 1999.

[38] Cutter Consortium. Poor Project Management Number-One Problem of Outsourced E-Projects. Technical report, Cutter Consoritum, Research Briefs, Nov 2000. `http://www.cutter.com/research/2000/crb001107.html`.

[39] The World Wide Web Consortium. W3C Homepage, 2002. `http://www.w3.org/`.

[40] G. E. Cormack, G. Griffiths, B. D. Hebbron, M. A. Lockyer, and B. J. Oates. Web Engineering: Methods and Tools for Education. In *Proceedings of the International Conference Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet*, Jul 2002.

[41] Donald D. Cowan and Carlos J. P. Lucena. Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. *IEEE Transactions on Software Engineering*, 21(3):229–243, Mar 1995.

[42] Andrew Davidson, Matthew Fuchs, Mette Hedin, Mudita Jain, Jari Koistinen, Chris Lloyed, Murray Maloney, and Kelly Schwarzhof. Schema for Object-Oriented XML 2.0: W3C Note 30 July 1999. `http://www.w3.org/NOTE-SOX/`, Jul 1999.

[43] Steve DeRose, Ron Daniel Jr., Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. XML Pointer Language (XPointer) - W3C Working Draft. Technical report, World Wide Web Consortium, Aug 2002.

[44] Steve DeRose, Eve Maler, and David Orchard. XML Linking Language (XLink) 1.0. Technical report, World Wide Web Consortium, Jun 2001.

[45] Devanshu Dhyani, Wee Keong Ng, and Sourav S. Bhowmick. A Survey of Web Metrics. *ACM Computing Surveys*, 34(4):469–503, December 2002.

[46] Alicia Diaz, Tomas Isakowitz, Vanesa Maiorana, and Gabriel Gilabert. RMC: A Tool To Design WWW Applications. December 1995.

[47] Michael R. Donat. Automating formal specification based-testing. In Michel and Max Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Conference CAAP/FASE, volume 1214 of Lecture Notes in Computer Science*. SpringerVerlag, April 1997.

[48] Ezra Ebner, Weiguang Shao, and Wei-Tek Tsai. The Five-Module Framework for Internet Application Development. *ACM Computing Surveys*, 32(1):40, 2000.

[49] ECMA. Standard ECMA-335: Common Language Infrastructure (CLI). Technical report, ECMA - Standardizing Information and Communication Systems, Dec 2001. `http://www.ecma.ch/ecma1/STAND/ecma-335.htm`.

[50] Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2):97–111, Jun 2001.

[51] Justin Wells et al. Webmacro Home Page, `http://www.webmacro.org/`, 2002.

[52] San Murugesan et al. *Web Engineering*, chapter Web Engineering: A New Discipline for Development of Web-Based Systems, pages 3–13. Number 2016 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2001.

[53] David C. Fallside. XML Schema Part 0: Primer. Technical report, World Wide Web Consortium, May 2001.

[54] 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*, pages 414–425, June 1998.

[55] A. Frazer. *Reverse Engineering – hype, hope or here?*, volume 12 of *UNICOM Applied Information Technology*, pages 209–43. Chapman & Hall, 1992.

[56] M. Gaedke and G. Graef. Development and Evolution of Web-Applications using the WebComposition Process Model. In *Proceedings of the International Workshop on Web Engineering at the 9th International World-Wide Web Conference (WWW9), Amsterdam, The Netherlands*, May 2000.

[57] M. Gaedke, C. Segor, and H.-W. Gellersen. WCML: Paving the Way for Reuse in Object-Oriented Web Engineering. In *ACM Symposium on Applied Computing (SAC), Villa Olmo, Como, Italy*, March 2000.

[58] 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.

[59] Martin Gaedke and Jorn Rehse. Supporting Compositional Reuse in Component-Based Web Engineering. In *Proceedings of Symposium of Applied Computings (2)*, pages 927–933, 2000.

[60] Martin Gaedke, D. Schempf, and Hans-Werner Gellersen. WCML: An enabling Technology for the Reuse in object-oriented Web Engineering. In *Poster-Proceedings of the 8th International World Wide Web Conference (WWW8), Toronto, Ontario, Canada*, May 1999.

[61] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1995.

[62] F. Garzotto, P. Paolini, and D. Schwabe. HDM - A Model-based Approach to Hypermedia Application Design. *ACM Transactions on Information Systems*, 11(1):1–26, Jan 1993.

[63] Franca Garzotto, Paolo Paolini, and Luca Mainetti. Navigation Patterns in Hypermedia Databases. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, volume III, pages 370–379. IEEE Computer Society Press, January 1993.

[64] H.-W. Gellersen and M. Gaedke. Object-oriented Web application development. *IEEE Internet Computing*, 1(3):60–68, Jan-Feb 1999.

[65] 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.

[66] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Request for Comments 2616: Hypertext Transfer Protocol – HTTP/1.1. Technical report, UC Irvine and Compaq/W3C and W3C/MIT and Xerox and Microsoft, Jun 1999.

[67] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering, 2nd edition*. Prentice-Hall, Englewood Cliffs, NJ, 2002.

[68] Athula Ginige and San Murugesan. Web Engineering: An Introduction. *IEEE Multimedia, Special Issue on Web Engineering*, 8(1):14–18, March 2001.

[69] Torsten Gipp and Jürgen Ebert. Conceptual Modelling and Web Site Generation using Graph Technology. Fachberichte Informatik 4–2001, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.

[70] Jaime Gomez, Christina Cachero, and Oscar Pastor. Conceptual Modeling of Device-Independent Web Applications. *IEEE Multimedia*, 8(2):26–39, April-June 2001.

[71] Jaime Gomez, Cristina Cachero, and Oscar Pastor. Extending a Conceptual Modelling Approach to Web Application Design. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, Lecture Notes in Computer Science, pages 79–93. Sprnger-Verlag, Berlin, June 2000.

[72] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Conformance testing with abstract state machines. Technical report msr-tr-2001-97, Microsoft Research, Oct 2001. Available from `http://research.microsoft.com/pubs`.

[73] Gregory R. Gromov. History of Internet and WWW: The Roads and Crossroads of Internet History. `http://www.internetvalley.com/intval1.html`, 2002.

[74] Natacha Guell, Daniel Schwabe, and Patricia Vilain. Modeling Interactions and Navigation in Web Applications. In Stephen W. Liddle, Heinrich C. Mayr, and Bernhard Thalheim, editors, *Proceedings of ER Workshops 2000 on Conceptual Modeling Approaches for E-Business and The World Wide Web, Salt Lake City, Utah, USA*, volume 1921 of *Lecture Notes in Computer Science*, pages 115–127. Springer, Oct 2000.

[75] Frank Halasz and Mayer Schwartz. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2):30–39, February 1994.

[76] International Function Point User Group, Westerville, Ohio, USA. *Function point counting practices manual, Release 4.0*, 1994.

[77] T. Isakowitz, A. Kamis, and M. Koufaris. Extending the Capabilities of RMM: Russian Dolls and Hypertext. In *Proceedings of the Thirtieth Annual Hawaii International Conference on System Sciences*, 1996.

[78] T. Isakowitz, A. Kamis, and M. Koufaris. The Extended RMM Methodology for Web Publishing. In *Working Paper IS98 -18, Center for Research on Information Systems*, 1998.

[79] 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.

[80] A. Franzke J. Ebert. A Declarative Approach to Graph Based Modeling. In G. Tinhofer E. Mayr, G. Schmidt, editor, *Graphtheoretic Concepts in Computer Science*, number 903 in Lecture Notes in Computer Science, pages 38–50, Berlin, 1995. Springer.

[81] Michael Jackson. *Software Requirements and Specifications*. Addison Wesley Longman, 1995.

[82] D. Ross Jeffery, Graham C. Low, and M. Barnes. A Comparison of Function Point Counting Techniques. *IEEE Transactions on Software Engineering*, 19(5):529–532, May 1993.

[83] Jean-Marc Jezequel and Bertrand Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1):129–130, Jan 1997.

[84] Clemens Kerer and Engin Kirda. MyXML Home Page, `http://www.infosys.tuwien.ac.at/myxml/`, 1999-2001.

[85] Clemens Kerer and Engin Kirda. Layout, Content and Logic Separation in Web Engineering. In *9th International World Wide Web Conference, 3rd Web Engineering Workshop, Amsterdam, Netherlands, May 2000*, number 2016 in Lecture Notes in Computer Science, pages 135–147. Springer Verlag, 2001.

[86] 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, Oct 2001.

[87] Clemens Kerer, Engin Kirda, and Roman Kurmanowytsch. A Generic Content-Management Tool for Web Databases. *IEEE Internet Computing*, 6(4):38–42, July/August 2002.

[88] W. Kim. *Advanced Database Systems*. ACM Press, 1994.

[89] Engin Kirda. Web Engineering Device Independent Web Sevices. In *23rd International Conference on Software Engineering, Doctoral Symposium, Toronto, Canada*, pages 795–796, Mar 2001.

[90] Engin Kirda. *Engineering Device-Independent Web Services: An XML/XSL-based approach to creating flexible and extensible multi-device services* . PhD thesis, Technical University of Vienna, August 2002. `http://www.infosys.tuwien.ac.at/~ek/phd.pdf`.

[91] Engin Kirda, Mehdi Jazayeri, Clemens Kerer, and Markus Schranz. Experiences in Engineering Flexible Web Services. *IEEE Multimedia*, 8(1):58–65, January-March 2001.

[92] Engin Kirda and Clemens Kerer. Engineering Distributed Web Services with MyXML. Technical Report TUV-1841-2001-12, Distributed Systems Group, Technical University of Vienna, 2002.

[93] 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.

[94] Engin Kirda, Clemens Kerer, Mehdi Jazayeri, and Christopher Krügel. Supporting Multi-device Enabled Web Services: Challenges and Open Problems. In *Proceedings of the 10th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Boston, MA, USA*. IEEE Computer Society, Jun. 2001.

[95] Engin Kirda, Clemens Kerer, and Gerald Matzka. Using XML/XSL to build adaptable database interfaces for Web site content management. In *Proceedings of the XML in Software Engineering Workshop (XSE 2001), 23rd International Conference on Software Engineering (ICSE 2001)*, May 2001.

[96] R. Klapsing and G. Neumann. Applying the Resource Description Framwork to Web Engineering. In *Proceedings of the 1st International Conference on Electronic Commerce and Web Technologies: EC-Web 2000. Springer*, Lecture Notes in Computer Science. Springer Verlag, 2000.

[97] Reinhold Klapsing. Semantics in Web Engineering: Applying the Resource Description Framework. *IEEE Multimedia*, 8(2):62–68, April-June 2001.

[98] G. Kotonya and I. Sommerville. *Requirements Engineering: Processes and Techniques*. John Wiley and Sons, 1998.

[99] Klaus Kronlof, editor. *Method Integration: Concepts and Case Studies*. Wiley Series in Software-Based Systems. John Wiley and Sons, January 1993.

[100] C. Kuhnke, J. Schneeberger, and A. Turk. A schema-based approach to Web engineering. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, pages 2289–2298, Jan 2000.

[101] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. Technical report, World Wide Web Consortium, Feb 1999.

[102] Andrew Layman, Edward Jung, Eve Maler, Henry S. Thompson, Jean Paoli, John Tigue, Norbert H. Mikula, and Steve De Rose. XML-Data: W3C Note 05 January 1998. `http://www.w3.org/TR/1998/NOTE-XML-data-0105/`, Jan 1998.

[103] Gary T. Leavens. Modular specification and verification of object-orient programs. *IEEE Software*, 8(4):72–80, Jul 1991.

[104] B. Lientz and E. Swanson. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–71, 1978.

[105] F. Manola. Technologies for a Web object model. *IEEE Internet Computing*, 3(1):38–47, Jan-Feb 1999.

[106] Hermann Maurer. *Hyper-G now Hyperwave, the next generation Web solution*. Addison-Wesley England, 1996.

[107] Stefano Mazzocchi. The Cocoon 2 Project Home Page, http://xml.apache.org/cocoon/, 1999-2002.

[108] E. Mendes, N. Mosley, and S. Counsell. Web metrics - estimating design and authoring effort. *IEEE Multimedia - Special Issue on Web Engineering*, 8(1):50–57, Jan-Mar 2001.

[109] E. Mendes, N. Mosley, and S. Counsell. Comparison of Web size measures for predicting Web design and authoring effort. In *IEE Proceedings - Software*, volume 149, pages 86–92, Jun 2002.

[110] Bertrand Meyer. Lessons from the Design of the Eiffel Libraries. *Communications of the ACM*, 33(9):68–88, Sep 1990.

[111] Bertrand Meyer. Applying 'Design By Contract'. *IEEE Computer*, 25(10):40–51, Oct 1992.

[112] Bertrand Meyer. Design by Contract, Components and Debugging. *Journal of Object-Oriented Programming*, 11(8):75–79, Jan 1999.

[113] Bertrand Meyer, Jean-Marc Nerson, and Masnaobu Matsuo. EIFFEL: Object-Oriented Design for Software Engineering. In Howard K. Nichols and Dan Simpson, editors, *Proceedings of the 1st European Software Engineering Conference (ESEC)*, volume 289 of *Lecture Notes in Computer Science*, pages 221–229. Springer Verlag, Berlin, Sep 1987.

[114] Microsoft Corporation. The Microsoft .NET Web Site - `http://www.microsoft.com/net/` , 2002.

[115] Microsoft Corporation. The Microsoft Research Homepage - `http://research.microsoft.com/` , 2002.

[116] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 2000.

[117] P. Naur and B. Randell. Software Engineering: Report on a Conference by the NATO Science Commitee. Technical report, Oct 1968.

[118] J. M. Neighbors. DRACO: A Method for Engineering Reusable Software Systems. *Software Reusability*, I:295–320, 1989.

[119] T. Nguyen. LifeWeb: An Evolvable Web. `http://goanna.cs.rmit.edu.au/~tln/papers/evolution/paper.htm`, 2000.

[120] Thuy-Linh Nguyen and Heinz Schmidt. Creating and managing documents with LifeWeb. In *Proceedings of AusWeb99, Fifth Australian World Wide Web Conference, Southern Cross University*, 1999.

[121] Object Management Group. Object Constraint Language Specification Version 1.1. Technical report, Sep 1997.

[122] Object Management Group. Unified Modeling Language Specification Version 1.4. Technical report, Sep 2001.

[123] R. Prieto-Diaz. Domain Analysis for Reusability. In *Proceedings of 11th International Computer Software and Applications Conference (COMPSAC '87)*, pages 23–29, 1987.

[124] Debra J. Richardson, S. Leif-Aha, and T.O. OMalley. Specification-based Test Oracles for Reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, May 1992.

[125] Debra J. Richardson and Alexander L. Wolf. Software testing at the architectural level. In *Joint Proceedings of the SIGSOFT' 96 Worshops*. ACM Press, October 1996.

[126] T. Rollo. Sizing E-Commerce. In *Proceedings of the ACOS 2000 - Australian conference on Software measurement, Sydney, Australia*, 2000.

[127] Louis Rosenfeld and Peter Morville. *Information Architecture for the World Wide Web*. O'Reilly & Associates, February 1998.

[128] J. Rumbaugh, I. Jacobson, and G. Booch. Unified Modeling Language Reference Manual, 1999.

[129] 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.

[130] 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.

[131] Daniel Schwabe and Gustavo Rossi. An Object Oriented Approach to Web-Based Application Design. *Theory and Practice of Object Systems*, 4(4), 1998. Wiley and Sons, New York, ISSN 1074-3224.

[132] Daniel Schwabe and Gustavo Rossi. The Object-Oriented Hypermedia Design Model. *Communications of the ACM*, 38(8):45–6, August 1995.

[133] 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.

[134] Neelam Soundarajan and Benjamin Tyler. Testing components. In *Workshop on Specification and Verification of Component-Based Systems, OOPSLA 2001*, pages 1–6. Published as Iowa State Technical Report #01-09a, Oct 2001.

[135] Sun Microsystems. The JAXB Home Page, `http://java.sun.com/xml/jaxb/`, 2002.

[136] Clemens Szyperski. *Component Software, beyond object-oriented programming*. Addison-Wesley, Reading, Mass. and London, 1997.

[137] 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.

[138] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Englewood Cliffs, NJ, 2002.

[139] Toby J. 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.

[140] The Apache Foundation. Tomcat Servlet Containter - `http://jakarta.apache.org/tomcat` , 2002.

[141] The Apache Foundation. Xerces XML Parser - `http://xml.apache.org/xerces-j` , 2002.

[142] The Eclipse Consortium. The Eclipse Project Home Page, `http://www.eclipse.org/` , 2002.

[143] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. Technical report, World Wide Web Consortium, May 2001.

[144] Sayed Sajeev Thuy-Linh Nguyen, Xindong Wu. LifeWeb: An object-oriented model for the Web. In *Proceedings of SCI'98 and ISAS'98, Fourth World Multiconference on Systemics, Cybernetics and Informatics and Information Systems, Analysis and Synthesis*, volume 3, pages 301 – 308, 1998.

[145] James Hendler Tim Berners-Lee and Ora Lassila. The Semantic Web. *Scientific American*, May, 2001.

[146] W. Tracz, L. Coglianese, and P. Young. Domain-specific Software Architecture Engineering Process Guidelines. Technical report, 1992.

[147] United Kingdom Software Metric Association. *MKII function point analysis counting practices manual, version 1.3.1*, Sep 1998.

[148] WAP-Forum. Wireless Markup Language Specification `http://www.wapforum.org/what/technical.htm`. Technical report, June 1999.

[149] J. Warmer and A. Kleppe. *The Object Constraint Language*. Precise Modeling with UML. Addison-Wesley, Reading, Mass., 1998.

[150] World Wide Web Consortium. Cascading Style Sheets. Technical report, Jan 2000. `http://www.w3.org/Style/CSS/`.

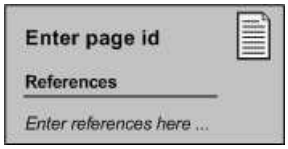
[151] M. V. Zelkowitz, A. C. Shaw, and J. D. Gannon. *Principles of software engineering and design*. PH., 1979.

APPENDIX

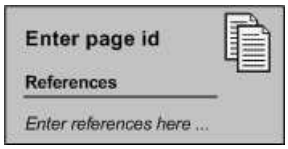
The itemization below shows the visual modeling elements used for the requirements and design diagrams in the XGuide process.

Diagram elements can incorporate a *References* and a *Description* section. The references section accepts a list of page or component identifiers and was introduced to avoid flooding the diagram with arrows that would indicate the relationship. The description of an element, on the other hand, merely provides a short statement on the purpose of the element.

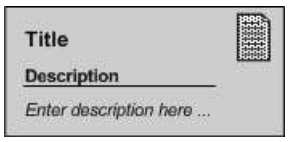
The input and output interfaces of a diagram elements (applicable only to simple, multi, and external pages, application logic processes and components) are attached properties of the element and not directly visible in the diagram.



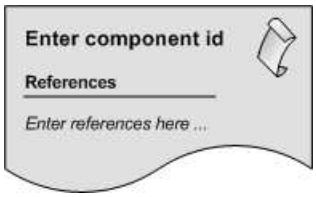
A *simple page* represents a traditional Web page. Typical examples for simple pages are homepages, sitemaps, or search pages. The simple page element is characterized by the single page icon in the upper right corner.



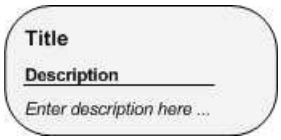
Multi pages represent a set of similar pages. Basically this means that a group of pages share common characteristics (such as layout, structure and navigational dependencies) and only differ in their content. Product catalogues as in our example Web site often use multi pages. They define a single page template and only exchange the content in this template to present all products in a consistent way. Good examples can also be found in other domains with well-structured information such as legal documents, human resources or financial information. XGuide models depict multi pages as rectangular elements with two cascaded page icons.



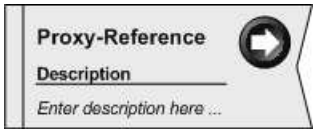
External pages (with a gray page icon) are similar to simple pages but are not included in the scope of the project. Examples for such external pages (or services) could be third-party Web sites that act as part of the Web application or legacy systems that have to be integrated. External pages have an associated short description to clarify the functionality of the external entity.



Following the component-based approach, XGuide supports so-called *Web Components* to model reuse and composition relationships. We think of a Web component as a reusable, configurable page fragment that can be reused and composed with other components to form the actual Web page. Further generalizing the Web component concept, XGuide not only supports composition of Web components into pages but also the composition of Web components into larger Web components that can then be reused as separate entities. Thus a page is a special, top-level component that cannot be further composed. Typical examples for Web components are the navigation structure of a site and a common header or footer fragment that appears on all pages.



Application logic processes model the functionality of dynamically generating Web pages and components. Application logic processes are referenced from pages or components, process the respective request (e.g., book a ticket, check the status of a reservation, or any other back-end business workflow) and produce another page as output.



A *proxy* has a unique identifier and is a representative of the element with the same identifier. It is used to keep the diagram readable, to avoid too many arrows through large parts of the diagram, and to facilitate referencing of elements if the diagram is split across several pages.

The *additional requirements* element can be added to any diagram element and indicates additional (external) requirements for that element. Such requirements are first captured on a requirements card for the respective element and eventually collected in the requirements document. Typical usage examples for this element are non-functional requirements such as response times, resource consumption, security constraints, etc.

Navigational dependencies, i.e., hyperlinks, between any two model artifacts are expressed using arrows connecting the source artifact (i.e., page or component) and the destination of the hyperlink. Such navigational dependencies do not describe what the source or destination element in a page is but state conceptually that the destination page is directly reachable from the source page.