Model-based Engineering of Web Applications: The flashWeb Method

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart
zur Erlangung der Würde
eines Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

vorgelegt von

Mihály Jakob

aus Budapest

Hauptberichter:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Mitberichter:	Prof. Dr.-Ing. Dr. h. c. Theo Härder
Tag der mündlichen Prüfung:	25.07.2011

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2011

Foreword

This dissertation was written during my work at the Institute of Parallel and Distributed Systems at the University of Stuttgart. I am grateful to everyone who have helped me, both direct and indirect, to complete this work. However, I would like to mention those individuals who have contributed the most.

First, I would like to thank Prof. Bernhard Mitschang for supervising all of my research activities at the university, for helping me to publish numerous research papers, and at last for guiding me through the process of writing this dissertation.

Next, I would like to thank Prof. Theo Härder for his interest in my research topic and for the thorough examination of this dissertation that have made it a much smoother reading experience.

Finally, I am most grateful for the support of my family during the entire process of writing this dissertation, especially my lovely wife Melanie and my mother Elisabeth, who have always been very supportive, patient and understanding.

Abstract

Since the first Web page went online in 1990, the rapid development of the World Wide Web has been continuously influencing the way we manage and acquire personal or corporate information. Nowadays, the WWW is much more than a static information medium, which it was at the beginning. The expressive power of modern programming languages is at the full disposal of Web application developers, who continue to surprise Web users with innovative applications having feature sets and complexity, which are comparable to that of traditional desktop applications. The global availability of information and applications make the WWW a very attractive medium for both private and business purposes.

Accordingly, Web application development is an important topic both in industry and research. Unfortunately, most Web application development projects are still conducted in an ad-hoc manner, without relying on a well-defined software development process. Mistakes that have been made in the early days of software development are often repeated in current day Web projects. Some experts even speak of the *Web crisis* referring to the *software crisis* of the 1970s. Therefore, it is advisable that Web projects start to employ appropriate software development processes. On the one hand, this seems like an easy task, as there already exist a multitude of process models designed for software development. On the other hand, Web applications have some unique features that make it difficult to apply a standard development process. Therefore, there is a great need for methods and tools that are specific to Web applications and can be employed for standard development processes.

Over the last decade, the Web engineering research community has produced numerous methods for Web application development. Most of these methods are model-based, i.e., they propose (graphical) models that capture different aspects of a Web application and are intended to be used in the design phase of a development process. Most methods employ three types of models that capture the application's content, navigation structure and presentational aspects. A strong focus is usually on modeling content and the application's navigational characteristics. However, as Web applications get increasingly interactive and incorporate content management features, it gets more and more important to support the modeling of content management functionality. Unfortunately, most methods treat content management operations as second-class citizens. They are usually considered only on a very abstract level and are incorporated into existing models, which produces various problems.

This work presents the flashWeb method, which fills in many gaps that are left open by existing solutions. First, the method introduces the additional Operation Model, which can

be used to explicitly specify content management operations. Operations are represented by model elements that can be combined in a flexible manner into composite operations, which represent more complex pieces of application logic. Second, the flashWeb method introduces a novel approach to combine different models by employing direct graphical connections between model elements. Finally, the method is supported by a CAWE tool that allows to create flashWeb models and to generate a fully-functional Web application from them.

Zusammenfassung

Seit der Veröffentlichung der ersten Webseite im Jahr 1990 beeinflusste die rasche Entwicklung des WWW die Art und Weise, wie wir Informationen zu persönlichen oder geschäftlichen Zwecken beziehen und verwalten. Das WWW hat sich von einem statischen Informationsmedium, was es zu Anfang war, in beträchtlichem Maße weiterentwickelt. Webapplikationsentwicklern stehen heutzutage vielfältige Möglichkeiten moderner Programmiersprachen zur Verfügung, die bezüglich Funktionalität und Komplexität herkömmlichen Desktop-Anwendungen in nichts nachstehen. Die globale Verfügbarkeit macht das WWW zu einem sehr attraktiven Informations- und Applikationsmedium sowohl für private als auch für geschäftliche Zwecke.

Es ist daher nicht allzu überraschend, dass Webanwendungsentwicklung sowohl in der Industrie als auch in der Forschung eine wichtige Rolle spielt. Es existieren viele Sprachen, Frameworks und Werkzeuge, welche für die Webanwendungsentwicklung eingesetzt werden können. Unglücklicher Weise werden viele dieser Technologien ad-hoc eingesetzt und Webanwendungsprojekten liegt meistens keine fundierte Methodik zu Grunde. Das Konzept des modellbasierten Web-Engineerings versucht, diesen Missständen ein Ende zu bereiten. Die grundsätzliche Idee hinter diesem Konzept ist die Verwendung von (graphischen) Modellen, welche für die Sammlung von Anforderungen und für den Entwurf der Webanwendung eingesetzt werden können. Nach dem Entwurf dienen die Modelle als Vorlage für die Programmierung oder als Eingabe für ein Generierungswerkzeug, das in der Lage ist, aus den Modellen eine funktionsfähige Implementierung zu generieren.

Diese Arbeit stellt die flashWeb Methode vor, die auf erprobten Konzepten des modellbasierten Web-Engineerings basiert. Die Methode führt ein neues Modell für die Erfassung von Content-Management-Operationen ein und stellt ein neues Konzept für die graphische Verbindung von verschiedenen Modellen vor. Diese Erweiterungen versetzen die flashWeb Methode in die Lage, Webanwendungen mit komplexer Content-Management-Funktionalität erstellen zu können. Des Weiteren ermöglichen die semantischen Verbindungen zwischen den verschiedenen Modellen die Generierung funktionsfähiger Webanwendungen aus den Modellen.

Modellbasierte Methoden für Webanwendungsentwicklung konzentrieren sich grundsätzlich auf die Modellierung von drei verschiedenen Aspekten. Erstens unterstützen sie die Modellierung von Inhalten. Zu diesem Zweck, wird oft ein so genanntes “Datenmodell” oder ein “Konzeptionelles Modell” eingesetzt. Diese Modelle lehnen sich oft an die Notation eines UML-Klassendiagrammes an und haben den Zweck, Objekte und Objektbeziehungen für die Datenhaltung der Webanwendung zu definieren. Der zweite Modellierungsaspekt, der oft ei-

ne Rolle spielt, beschäftigt sich mit der Navigationsstruktur der Webanwendung. Hierzu wird oft ein so genanntes "Hypertext Modell" oder ein "Navigationsmodel" verwendet. Die Navigationsstruktur einer Webanwendung spielt selbstverständlich eine wichtige Rolle, deswegen wird diesem Modelltyp in der Regel viel Aufmerksamkeit geschenkt. Das dritte Modelltyp, das oft von Web-Engineering-Methoden eingesetzt wird, ist ein so genanntes "Benutzerschnittstellenmodell" oder "Präsentationsmodell". Dieser Modelltyp hat die Aufgabe die Benutzeroberfläche der Webanwendung zu definieren.

Leider wird die Definition von Anwendungslogik von vielen Web-Engineering-Methoden vernachlässigt. Es gibt zwei grundlegende Probleme, die häufig anzutreffen sind. Zum einen werden Operationen der Anwendungslogik auf einer eher abstrakten Ebene betrachtet, so dass die Generierung einer funktionsfähigen Webanwendung, die die entsprechenden Operationen umsetzt, nicht möglich ist. Zum anderen wird die Definition von Operationen, wenn überhaupt vorhanden, in bereits existierende Modelle integriert. Modelle die hierzu zweckentfremdet werden sind oft das Datenmodell oder das Navigationsmodell.

Ein wichtiger Aspekt in der modellbasierten Webanwendungsentwicklung ist die Definition von semantischen Verbindungen zwischen verschiedenen Modellen ohne hierdurch unnötige Einschränkungen einzuführen. Ein Beispiel hierfür ist die Definition eines Datenobjektes und dessen Einbindung in das Navigationsmodell und in das Benutzerschnittstellenmodell. Die Information, die durch das Datenobjekt repräsentiert wird, soll sinnvoll in die Navigationsstruktur der Webanwendung eingebunden und dem Benutzer ansprechend präsentiert werden.

Einerseits sollen die Verbindungen zwischen den Modellen genügend Flexibilität bieten, so dass der Entwickler in der Lage ist, Webanwendungen mit beliebiger Datenstruktur, Navigationsstruktur und Benutzeroberfläche zu spezifizieren. Andererseits sollen die Verbindungen konkret mit Hilfe einer formalen Sprache ausgedrückt werden können, so dass sie eine geeignete Vorlage für Codegenerierung bieten.

Leider weisen viele Web-Engineering-Methoden hinsichtlich dieser Anforderungen Defizite auf. Eine häufig anzutreffende Schwachstelle ist ein dominantes Datenmodell, das alle anderen Modelle, wie zum Beispiel das Navigationsmodell, beeinflusst. Ist dies der Fall, hat der Webanwendungsentwickler keine Möglichkeit eine beliebige Benutzeroberfläche zu definieren, da das Navigationsmodell und das Präsentationsmodel aus dem Datenmodell abgeleitet werden müssen. Ein anderes Problem, das ebenfalls häufig auftrifft, sind lose Verknüpfungen zwischen den Modellen, zum Beispiel nur per Namensgebung, so dass eine direkte Codegenerierung aus den Modellen nicht möglich ist.

Eine wichtige Innovation dieser Arbeit ist die Vorstellung des Operationenmodells, das eine grafische Notation für die Definition von einfachen und zusammengesetzten Operationen bietet, mit deren Hilfe ein Großteil der Anwendungslogik einer Webanwendung spezifiziert werden kann. Das Operationenmodell verbindet das Datenmodell und das Benutzerschnittstellenmodell einer Webanwendung in einer einzigartigen und flexiblen Weise, so dass der Webanwendungsentwickler beliebige Inhalte und Anwendungslogik in die Benutzeroberfläche der Webanwendung integrieren kann. Diese Flexibilität bietet keine andere Web-Engineering-Methode.

Die Verwendung eines zusätzlichen Modells für die Definition von Operationen bietet viele Vorteile. Erstens ist ein Großteil der Anwendungslogik der Webanwendung in einem einzigen Modell gekapselt. Zweitens können in diesem Modell reine Datenoperationen klar von komplexerer Anwendungslogik getrennt werden. Schließlich werden Operationen nicht in andere Modelle gepackt sondern nur per Referenz eingebunden. So muss jede Operation nur ein Mal

definiert werden und kann beliebig oft zur Anwendung kommen.

Eine weitere Neuerung, die in dieser Arbeit präsentiert wird, ist die Vorstellung einer grafischen Notation für die Verbindung verschiedener Modelle einer Webanwendung. Zu diesem Zweck werden gerichtete Kanten eingesetzt, die Elemente des Operationenmodells eindeutig mit Elementen des Datenmodells und des Benutzerschnittstellenmodells verbinden. Diese Art der Modellverknüpfung bringt verschiedene Vorteile mit sich. Zum einen bieten die Verbindungen einen guten Überblick über die gesamte Anwendungslogik der Webanwendung. Zum anderen ist diese Art der Verbindung formal eindeutig, so dass aus den Modellen eine funktionsfähige Webanwendung generiert werden kann.

Der strukturierte Aufbau dieser Arbeit sorgt für eine klare Präsentation der erarbeiteten Lösungen. Kapitel 2 enthält Grundlagen rund um den Begriff *Webanwendung*. Neben einem kurzen historischen Rückblick, werden alle relevante Begriffe und Technologien vorgestellt. Kapitel 3 präsentiert die Disziplin der modellbasierten Webanwendungsentwicklung. Es werden geschichtliche Hintergründe präsentiert, die Rolle von Modellen für die Webanwendungsentwicklung erörtert, die grundsätzliche Vorgehensweise bei der Webanwendungsentwicklung erklärt und schließlich verschieden Web-Engineering-Methoden miteinander verglichen. Kapitel 4 stellt die in Rahmen dieser Arbeit entwickelte Web-Engineering-Methode namens flashWeb vor. Nach einem generellen Überblick über die Methode werden alle Modelle, die die Methode einsetzt und die Art und Weise wie die Modelle miteinander verknüpft werden können, detailliert vorgestellt. Kapitel 5 beschreibt wie die flashWeb Methode die schnelle Implementierung von Webanwendungen ermöglicht. Im Zentrum der Betrachtung steht das flashWeb CAWE (Computer-aided Web Engineering) Werkzeug, das die vollautomatische Generierung von Webanwendungen aus den Modellen der Methode ermöglicht. Das abschließende Kapitel 6 bewertet die erarbeitete Lösungen. Es werden Vor- und Nachteile der flashWeb Methode erörtert, der Vergleich mit anderen Web-Engineering-Methoden gezogen und statistische Ergebnisse der Codegenerierung präsentiert.

Contents

List of Figures

CHAPTER 1

Introduction

It is easy to find superlatives when it comes to describing the World Wide Web. Its development has been truly remarkable right from the beginning and it has transformed the way we think about acquiring information both in personal and in business life. For example, before the WWW era, planning a weekend in a foreign city or just a nice evening in one's home town was considerably more difficult and time-consuming. Traditional information sources like city guides and newspapers are not available around the clock, they cost more than a few minutes of online time and of course they lack interactivity. In contrast to that, the WWW offers us interactive maps, for example, to find a nearby movie theatre, the home page of the theatre informs us about movies that are currently playing and, in most cases, it allows us to make a reservation online. Another scenario from business life could involve a corporate employee looking for the smallest price for a certain product. Instead of acquiring different prices from possible suppliers by phone, the employee may use online catalogs to compare products and prices or may even utilize a dedicated Web portal that integrates product data from different suppliers to find the appropriate product at the best conditions automatically.

Of course, these are just simple example scenarios and the WWW provides a large variety of services and a huge amount of information just about any imaginable topic. However, the Web has been not always this versatile. In its early days, it contained only static text, which was extended over time with additional resources, e.g., images, videos, etc. It was the introduction of the first Web applications that gave a glimpse of the WWW's true potential. Nowadays, behind almost every portal on the Web, there are one or more Web applications and the WWW has evolved from a simple information medium to an application platform.

Accordingly, Web application development has become a prominent topic. There exists a myriad of languages, frameworks and tools, which can be used for Web application development. New technologies, buzzwords, and marketing terms, e.g., Web 2.0, come and go frequently, indicating the rapid evolution of the Web. Unfortunately, many of these technologies are used in an ad-hoc manner and Web application development projects are rarely based on a solid Web engineering method. The concept of Model-based Web Engineering tries to bring order to this chaos. The central idea behind the term is to employ a set of (graphical) models that can be used to capture requirements and to design the Web application. After the design phase, the models can be used as a blueprint for creating the implementation or may

be even provided as input to a code generator that is capable of producing the implementation automatically.

In this work, the flashWeb method is introduced, which integrates well-established concepts of model-based Web engineering and introduces a new model for defining content management operations as well as a novel model-weaving approach. With these extensions, the flashWeb method is capable of specifying Web applications with comprehensive content management functionality. Furthermore, the models are cohesive enough to generate a fully functional implementation from them.

1.1 Problem Domain

The most prominent development methods in the field of model-based Web engineering, the Web Modeling Language (WebML) [CFB00], the Object-Oriented Hypermedia Design Method (OOHDM) [SR98], and the UML-based Web Engineering (UWE) [BKM99] employ different graphical models to design Web applications. These and other methods usually focus on three main aspects in order to describe a Web application and provide different graphical models to capture corresponding characteristics. The first aspect concerns the modeling of the Web application's content. To this end, the methods employ a "Data Model" or a "Conceptual Model" that is usually very similar to a standard UML class diagram. The aim of this model is simply to specify objects and relationships that are used for storing Web application content. The second aspect of importance and accordingly the second model type that is provided by these methods is concerned with capturing the Web application's navigation structure. A corresponding model is usually named the "Hypertext Model" or the "Navigation Model". Hypertext is a special feature of Web applications, thus this model usually receives a lot of attention. Finally, the last model that is usually offered by a Web engineering method defines the Web application's user interface. Accordingly, it is called the "User Interface Model" or the "Presentation Model".

On the one hand, Rode comes to the conclusion that model-based Web engineering methods developed by the research community are more sophisticated than commercial products if it comes to the graphical modeling of Web applications [Rod05]. Commercial modeling tools for Web applications have evolved from traditional software tools, therefore they are good at modeling content structure and application behaviour but they are poor at dealing with navigation structures and hypertext. However, they are usually better at generating executable code for some target platform, e.g., J2EE or .NET.

On the other hand, most research approaches do not consider application behavior, i.e., content management operations and other application logic in an appropriate fashion. There are two general problems with the way such operations are handled. First, in many cases operations are considered at a very high level, thus a partial or full generation of the corresponding functionality is out of the question. A corresponding example is an operation *purchaseProduct*, which of course is far too general to be used for code generation. Purchasing a product in an online store usually includes a set of different steps that are to be modeled in more detail. Second, operations are usually integrated into one of the existing design models. For example, the WebML integrates content management operations into its hypertext model thereby complicating the model and making it more difficult to understand the actual navigation structure of the Web application.

An important aspect in model-based Web engineering is creating semantic connections between different models without introducing unnecessary restrictions. For example, if the content model of a Web engineering method defines content objects, then the navigation model and the user interface model of the same method should provide a way to represent the content object in the Web application's navigation structure and integrate it into the user interface. However, semantic connections between models should be flexible enough and allow the Web application developer to create arbitrary content structures, navigation structures, and user interfaces. On the other hand, it is desirable that semantic connections are precise enough to be expressed with a formal language, thus the models may be used as a basis for code generation. Unfortunately, most Web engineering methods have deficits regarding these aspects. A common problem is that the navigation model of a Web engineering method is usually closely related to the method's content model. This is, for example, the case for the OOHDM and the UWE methods. Both methods derive the navigation structure of the application from the content model, thus navigation nodes are basically views of content nodes, which forces the Web application designer to create a navigation structure that reflects the content structure of the Web application very closely.

1.2 Contribution

This work contributes to the research area of Web engineering in multiple ways. The main aspects of the contribution are summarized in the following listing.

Analysis of the status quo. As a prerequisite for the proposed innovations presented in this work a comprehensive and representative selection of existing Web engineering methods are analyzed and compared to each other (see Section 3.4). The comparison includes multiple aspects such as employed models, model weaving capability, and code generation. Furthermore, each of the four selected Web engineering methods are analyzed using a common example scenario, which allows a thorough and fair comparison. The result of the analysis is a set of weaknesses that are eliminated by the modeling approach presented in this work.

Introduction of the Operation Model. The main innovation of this work is the introduction of the Operation Model (see Section 4.4) that provides a graphical notation for defining basic and composite operations, which may be used to specify a large part of the Web application's business logic. The Operation Model connects the Content Model and the Composition/Navigation Model of the flashWeb method in a unique and flexible manner, thereby allowing the Web application developer to seamlessly integrate content and business logic into the Web application's user interface. With this additional model, the Web application developer is provided with a flexibility that is not offered by any other Web engineering method. The idea of the Operation Model has been published at the ICWE 2006 [JSKM06a].

Introduction of a new model weaving approach. The second important innovation that is introduced in this work is a novel model weaving approach (see Section 4.2) that allows to combine generic user interface components and operations to achieve maximal flexibility for defining the Web application. The flashWeb method provides a graphical notation (directed

edges) to connect elements of different models. This novel approach allows the Web application developer to specify the dynamic content for a user interface element quickly and in an intuitive manner. Additionally, the graphical connections provide a good overview of the Web application for the developer. Finally, flashWeb's model weaving approach defines formal connections between the models and creates a cohesive specification of the entire Web application, thereby facilitating the generation of a fully functional implementation. The idea of this novel model weaving approach has been published at the MDWE 2006 [JSKM06b].

Complete object-oriented design. An important characteristic of the proposed Web engineering method is to rely on object-oriented principles throughout the entire design process. Most existing Web engineering methods use a mixture of object-oriented ideas and concepts from relational modeling. The flashWeb method provides a cohesive object-oriented view of the Web application, which is familiar to developers, who are used to think in object-oriented terms.

Proof of concept. The last contribution of this work is a proof of concept by introducing the flashWeb CAWE tool (see Section 5.2) that is composed of a graphical editor and a code generator plugin. The editor demonstrates that it is easy to create flashWeb's graphical models and the code generator plugin creates an executable implementation of the Web application on-the-fly. The flashWeb method and the CAWE tool has been presented at ER 2007 [JSSK07].

1.3 Overview

This work provides a comprehensive overview of model-based Web application development in general and introduces the flashWeb method, which extends existing solutions with an additional model and a novel model weaving approach. This work is divided into six chapters. This first chapter provides an initial introduction to the research area of model-based Web application development, highlights some of the challenges in this area and summarizes the work's contribution.

Chapter 2 gives a detailed explanation of what Web applications are. This knowledge is of course a prerequisite for understanding how to develop Web applications, thus, it builds a solid foundation of the knowledge presented in this work. To this end, Section 2.1 gives an introduction to the early history of the WWW beginning from the idea of a hyperlinked information system at CERN to the emergence of the first search engines. Subsequently, Section 2.2 introduces Web applications by providing a set of basic definitions that are used throughout this work and also contains a categorization of Web applications. Additionally, Section 2.2.3 specifies an example Web application that is used in this work to demonstrate the features of different model-based Web engineering solutions and to compare them. Finally, Section 2.3 contains basic background information about technologies that are used to implement Web applications. To this end, different sub-sections introduce core Web technologies, different architectural patterns, and frameworks for Web application development.

The topic of Chapter 3 is model-based Web engineering, i.e., the discipline of developing Web applications using different (graphical) models. The chapter begins with a short historical overview of the Web engineering research area in Section 3.1. After that, Section 3.2 explains the role of models for software development in general and specifically for Web application

development. Section 3.3 describes the Web application development process. It explains why Web applications are different from traditional desktop applications, it introduces the Web application life cycle and provides information about different process models for Web application development. Finally, Section 3.4 introduces and evaluates four prominent Web engineering methods that have been proposed by the Web engineering research community since the mid 1990s.

The centerpiece of this work is Chapter 4, which introduces the flashWeb Web engineering method. The chapter begins with the description of the method's most important characteristics in Section 4.1, it continues with a general overview of the method in Section 4.2, before subsequent sections introduce the method's four models, namely the Content Model in Section 4.3, the Operation Model in Section 4.4, the Composition/Navigation Model in Section 4.5, and the Presentation Model in Section 4.6.

Chapter 5 describes how to implement Web applications using the flashWeb method. To this end, Section 5.1 outlines the general implementation strategy of flashWeb, which is in a nutshell the utilization of a CAWE tool to create graphical models and to generate a ready-to-run Web application from them. Section 5.2 introduces the flashWeb CAWE tool, Section 5.3 provides details about the Zope 3 Web application framework, which is the target framework of the code generator that is described in Section 5.4.

Chapter 6, the final chapter of this work, contains a detailed evaluation of the flashWeb method. Sections 6.1 and 6.2 explain the advantages and limitations of the method, respectively. Section 6.3 describes how the flashWeb method supports different process models for Web application development. Section 6.4 summarizes the differences between flashWeb and other Web engineering methods, which have been introduced in Section 3.4. Section 6.5 provides statistics about the generated Web applications. Finally, Section 6.6 outlines some enhancements and extensions of the flashWeb method as potential future work.

CHAPTER 2

Web Applications

As established in the previous chapter, the World Wide Web (WWW) has become a major factor in business and in private life. But how did it come to that and what are Web Applications after all? The first part of this chapter gives a short historical overview of the WWW's development from the very beginning of hypertext systems, through the creation of the first Web page to the emerging of large Web sites. After that, the second part deals with the evolution of Web sites into Web applications. Finally, the third major part of this chapter provides insight into the technological background of Web Applications.

2.1 Before the Age of Web Applications

The WWW is a complex information system which is implemented by a huge number of computers connected by the Internet. Therefore, the success story of the WWW is also a success story of the Internet, which is the underlying communication network. Like its predecessors, the telegraph, the telephone, and the radio, which were all invented in the 19th century [MPSS99], the primary task of the Internet is basically to transmit information. However, its protocols: the Internet Protocol (IP) [Uni81] and the Transmission Control Protocol (TCP) [CK74][CDS74] and its communication peers: PCs, Laptops and mobile phones are far more sophisticated than those of its predecessors.

According to estimates in January 2008, there were over half a billion permanently reachable computers on the Internet [URI08g] and about 1.3 billion people [URI08h] who had access to it. These figures are growing at a very high rate and they will probably double during the next ten years. Obviously, the Internet is an extremely popular communication network and the WWW is one of its most popular services.

The original task of the WWW was to provide a collection of documents to its users that were interconnected by references (hyperlinks). However, the idea of a hypertext system was invented long before the creation of the first Web page. The first significant publication regarding a hypertext system was the article "As We May Think" [Bus45] in the Atlantic Monthly by Vannevar Bush in 1945. In his publication, Bush describes a hypertext system called "Memex",

a mechanical device that allows for the efficient storage and retrieval of information using microfilm as storage medium.

In 1965, Nelson proposed a futuristic library system called "Xanadu", which should allow for the storage and publication of information in a non-linear manner. His vision was that this computer system stored all the world's information in so called *hypertext* form and that this information was freely created and shared among arbitrary individuals. Nelson was the first who used the term *hypertext*. Remarkably, today's WWW shares many common characteristics with Nelson's Xanadu.

Another visionary of the 20th century working in the field of hypertext systems was Doug Engelbart who developed a collaborative workspace called On Line System (NLS) [Eng62] [GC00], which allowed several users to work on hypertext in a parallel manner. Engelbart also invented the "Mouse", which he demonstrated together with NLS at the Joint Computer Conference in San Francisco in 1968.

Howewer, it wasn't before 1990 that the first Web page went online. Tim Berners-Lee, the inventor of the World Wide Web, started to work on the predecessor of the WWW at CERN, the European Organization for Nuclear Research [URI08c], in 1980. He called the program Enquire Within Upon Everything (Enquire) [BL00], which he used to organize people, projects, and computer programs.

He called the second version of his programm the World Wide Web, which he described in a proposal [BL89] at CERN in 1984. During the next few years he developed all cornerstone technologies of the WWW: the Hypertext Transfer Protocol (HTTP) [BL91][BLFG+99], the Uniform Resource Identifier (URI) [BL94] and the Hypertext Markup Language (HTML) [BLC93]. He programmed a simple Web server and registered it under the domain "info.cern.ch" [URI08m]. He also created the first Web client running on his NeXT personal computer. Finally, he created the very first Web page documenting the WWW project under the URL *http://info.cern.ch/hypertext/WWW/TheProject.html* [URI92] in 1990. Although, the WWW was operational in late 1990, besides Berners-Lee's initial Web pages, there was virtually no content on it and it was unknown to the public. At that time, information systems like the Wide Area Information Servers (WAIS) [PFG+94][URI03], Gopher [AML+93], or News using the Network News Transport Protocol (NNTP) [KL86] were more popular.

During the following years, several Web browsers like "Erwise", "ViolaWWW" or "Mosaic" were developed and towards the end of 1993 *info.cern.ch* registered 10.000 hits per day. Since that year the number of Web sites world-wide has been growing exponentially [URI08l]. With the number of Web sites growing fast, the need for search capability in the WWW arose. Accordingly, in 1994 two of the first major search engines entered the scene, Lycos and InfoSeek. Both had extensive indexes of the WWW and could handle search queries in a timely fashion [SM98]. However, in 1995 they were quickly superseded in terms of popularity by AltaVista, that introduced natural language queries and boolean expressions. Since then a large number of different browsers, Web servers, and search engines have been introduced and are used by millions of people all over the world.

2.2 From Web Sites to Web Applications

Since its emerging in 1990, the WWW has been developing at a tremendous rate. Web sites that started as a collection of a few pages have grown to huge portals attracting a considerable

amount of visitors. In the beginning, Web sites were merely a static collection of pages. User interactivity was restricted to simple navigation through predefined references (hyperlinks) between Web pages. However, soon enough, Web sites began to execute simple computations and to offer personalization facilities. Web sites started to provide Web forms, which allowed Web site visitors to enter text or make some simple choices, e.g., using radio buttons. The input was analyzed and the Web server generated a corresponding response. However, the Web site visitor had to input that information each time he visited the Web site. Later on, the state of user interaction was captured with so-called "Cookies", pieces of information that were stored at the client side, ensuring that personalization settings lasted until the next visit of the user. Over time, many Web sites have evolved into fully-fledged Web Applications that are capable of executing complex computations and are adaptable to user requirements in many ways. Current-day Web Applications may take into account the user's geographic location or consider the user's browser settings, e.g., adjusting the user interface to the preferred language. Additionally, modern Web portals allow the user to sign up and to create an account that may store a complete user profile and diverse user preferences permanently.

Many definitions exist for the term "Web Application" [Con03][KPRR06][LH99][Wöh04]. Some are very general, stating that everything that may be accessed through a browser is a Web Application [Wöh04]. Other definitions are more restrictive and require that the user is allowed to change the Web Application's internal state, thus according to that state change the Web Application may adapt its behavior [Con03]. Taking different definitions and the decentralized nature of the WWW into account, it is hard to determine when the first Web Applications emerged.

2.2.1 Basic Terminology

Surprisingly, several resources that deal with Web Application development fail to provide a clear basic terminology. In contrast to that, this section introduces definitions for the most important terms that are relevant for this work. The following definitions have been created using ideas from different Web engineering resources and lexicons [LH99][Con03][KPRR06][CS06] [RP06].

The WWW has started as a hypertext system. However, Web Applications incorporate more and more features of traditional software. For example, Google offers a set of *office* Web Applications [URI08d] that resemble traditional applications of an office suite, e.g., a word processor and a spreadsheet editor that run in a Web browser without prior installation. However, the vast majority of the WWW is still composed of simple hypertext and hypermedia resources, thus, it is natural to begin with the definitions of these terms.

Definition 1 *`Hypertext` is a collection of one or more text resources that include at least one "internal" or "external" reference to a text resource of the same collection. An "internal" reference is a connection from one section of a text resource to another section of the same text resource. An "external" reference is a connection from one section of a text resource to a section of another text resource.*

Note that this definition is kept as general as possible in order to avoid any dependencies from existing hypertext systems, e.g., the WWW. To this end, the term *text resource* is used instead of the term *document*. A document suggests a somewhat rigid idea of something that has a title, a text body, and perhaps some sections. Although most hypertext systems employ

the notion of a document, e.g., Web pages are implemented with HTML documents, historical [Bus45] and future hypertext systems have used and may use different concepts.

Additionally, this definition for the term "hypertext" includes the definitions of "internal" and "external" references, also called hyperlinks. This distinction is important, because a single text resource that contains a single internal reference already constitutes a hypertext.

Finally, the provided definition abstracts from any implementation aspects. There are no assumptions about storage mediums, how hyperlinks in text resources may be activated or if the implementing system is a mechanical device or a computer. Accordingly, the provided definition identifies hypertext as an *information resource* with certain characteristics. It is not to be confused with the implementing system, i.e., a hypertext system.

Definition 2 `Hypermedia` *is an amplification of hypertext. Besides simple text, hypermedia may include further resource types, e.g., images, audio, video, computer animation, etc. References between resources may originate from or target any media type.*

The difference between hypertext and hypermedia is simple. As the provided definition for hypermedia states, in contrast to hypertext, it may include additional media types. Of course, besides audio and video, text is also a media type, thus hypertext is a special type of hypermedia, using only text as medium.

As a matter of fact, most systems that employ hypertext are actually hypermedia systems. The historical Memex [Bus45] that was conceived by Vannevar Bush in 1945 was already a hypermedia system. Besides books and telecommunication records, it was also supposed to store photographs. The WWW started as a pure hypertext system. The first Web pages [URI92] created by Tim Berners-Lee in 1990 contained only text. However, as an URI may identify any media type, webmasters of the WWW's early days started very soon to reference images from their Web pages and the WWW evolved into a hypermedia system.

Note that theoretically each media type in a hypermedia system may contain outgoing and incoming references. In this context, an outgoing reference is a link that points *from* a resource that is the focus of consideration to another resource and an incoming reference is a link that points *to* a resource that is being considered. Incoming references are very common in current-day hypermedia systems (e.g. the WWW) for all media types, because it is very easy to reference any kind of resource using an URI. However, outgoing references must be supported by the corresponding media type and the hypermedia system. In case of the WWW, this is trivial for text resources, as HTML provides the *<a>...* tag for defining links. It is somewhat more difficult to provide outgoing references from images. To this end, HTML provides the *<map>...</map>* element that allows to add an interactive overlay map to an image defining different image areas that may be associated with outgoing references. However, this technology has not been adapted for videos yet. Thus, it is not possible to stop playback of a WWW video resource and click on a certain area of a video frame to be redirected to an associated resource.

The provided definition for the term "hypermedia" identifies an information resource and not the implementing system. Thus, subsequently the corresponding definition is provided.

Definition 3 *A* `Hypermedia System` *is an application that allows to create, to publish, and to retrieve hypermedia resources.*

Again, the provided definition is kept general. Although, this definition of a hypermedia system suggests, that it is some kind of an application that supports the management of hypermedia, it does not imply a specific implementation, e.g., data files on a distributed computer network. However, hypermedia systems have some common characteristics and goals (and certain ways to support them), which are elaborated in the subsequent sections.

The first goal of a hypermedia system is to support creating and modifying hypermedia content. To this end, the WWW provides HTML for the creation of Web pages and URIs for identifying different hypermedia resources. During the early days of the WWW, building Web pages was somewhat difficult. At that time, there were no HTML editors, thus Web pages had to be written using plain text editors. Thus, content creators were forced to learn the HTML syntax if they wanted to build Web pages. Although HTML is not a difficult language, only the large number of HTML editors, that are nowadays available, open up the world of Web page authoring to everyone.

The second goal that should be supported by a hypermedia system is content publishing. This aspect has become easier during the evolution of the WWW. To publish content on the WWW, one has to set up a Web server and put some Web pages on it. However, Internet service providers offer shared Web servers that may be rented and some advertisement-financed Web portals even provide Web space free of charge. Thus, virtually anyone who has access to the Internet may publish information with little effort. Additionally, more and more collaborative Web portals are being established, which allow users to contribute to mailing lists, forums and wikis without registering for a specific service.

However, easy and unsupervised access to the WWW also generates some problems. Content is created, published, and removed from the WWW at a very high rate. Thus, the quality and availability of information that is to be found is unpredictable and because of the size of the WWW the search process for a certain piece of information may be very tiresome.

Finally, the last and probably most important goal of a hypermedia system is the support for content retrieval. To this end, information should be always available, it should be easy to locate a given piece of information and support for revisiting already localized content should be provided. Unfortunately, the WWW arouses mixed feelings when it comes to judge its content retrieval capability.

On the one hand, it provides the technological support that is required to fulfill the above requirements. First, hyperlinks are easy to specify in HTML, thus they are commonly used. Second, computer technology ensures that the browsing of hyperlink structures is fast. After selecting a hyperlink on a Web page, the hyperlink's target resource is usually presented in a fraction of a second (assuming a reasonable Internet connection). Thirdly, browsers allow to store *bookmarks* to resources, thus the repeated visiting of already identified information is well supported.

Additionally, there are many services that facilitate content retrieval. The most prominent are search engines that administer indexes covering large portions of the WWW and allow the user to employ a keyword search resulting in a set of Web pages that match the users query. Another practical starting point for WWW search are catalogs that maintain hierarchical Web page indexes that may be searched or browsed manually. Finally, there exist many knowledge portals, i.e., forums, wikis, encyclopedias that may be directly consulted.

However, due to the free nature of the WWW a vast majority of its content is practically irrelevant to any certain individual. Thus, search engines, which are still the primary starting points for most search activities, deliver thousands of results that make it difficult to identify

those results that are relevant for the user. The WWW is developing at a tremendous rate not only in size but regarding technological aspects. Web sites are replaced by complex Web applications that are continuously catching up to traditional software regarding functionality. Ultimately, the WWW is currently developing into an *application space*, where the user may not only find hypermedia resources but also powerful applications that provide diverse functionality.

Definition 4 *A `Web Application` is a software system that is based on technologies of the World Wide Web and is able to process user input and to return a computed result through its user interface.*

This definition contains some aspects that should be explained in detail. First of all, a Web Application is a special kind of software system that is implemented using WWW technologies. In more detail, this statement means that a Web Application runs on a computer that is accessible via TCP/IP, which is the basic Internet protocol for data transport and HTTP, which builds on TCP/IP to allow the transport of hypermedia resources between a Web client and a Web server. Accordingly, the Web Application runs on a Web server that potentially employs an application server, it is identifiable by an URI, and accessible through a Web browser.

The usage of WWW technologies is of course an essential characteristic of Web applications. A definition in one of the numerous books about Web engineering [KPRR06] states that Web Applications are "based on standards of the World Wide Web Consortium (W3C)". Although the W3C is a prominent authority that supervises the development of many key WWW technologies, it is not advisable to restrict Web Applications to this technology subset. On the one hand, there are many technologies, e.g., JavaScript that are commonly used for Web Application development and are not supervised by the W3C, on the other hand, there exist numerous W3C technologies that may be useful but are not necessary to create Web Applications. Although the term "WWW technologies" is not overly precise, it is more suitable to characterize the required set of technologies for Web Application development.

Also, the utilization of WWW technologies for application development does not mean that a Web Application must run on the WWW. A Web server and a Web application may be set up on any given computer that does not have to be connected to the Internet. Correspondingly, a Web Application is a software system *for* the WWW but not necessarily *on* the WWW.

Another definition states that a Web Application is an application that may be accessed through a Web browser [Wöh04]. This is of course a very simplistic and dubious definition. On the one hand, most browsers incorporate features that, for example, allow the user to browse the local file system using the *file://* naming scheme. Thus, most browsers are capable of opening a simple text document that does not include any HTML code and is obviously not a Web Application. On the other hand, in Web engineering research, it is customary to make a clear distinction between *Web sites* and *Web applications*.

According to the definition that is provided in this work, a Web Application is a system that accepts user input, processes it, and returns some kind of an output. The simple activation of hyperlinks (browsing) on Web pages does not account as user input and the redirecting of the user to the target resource of a hyperlink should not be regarded as data processing. Therefore, a Web site, which is a simple collection of hypermedia resources is not a Web Application.

The definition of Conallen [Con03] is based exactly on this distinction. He argues that the user of a Web Application must be allowed to change the Web Application's internal state so that this internal change may determine its behaviour. He also states that according to his definition search engines are not to be considered as Web Applications. He points out that after

the user of a search engine submits a query and receives a search result, the internal state of the search engine stays unchanged. Obviously, Conallen's definition requires the Web Application to be able to store a change of state permanently.

The definition used in this work is a little more liberal. As already discussed, it is required that the user is allowed to provide input that is processed and that the result of this processing is returned to the user. However, an internal state change is not necessary. Accordingly, a search engine that accepts user input, computes search results, and provides them to a user is regarded to be a Web Application.

2.2.2 Categorization

Today the WWW offers a large number of Web Applications that are employed for different purposes. Accordingly, Web Applications are developed to achieve various goals, they operate in different application domains and have varying complexity. Web Applications have evolved from Web sites, thus in most cases, they also exhibit many characteristics typical for Web sites. The following subsections provide different approaches for the categorization of Web applications that have been derived from various resources [Pow98][LH99][KPRR06][BGP00].

2.2.2.1 Development History

A straightforward approach is to track the development of Web Applications by time and complexity as Figure 2.1 (see [KPRR06]) demonstrates.

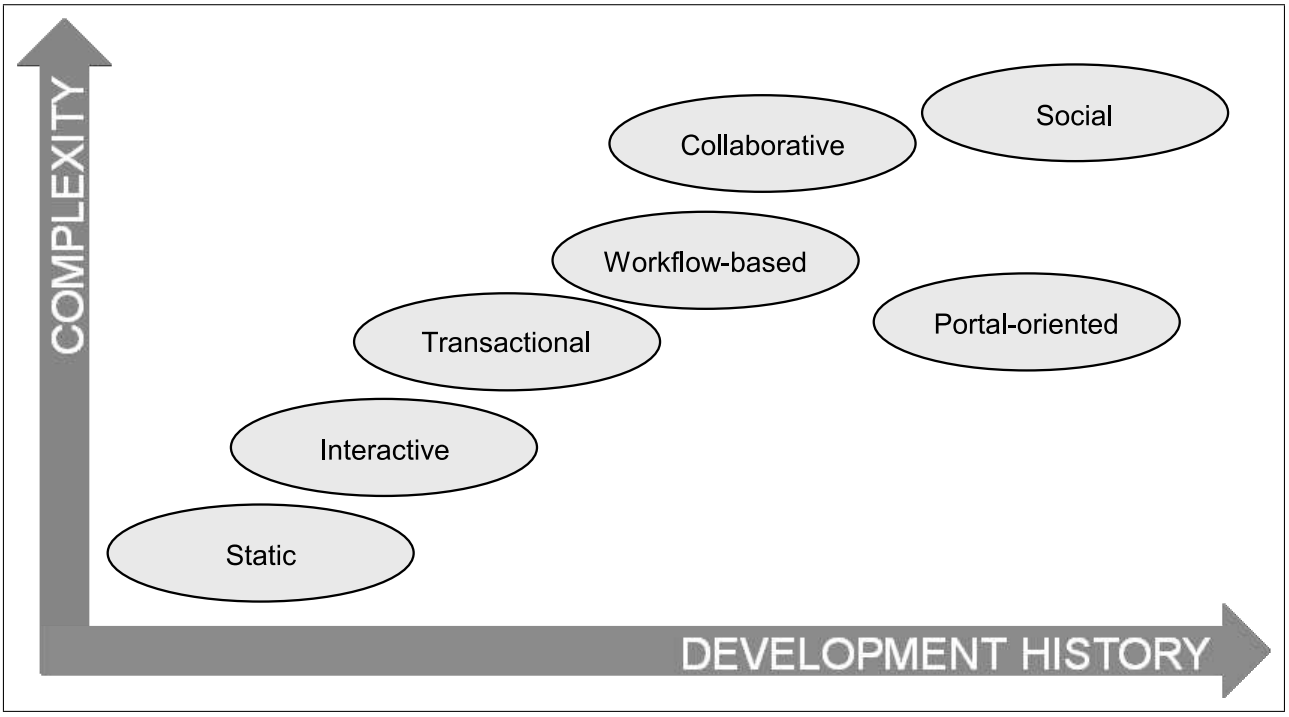

Figure 2.1: Web Application Categories

Static Web sites were the predecessors of Web Applications. Besides standard navigation capability, i.e., the user was able to activate hyperlinks, they did not offer any further interactive features. *Interactive* Web Applications appeared as Web sites began to incorporate interactive

user interface elements. The user was allowed the first time to input data via Web forms and the Web server generated corresponding response documents via the Common Gateway Interface (CGI) technology. However, these first Web applications allowed a rather small degree of interaction, because access to the Web Application's content was read-only. As the need for the management of large data volumes arose, i.e., the first online shops and online reservation systems appeared, Web Applications started to employ database management systems for backend storage. These *transactional* Web Applications provided the first time write access to the Web Application's content. Additionally, database technology provided a consistent data storage that was safe to operate in a multi-user scenario. *Workflow-based* Web Applications employ workflow technology and are to be splitted into two categories. First, Web Applications may utilize an internal workflow engine for process management. In this case, the Web Application is really workflow-based as workflow technology is directly employed to achieve goals. Second, the Web Application may be participating in a global workflow, acting as a single service, e.g., in a Web Services scenario. *Collaborative* Web Applications are usually employed if several individuals or groups are supposed to create, manage, and share content. In this category, the subject of collaboration may vary. Groupware applications focus on document management, email, and shared appointment scheduling and are usually employed for project coordination and management. In contrast to that, forums, wikis, and chatrooms are more generic in nature and may be used for general information exchange without a particular focus. A current trend in the WWW goes towards the sharing of personal information. This trend is supported by so called *social* Web Applications, which facilitate the building of social networks, e.g., student or alumni portals. Another example are Weblogs that many individuals use to regularly publish personal experiences and opinions on diverse topics. Finally, *portal-based* Web Applications support the building of central access points and online communities by providing key communication features, e.g., user accounts, events, news, etc.

Of course, there are many Web Applications that belong to more than one of the introduced categories. For example, social Web Applications are often collaborative in nature and are often implemented as a Web portal. Additionally, they may utilize workflow technology or a database management system.

2.2.2.2 Document-centric vs. Application-centric

Web sites are composed of a number of Web pages that are usually implemented as HTML documents. An HTML document may include different hypermedia resources, e.g., images, videos, etc. In contrast to the early days of the WWW, these resources are usually directly embedded into the document, and most Web browsers are capable of rendering an integrated document view that may, for example, display text, images, and video playback at the same time. A *document* represents a basic concept regarding the structure of hypermedia content. Therefore, Web sites are to be considered *document-centric*.

The first Web Applications appeared as Web sites started to incorporate features that are usually only provided by traditional software applications. The user was allowed to input some data using simple HTML forms and the Web Application responded correspondingly. Thus the first Web Applications were actually more or less Web sites with some application functionality, and therefore still to a great extent document-centric.

However, nowadays WWW technologies have developed so far that some Web applications are difficult to tell apart from traditional desktop applications. This development have been

boosted especially through the introduction of the AJAX [URI05] technology set. The main achievement of AJAX is the introduction of asynchronous communication between Web client and Web server that allows for a much faster response of the Web Application's user interface and ultimately a *look & feel* that resembles traditional desktop applications. A good example for this category of Web Applications is Google's Web office suite [URI08d]. It provides a set of applications that do not rely on the concept of Web pages to implement application logic. Therefore, these applications are regarded *application-centric*.

2.2.2.3 Application Domains

Web Applications are omnipresent in the WWW. No matter what kind of information one needs, the WWW probably has the answer and, in many cases, there exist a specialized Web Application that helps to find or process the information in an efficient way. Thus, it is obvious that Web Applications may be also categorized by the application domain. The following paragraphs describe some of the most important domains.

One of the more prominent Web Application domains is e-business. In this domain, Web Applications are employed for business-to-business (B2B), for business-to-consumer (B2C), or for consumer-to-consumer (C2C) purposes. In the B2B area, enterprises operate online product catalogs, ordering and logistic systems that may be used by other companies to plan or conduct business transactions. In the B2C area, enterprises often offer their products in online shops to end customers. Finally, in the C2C area, many online portals allow private individuals to rent, buy, or sell virtually any kind of product from or to other individuals.

Another large application domain is education. Governments, universities, schools, and other educational institutions of many nations offer a wide range of portals that provide educational content. Besides presenting many hypermedia resources, these portals employ a variety of Web Applications that target different audiences. The range of applications spans from small demonstrations for pupils to complex online exam systems for students and graduates.

A further popular application domain is entertainment. The main application providers in this area are information portals like Yahoo!, AOL, and MSN. The application range is remarkable. Besides traditional board and card games, portal visitors may choose from a variety of modern video games from diverse areas, e.g., sports, adventure, or role playing.

Last but not least, Web Applications are often used in the area of informational collaboration. For any given topic there exists a number of forums, wikis, and chat rooms, where participants may exchange ideas about their favorite topics. Additionally, over the past few years a number of general information portals and encyclopedias have been established that are used by the public to share information freely. Examples are Wikipedia [URI08n], which is the largest online encyclopedia containing about 10 million articles in over 250 languages at the time of writing and Flickr [URI08f] a photo-sharing portal that allows geographical tagging of photos having two billion photos and and an upload volume of 4-5 million photos per day.

2.2.2.4 Content Management Capability

All previous categorizations are useful to get a good overview about specific characteristics, employed technologies, and application domains of Web applications. However, Web engineering methods should not support a specific technology or application domain exclusively. They must remain generic in nature so that the design and architecture of the Web Application remains independent from implementing technologies and a potential application domain.

Therefore, in this section, a further differentiation is introduced that classifies Web applications regarding their content management capability. This classification is used in Section 3.3 to differentiate between existing Web engineering methods and the approach presented in this work.

Static Web Sites. A Web site is a collection of Web pages that contain hypermedia resources. Its only purpose is to provide information to the Web site's user. The presented information may be traversed using the hyperlink structure of the Web site, and besides activating hyperlinks the user is not able to interact with the site. Especially, the Web site does not provide Web forms that allow the user to input data. Thus, the Web site's content is completely static and it does not react to user behaviour in any way.

Note that according to Definition 4, a static Web site is of course not a Web Application. This category is still introduced here as a contrast to subsequent categories that do define Web applications. This category is also useful for categorizing Web engineering methods introduced in Section 3.3.

An example for this category is a portal that provides information about books. It may contain several indexes that list books by title, category, or author, and for each book a separate page may exist that describes the book. Additionally, for each author an author page may exist including a short biography. The user of the book portal may use the indexes to find a page that describes the book of his choice. From the book page he may browse back to an index or to the page of the book's author. All information of the book portal is static and the only way for the user to interact with the portal is to browse the provided content.

Simple Web Applications. In contrast to a Web site, a Web Application allows for user interaction. Besides the presentation of static content, a Web Application that belongs to this category provides user interface components that may be used by the user to influence the Web Application's behaviour. Typical user interface components for this purpose are links, buttons, and text entry fields. The provided user input, i.e., text that is entered in a form or the activation of a particular link or button, is interpreted by the Web Application and a corresponding response is generated. However, the user's input is not stored and it does not modify the Web application's content in any way.

The previous book portal example may be extended from a Web site to a simple Web application by providing some components for user interaction. For example, the portal may include a search form that allows the user to search for books or authors. To this end, the user may enter some keywords into the search form and submit the query. The book portal then may analyze the query and return some results. However, the user's interaction is still limited as he is not allowed to modify the Web Applications content. For example, it would be nice if the user would be able to write a comment about a book he already read and provide this information to other users.

Web Applications for Content Management. Web Applications that support content management functionality belong to this category. Content management operations (CRUD) allow the Creation, the Retrieval, the Update, and the Deletion of content. Web Applications in this category are usually more interactive, because the users input may be stored permanently and can be used for personalization purposes in the same or future work sessions. Additionally,

only Web Applications, which support CRUD operations, may be used for collaboration purposes. The permanent storage of information is a prerequisite for collaboration as the input of one user must be stored if it should be presented to another user later on.

Accordingly, the book portal application that supports basic content management may allow the user to write book reviews, which may be stored on the portal server permanently. Additionally, the book portal may allow a user to manage his own reviews, i.e., to change or to delete those reviews, which have been created by the user. Eventually, the user is allowed to create, modify, and delete content of the book portal.

2.2.3 Example Application - The Book Portal

This section specifies the *Book Portal*, an example Web Application that is used throughout the entire work for different purposes. First, in Section 3.4, a number of existing Web engineering methods are described. In order to facilitate the comparison of these methods, all of them are illustrated using the Book Portal example. Second, the description of flashWeb models and the demonstration of the code generation capability of the approach are based on this example. On the one hand, this general example ensures the consistent presentation of the flashWeb approach, on the other hand, it allows to compare it to other Web engineering methods.

The Book Portal example Web Application supports a basic subset of the functionality that is usually supported by a real-world online book store. The focus is rather on functionality than size of the example. It shows information about several objects, e.g., books and authors, it provides common navigational access structures like menus and indexes and it supports content management.

2.2.3.1 Content

The main purpose of the Book Portal example Web Application is to provide information about books. Each book record of the portal provides some basic information, e.g., the book's title, a short description, or the number of the book's pages. Books are organized by a hierarchical category system. Each book may be assigned to one or more categories, which may have an arbitrary number of subcategories. Additionally, the Book Portal provides information about book authors. An author record contains the author's first names, last name, and a short biography. For each book the portal stores the complete list of authors. The Book Portal also manages portal users and allows them to create book reviews. A review record contains a title, the review text and a score, which indicates the users opinion of the book. Finally, a user of the Book Portal is allowed to fully manage his reviews. Thus, he may delete an already written review or assign the review to another book. The assignment date is stored, thus it is clear when a certain review for a certain publication was issued. Table 2.1 provides an overview of objects that are relevant for this example.

2.2.3.2 Navigation Structure

After specifying content requirements for the Book Portal, this section outlines the portal's navigation structure. To this end, user interface pages and navigation paths between pages are sketched in Figure 2.2.

The `Portal` page is the main entry point of the Book Portal application. It contains the main menu that has entries pointing to the `Categories`, the `Authors`, and the `Search` pages. Note

Book	Title	The title of the book.
	Abstract	A short description of the book's content.
	Pages	The number of the book's pages.
	ISBN	The International Standard Book Number.
Category	Name	The name of the category.
Author	FirstNames	The author's first names.
	LastName	The author's last name.
	Biography	The author's biography.
Review	Title	The title of the review.
	Text	The review text.
	Score	A number between one and five indicating the quality of the book.
User	FirstNames	The user's first names.
	LastName	The user's last name.
	Nickname	An alias that is shown to other users of the portal.
	Email	The user's email address.

Table 2.1: Book Portal Types

that each page of the Book Portal contains the main menu, thus it is possible to navigate from them to these main areas. However, corresponding navigation arrows, e.g., starting from the `Portal` page and pointing to the `Categories`, the `Authors` and the `Search` pages are not repeated for each page in Figure 2.2.

The `Categories` page provides a list of books for the selected category and allows the user to navigate to the `Book` page, which provides information about a selected book. From the `Book` page, the user may navigate back to the `Categories` page or forward to the `Author` and `Review` pages, which provide information about book authors and book reviews. Additionally, the `Book` page allows the user to go to the `Add Review` page to issue a review. The `Authors` page lists authors and allows navigating to the `Author` page, which shows information about a selected author. The last page that is accessible from the main menu is the `Search` page. It allows to execute a search, for that results are presented on the `Search Results` page.

Finally, the `User` page provides a personal overview for a logged-in user of the Book Portal and it lists the user's reviews. From this page, the user may navigate to the `Manage Review` page that allows to change a review or to assign a review to another publication.

2.2.3.3 User Interface Specification

The previous section provided an overview of the Book Portal's navigation structure. However, to facilitate the comparison of Web engineering methods that support the fine granular modeling of user interface components, each page of the portal should be defined more accurately. Subsequently, each page of the Book Portal application is described briefly and illustrated by a simple sketch. The first set of pages is depicted in Figure 2.3

The `Portal` page provides a horizontally-aligned main menu that is placed near to the top of the page. Note that this menu is included into each page of the portal. Subsequently, this fact is not mentioned repeatedly. The menu contains three entries that point to the `Categories`,

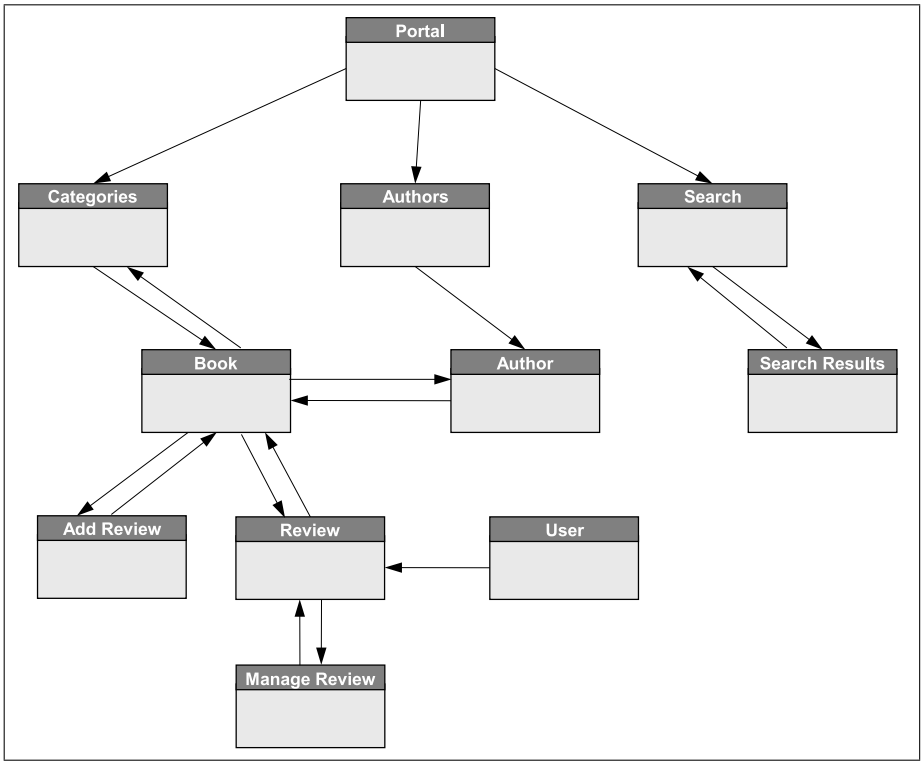

Figure 2.2: Navigation Structure of the Book Portal

the `Authors`, and the `Search` pages, which represent three different ways for the portal's user to look for books. The main area of the page shows a guided tour of the ten most popular books of the portal, depending on review scores issued by portal users. This view shows the title, the list of authors and the abstract for a selected book. The user may navigate through the ten most popular books using the corresponding navigation buttons.

The `Categories` page allows the user to select a book from a hierarchical category list. The page features two areas, occupying the left-hand and the right-hand sides of the page, respectively. The left area shows a category tree that may be browsed interactively by showing or hiding sub-categories, i.e., branches of the category tree. If a certain category is selected, all books of the corresponding category are listed in the right hand side area. This listing shows for each entry the book's title, and the list of authors. A certain book entry may be selected in order to navigate to the corresponding `Book` page.

Another area of the Book Portal application, which is accessible from the main menu is the search area. It consists of the `Search` and the `Search results` pages. The `Search` page is very simplistic. It contains a search form consisting of a text entry field and a *Search* button. After entering a seach term and activating the *Search* button, the user is redirected to the `Search Results` page. This page list the results of the search, i.e., a list of books that match the query. Entries of the result list may be selected in order to navigate to the corresponding `Book` page.

Additionally to browsing categories or conducting a keyword search, the portal user may also examine the portal's author database. Corresponding pages of the Book Portal, i.e., the `Authors` and `Author` pages are depicted in Figure 2.4.

The `Authors` page is the last page that is accessible via the main menu, which is contained by all pages of the Book Portal. This page displays an index of authors sorted by their last names. Each entry in the authors list contains the first names and the last name of the corre-

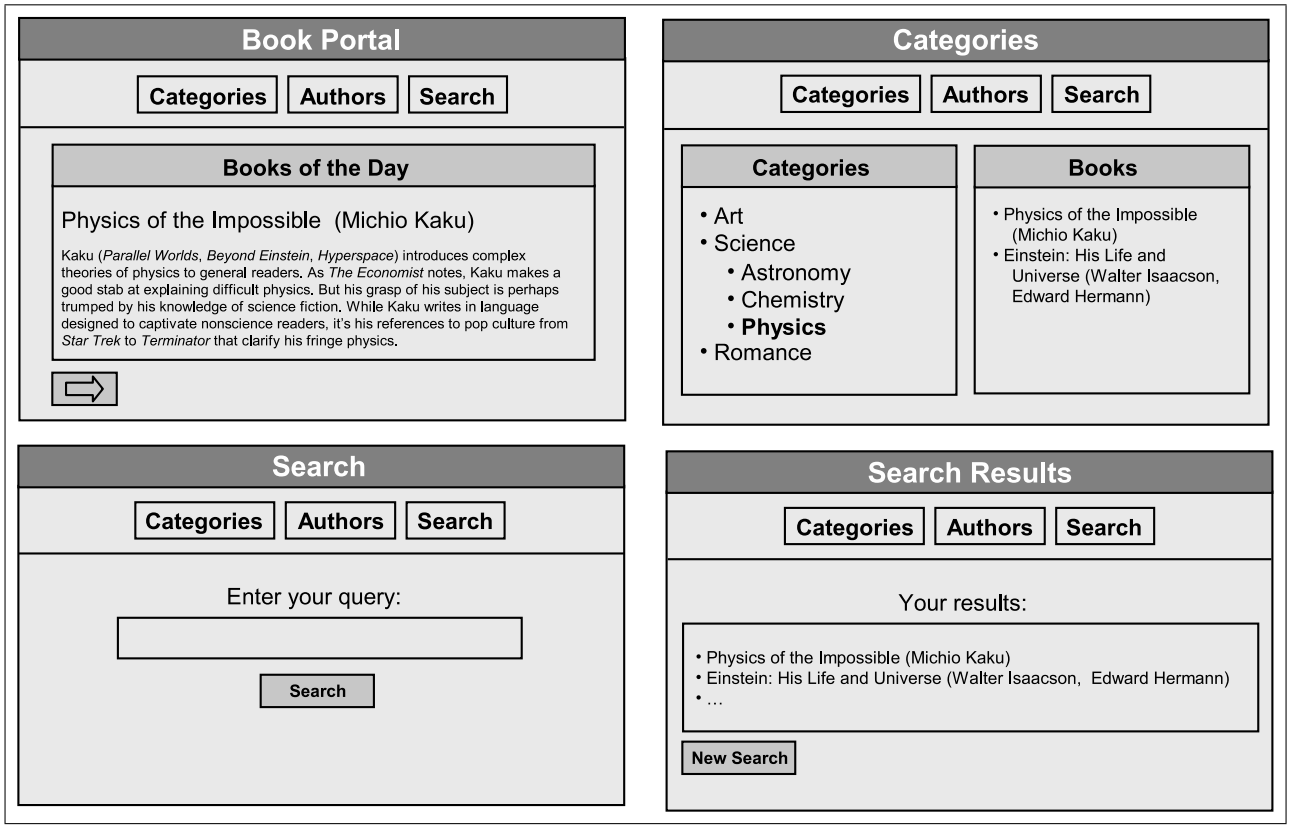

Figure 2.3: The `Portal`, `Categories`, `Search` and the `Search Results` Pages

sponding author. Entries may be selected in order to navigate to the `Author` page.

The `Author` page contains an overview of an author and his books. The author's name is shown at the top of the page and the left-hand side of the page displays the author's biography. The right-hand side of the page provides an index of the author's books. For each book the index displays the book title. The portal user may select a book entry from the index in order to reach the `Book` page.

The `Book` page is one the most important pages of the portal. It may be reached through different navigation paths of the portal, i.e, from the `Categories`, the `Author`, or from the `Search Results` pages. It is also the most complex page of the portal showing information about several content objects. The page shows an overview of a single book of the Book Portal and of some related objects as well. The book's title is shown at the top of the page and a short abstract on the left-hand side. The right-hand side contains the list of authors and some further details about the book, e.g., the books categories, the number of pages, etc. The bottom area shows all reviews that have been issued for the corresponding book by users of the Book Portal. A review entry shows the review title, the name of the user that issued the review, and the review score. The *Add Review* button at the bottom of the review area may be activated in order to navigate to the `Add Review` page.

The final set of pages of the Book Portal deals with presenting and managing reviews. Corresponding pages are depicted in Figure 2.5.

The `Review` page shows information about a book review. This page is accessible from the `Book` page, which lists reviews for a certain book or from the `User` page, which shows reviews of a certain user. The page displays the review title at the top and the review text on

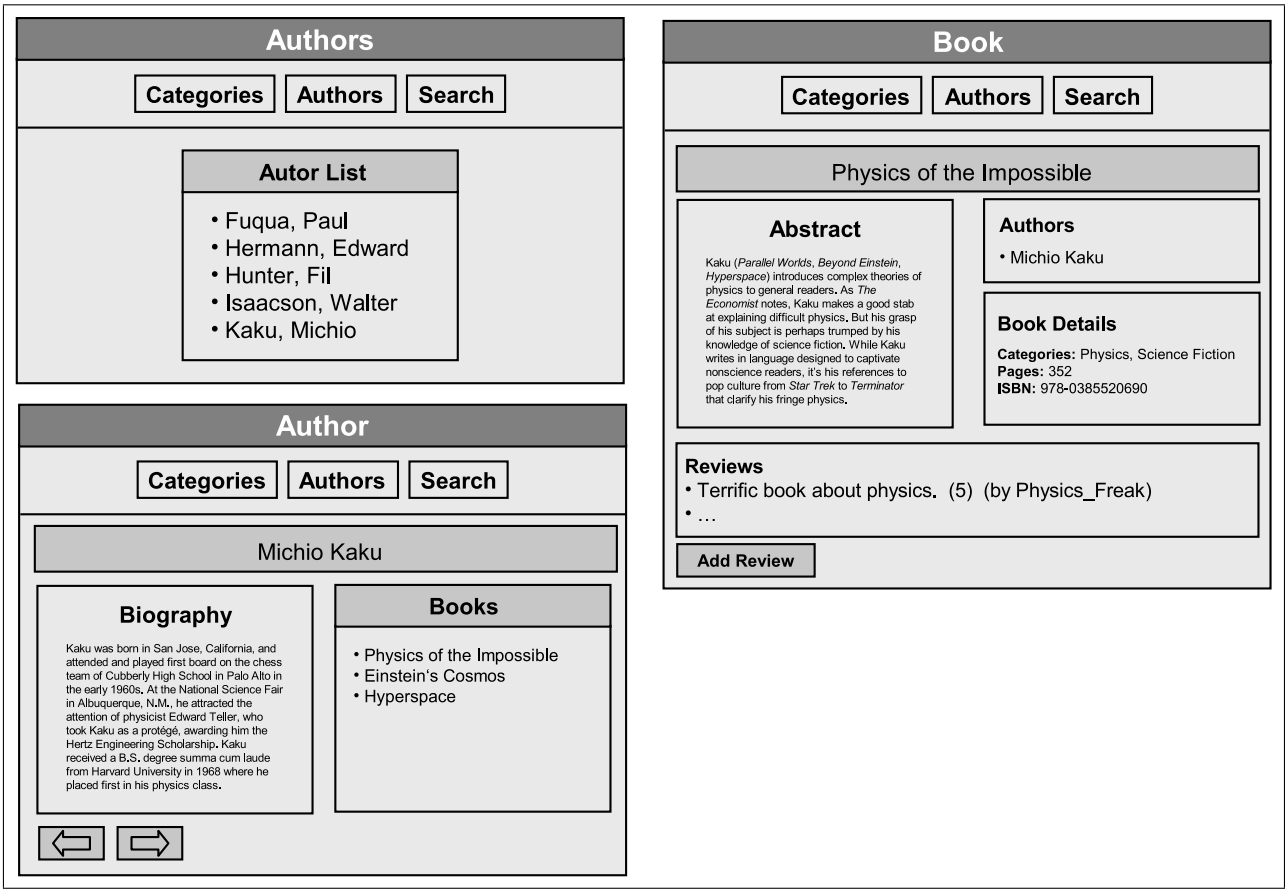

Figure 2.4: The `Authors`, `Author`, and the `Book` Pages

the left-hand side. The right-hand side of the page displays some additional information about the review, i.e., the review score, the nickname of the user who had issued the review, and finally the date on which the review was associated to a certain book. The `Review` page also provides a set of navigation buttons at the bottom of the page. The *Delete Review* and *Manage Review* button allow the user to delete a review or to navigate to the `Manage Review` page, respectively. Note that these buttons are shown only if the user is logged in. Finally, the *Back to the Book* button leads the user back to the book page.

The `Add Review` page allows a user to issue a review for a selected book and may be reached from the `Book` page. The page provides a Web form that contains a field for each attribute of a review, i.e., the title, the review text, and the score. At the bottom of the form, the *Add review* button allows to issue a review for the corresponding book using the values that were filled into the form. After saving the review, the user is sent to the corresponding book page.

A `User` is a special object of the Book Portal. It does not only represent simple content but is associated with a person who owns a user account and is allowed to log in to the Book Portal. Usually, there are a number of additional pages that provide functionality to support user management, e.g., a registration page, a log-in page, etc. However, for the sake of simplicity, these parts of the Web Application are omitted. Correspondingly, the `User` page simply displays information about a user. The page is divided into two vertical areas. The left-hand side of the page displays personal data, i.e., the name, and the email address of the user. The right-hand side area of the page lists all reviews of the user in an index. Each review entry contains

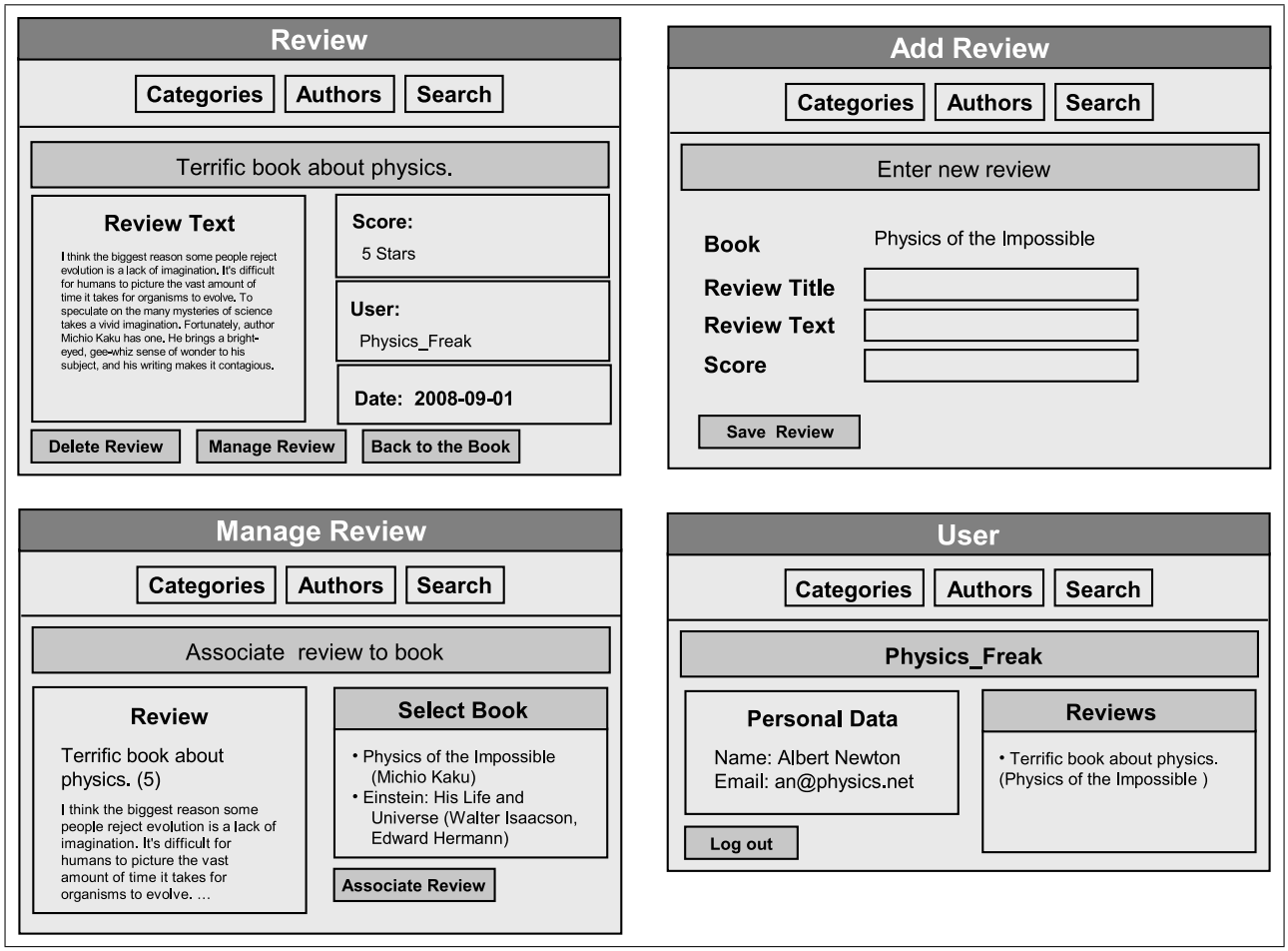

Figure 2.5: The `Review`, `Add Review`, `Manage Review`, and the `User` Pages

the title of the review, and the title of the publication for which the review was issued. After selecting a certain review from the index, the user is sent to the `Review` page.

Finally, the `Manage review` page displays an overview of a review and allows a user to associate a review to a book. Note that the Book Portal provides a simple way to create reviews. The `Book` page contains an *Add Review* button that leads to the `Add Review` page. Reviews issued on this page are automatically associated to the corresponding publication. However, the Book Portal also allows a user to reassign a review to an arbitrary book using the `Manage Review` page. This page contains two vertically-oriented areas. The left-hand side area displays a review, i.e., a title, the review text, and the review score. The right-hand side area shows the book to that the review is associated and provides a user interface component that allows to change this association, e.g., a selection of all books and a *Reassign* button.

2.3 The Technological Viewpoint

Previous sections explained what Web Applications are and how they may be categorized regarding different points of views. Additionally, the last section provided a comprehensive Web Application example that will be used throughout this work. However, until now, it is unclear which technologies may be used to implement Web Applications. Therefore, this sec-

tion sheds some light on this topic and introduces different models, protocols, and languages that together provide an execution environment for Web Applications.

2.3.1 The Big Technology Picture

A traditional application usually runs on a single computer and it is compiled for a certain computer architecture and operating system. The executable program, i.e., the complete program byte code resides on the computer running the application. The application may interact more or less directly with the hardware and software of the host computer, i.e., utilizing appropriate libraries of the operating system.

In contrast to that, a Web Application is not a single executable, which can be simply run on a computer. It depends on another application, the **Web browser**, which allows the user to interact with the Web Application. The Web browser runs on the **client** computer and interprets Web application code that is sent from the **Web server** using a range of protocols and languages that are explained in the subsequent sections. Executing a Web Application includes a client and one or more server computers thereby implementing the client-server paradigm (see Section 2.3.4.2). Client and server usually communicate over an intranet or over the Internet. Figure 2.6 depicts a typical communication scenario over the Internet.

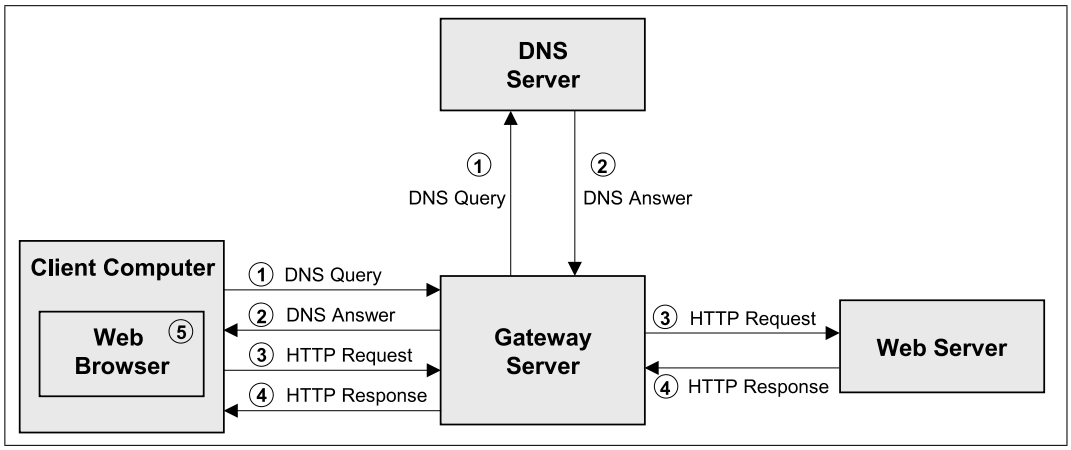

Figure 2.6: Web Application Execution

The user of the client computer starts the communication requesting a resource identified by a URL, e.g., *http://www.example.org*. However, the client does not know which computer on the Internet hosts the corresponding resource. It is only aware of the IP address of the **Gateway Server**, which connects the client computer to the Internet and of the IP address of the **DNS server** (see Section 2.3.3.2), which can resolve the given URL to the corresponding IP address. Correspondingly, the client sends a DNS query ① to the DNS server through the gateway server and requests the IP address for the given domain. The DNS server responds with a DNS answer ②, which includes the required IP address. Having the required IP address of the Web server that hosts the desired resource, the client communicates with it via HTTP (see Section 2.3.3.3). It sends an **HTTP Request** ③ to the Web server and, if the resource is available, it is sent back via an **HTTP Response** ④ to the client. In case of an error, e.g., the resource is not available, a corresponding HTTP response is returned. Finally, the Web browser ⑤ interprets the Web server's response, which is usually an **HTML** document and displays it to the user.

Note that this example describes only the request and delivery of a single Web page. During the execution of a Web Application, this pattern is used repeatedly to deliver a number of Web pages and other resources to the client so that the whole functionality of the Web Application may enfold.

Also note that the scenario depicted in Figure 2.6 is just a typical example. There are many circumstances that may alter this scenario. For example, the DNS resolver at the client, the component that tries to acquire the IP address of a certain host by communicating with the DNS server, usually operates a cache that stores IP addresses for frequently visited domains. Thus, the client computer does not have to ask the DNS server each time it requests a resource from the same Web server. Furthermore, the Internet gateway and the DNS server may be the same computer or they may be several computer nodes apart.

It is important that any Web Application developer has a basic understanding of this technology setting. Whereas low level communication, i.e., DNS resolving, is handled transparently by the operating system, elements of the HTTP communication, i.e, HTTP requests and HTTP responses, are fundamental units of Web Application development. They negotiate not only the simple transfer of Web Application resources but allow to exchange diverse information that control the execution of the Web Application. Examples include the exchange of Cookies (see Section 2.3.3.6) for storing control data at the client or the usage of ETags for caching purposes. Consequently, Web application development frameworks usually provide high-level objects and methods for controling HTTP communication.

2.3.2 Network Layer Models

The communication scenario shown in Figure 2.6 concerns a number of hardware and software components. There are potentially many computers involved that may be linked to each other via different connections, e.g., cable or wireless connections and the communicating participants may employ a number of protocols and languages on different levels of abstraction. This section introduces two different models that help to understand how this communication is organized and ultimately how components, protocols- and services play together at different levels to facilitate the operation of Web applications.

2.3.2.1 OSI Reference Model

The Open Systems Interconnection (OSI) Reference Model [ZD83][Bra07a][Tan03] was first proposed in 1983 and introduced a seven-layer model for inter-process communication, which is depicted on the left-hand side of Figure 2.7. The basic idea of the model is to create different layers for communication functionality at different abstraction levels so that only neighbouring layers may interact. Basic design principles for the model required that each layer may have a well-defined function and that the amount of information that must be exchanged between the layers should be minimal. Additionally, the number of layers should be large enough so that distinct functions may be encapsulated in different layers and small enough for an appropriate implementation. Subsequently, all layers of the OSI model are described briefly.

The **physical layer** deals with communication hardware and defines how bits are transmitted over a physical channel, e.g., a copper cable. This layer is typically concerned with mechanical and electrical issues, i.e., how many pins does a cable connector have, what voltages are used to differentiate between a 1 and a 0, how long the transmission of a single bit lasts,

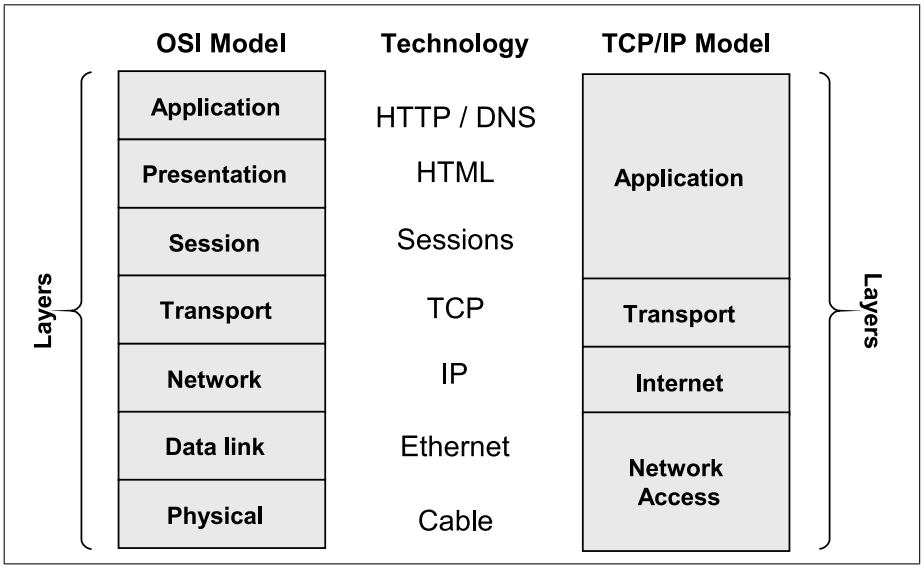

Figure 2.7: Network Reference Models

etc. Also, the beginning and the end of a transmission must be recognizable.

The **data link layer** controls the flow of data and ensures that transmission errors are detected and handled appropriately. This is achieved by breaking the data to be sent into **data frames**, e.g., a few thousand bits that are sent over the physical layer in a controlled manner. To this end, data frames are sent sequentially between sender and receiver. If an error occurs, e.g., the frame is lost, it may be retransmitted or the error is signaled to the upper layers. In many protocols of this layer, the receiver sends an **acknowledgment frame** to the sender in order to signal a successful transmission.

The **network layer** controls the operation of the network. It handles routing packets from source to destination. One key issue of routing packets is to determine the path of packets in the network. Paths may be static for all packets that are sent in the network, may be dynamically determined for each conversation or even be different for each single packet. Another important task of this layer is to handle different loads, i.e., to deal with senders that are too fast or receivers that are too slow. Last but not least, the network layer has to cope with routing packets between different networks. This is a difficult task because different networks may use different packet sizes or protocols.

The main task of the **transport layer** is to receive data from the session layer and regardless of the size of the data to ensure the sequential and error-free transport of data from the source to the destination. To achieve this, the data is usually split into smaller packets that have an appropriate size for the network layer. Note that the transport layer and all other layers that are higher up in the hierarchy of the OSI model are concerned with point-to-point communication. They are not aware of network paths, they only consider the source and the destination communication nodes.

The **session layer** provides the concept of a session between two computer programs that are communicating with each other. A session may be initiated, kept alive, or closed. In the keep alive phase the communication partners may utilize different services of the session, which include dialog control, i.e., which partner is to transmit and which is to receive data or synchronisation, i.e., making sure that long transmissions may be paused or continued.

The **presentation layer** defines the syntax and the semantics of information that is to be

transmitted between computer applications. This is important so that computer applications may rely on standards and do not have to re-implement common data encodings. Well-known examples are the ASCII standard that codes characters or the Multipurpose Internet Mail Extensions (MIME) for attaching different non-text information to emails.

The final layer of the OSI model is the **application layer**. It allows for protocols that provide high-level input and output routines that may be used by computer applications for communication. An example is HTTP (see Section 2.3.3.3) that allows to transfer Web pages and other resources between computer applications. As all other layers of the OSI model, the application layer relies on functionality provided by lower levels. Consequently, a typical Web browser relies on sessions and transfers data utilizing well-known file formats.

2.3.2.2 The TCP/IP Reference Model

Although the OSI model itself is still valid, associated protocols and implementations are rarely used due to complexity and performance issues. In contrast to that, protocols of the TCP/IP Reference Model [CK74][Tan03][Hun02] are widely used, whereas the model itself is not of much importance. The most prominent implementation of the TCP/IP model is the Internet. Consequently, all Internet applications, e.g., Web Applications rely on protocols defined by this model. The TCP/IP model is depicted on the right-hand side of Figure 2.7.

The **network access layer** is the lowest layer of the TCP/IP model. It corresponds to the first two layers of the OSI model and deals with sending data frames over the physical network. Actually, the model does not define standard protocols for this layer. Corresponding functionality is implemented by device drivers and operating system sub-routines, thereby being highly device dependent.

The **Internet layer** of the TCP/IP model is comparable to the network layer of the OSI model. Its main task is to route packets from one computer to another through one or more computer networks. To this end, the layer defines the **Internet Protocol (IP)** [Uni81], which provides several key formats and routines. The **IP address** uniquely identifies a computer in the network and an **IP packet** is the basic unit of transmission between communicating peers. Additionally, the Internet Protocol defines routines for routing IP packets and defines how packets may be fragmented or reassembled if necessary.

The **transport layer** of the model is similar to the transport layer of the OSI model. It makes sure that two peers are able to exchange data regardless of the size of the data or the location of the peers in the network. To this end, the layer defines the **Transmission Control Protocol (TCP)** [CK74][CDS74], which transports data from the source to the destination computer via splitting the data stream into several IP packets and passing them on to the Internet layer. The TCP process on the destination computer reassembles the packets into the original data stream.

The session and presentation layers of the OSI model proved to be an unnecessary overhead. It is still important that applications utilize standards that are defined in this context, however, extra layers for this functionality are not practicable. Thus, these layers and the application layer of the OSI model have been merged into the **application layer** of the TCP/IP model. The application layer provides higher-level protocols for direct communication between computer applications.

2.3.2.3 Tying it all together

The relevance of the introduced reference models for Web Applications becomes clear if protocols and languages that are essential for Web Applications are mapped to corresponding layers of the models. Figure 2.6, which shows a typical communication scenario that takes place during the execution of a Web Application, already introduced some of the higher-level protocols, e.g., HTTP and DNS. Figure 2.7 shows a more complete list of relevant technolgies and maps them to different layers of the OSI and TCP/IP models, respectively.

During the execution of a Web Application any two neighbouring computers that are involved in the process need to be able to exchange messages. To this end, a physical connection, e.g., a **computer cable** must exist that connects them. Lower-level protocols of the models ensure that communicating peers during the execution of a Web Application may utilize higher-level Web-application-specific protocols and languages. For example, if communicating computers are connected by cables, the **Ethernet** standard may be used, which defines protocols that allow to send data over coaxial or telephone cables but abstracts from the physical properties of these transport mediums. The **Internet Protocol (IP)** ensures that data packets are correctly routed between network nodes and the **Transmission Control Protocol (TCP)** establishes an error-free channel enabling peer-to-peer communication. Note that these lower level protocols are not Web-application-specific, however, they are basic elements of network communication and therefore also essential for communication between Web clients and Web servers.

Finally, the application layer of the TCP/IP model (and corresponding layers of the OSI model) provides room for Web Application-specific-protocols. Relying on lower-level protocols, Web client and Web server programs may directly communicate using HTTP, a protocol that defines the typical request-response pattern for communication between Web client and Web server. Additionally, Web pages that are sent by the Web server and are interpreted by the Web client are encoded in HTML.

Note that the TCP/IP model does not define an explicit Session Layer, however, many Web Applications utilize *sessions* to overcome the stateless nature of HTTP communication. Unlike the other technology examples in Figure 2.7 the notion of a session is a programming paradigm and not a concrete technology.

2.3.3 Technologies

The previous sections provided an overview of how Web Applications are executed and which technologies are involved in the process. Different protocols and languages were mapped to different layers of the introduced reference models for network communication. In this section, the most important technologies for Web Applications are introduced in more detail. Note that the discussion of lower-layer protocols, e.g., Ethernet, IP and TCP, is omited as they are not Web Application specific and, according to the philosophy of layered network models, they may be complemented or even replaced in future networks by other protocols. Note that this is already happening to Ethernet as many networks utilize wireless communication technology as an alternative. A detailed discussion of lower-level protocols may be found in [Tan03] and [Cla03].

2.3.3.1 Unified Resource Identifier

A Uniform Resource Identifier (URI) [BL94] is one of the core technologies of the WWW. It is a structured name that unambiguously identifies a resource in the WWW. As Web Applications are WWW resources, they must be identifiable by one or more URIs. An URI has the following syntax:

```
<scheme>://<authority><path><query><fragment>
```

The following example is a correct URI:

```
http://me@example.org:80/section/document.html?accessed=today#toc
```

A **scheme** defines the syntax and semantics of all other remaining components of an URI. Therefore, the URI syntax is very flexible and URIs may take diverse forms. For example, the "http" scheme may be used to identify resources on the WWW. A URI using this scheme is usually interpreted by a Web browser that tries to acquire the corresponding Web resource using the remaining sections of the URI.

The **authority** section identifies the authority that manages the required resource. For example, if the "http" scheme is used, the authority section identifies a computer in the network that is able to provide the corresponding resource. In this case the authority section may be composite and may use the following form:

```
<userinfo>@<host>:<port>
```

The **host** part is mandatory and identifies the computer on the network. To this end, the IP address or a domain name (see Section 2.3.3.2) for the host must be provided. Additionally, a **port** may be specified, which is assumed to be a port on the host computer that is bound to a Web server process providing Web resources. Finally, the **userinfo** part may be specified that identifies a user requesting the resource. Note that the current example defines the domain name "example.org" the port 80 and the user name "me".

The **path** and the **query** sections of the URI identify a resource managed by the authority. The path section has a hierarchical structure, whereas the query section may contain further information that may be used to identify a resource. The query section is delimited by a question mark (?) from the path section, if a query section is present. The path section of the URI is used by a Web server to identify a resource stored in a corresponding hierarchical storage. However, this is of course not mandatory and not always the case. Also, key-value pairs are often used as part of a query. Thereby, a *key* is delimited by an equal sign (=) from the *value* and several key-value pairs are delimited by ampersands (&). In the current example, the path section is "`/section/document.html`", which identifies the resource named "document.html" under the structure component named "section" and the query contains the key-value pair "`accessed=today`".

The final section of an URI is the **fragment** section. It may identify a specific fragment of a resource, a specific view of a resource, or also a completely different secondary resource. However, in most cases the fragment section is used to refer to a specific section of the primary resource. If the fragment section is present, it is delimited by the number sign (#).

Note that the terms URI and URL are often used as synonyms. However, this is not correct. A Uniform Resource Locator (URL) is a special URI that not only identifies a resource but also

locates it, i.e., also identifies the location of the resource on the network. A URL is a URI, thus it uses the same syntax. For example "`mailto:me@example.org`" is a URI but not a URL. It uses the "`mailto`" scheme and unambigously identifies an email address but does not locate it, as email addresses do not have a location. In contrast to that "`http://example.org/index.html`" is a URI and also a URL, because it also idendifies the location of the resource "`index.html`" on the host identified by the domain "`example.org`". Of course, all resources in the WWW must be locateable, thus URIs using the "`http`" scheme are always URLs, too.

Also note that URLs using the "`http`" scheme may identify a host in the authority section by using a domain name. This is a very important aspect for two reasons. First, names like "`www.example.org`" are much easier to memorize than the corresponding IP address "`208.77.188.166`". Second, the extensive growth of the WWW is only manageable because it relies on the highly scalable Domain Name System.

2.3.3.2 Domain Name System

The Domain Name System (DNS) is one of the core technologies of the Internet. It is a distributed system that maps hierarchical names to IP addresses, thus, Internet hosts may be identified by names that are easier to recognize. The hierarchical structure is not only used to structure the names themselves but also to distribute the authority for managing names at different levels of the name tree. Therefore, the DNS is a highly scalable distributed system that is managed by many countries, organizations, and individuals.

A domain name consists of labels separated by dots. For example, "`www.example.org.`" is a valid domain name. Note that the trailing dot is not accidental, it denotes the root of the name tree. Labels from the right to the left separated by dots build the name tree. Labels at the first level are also called Top Level Domains (TLD). They are managed by the Internet Assigned Numbers Authority (IANA) [URI08f]. Examples are "`com`", "`edu`" or "`info`" and additionally each country has a two character TLD according to the ISO 3166 list of country name abbreviations, e.g., "`de`" for germany. Labels at the second level are also called Second Level Domains. Usually, they are managed by the authority who is responsible for the corresponding TLD. Generally, the management of all further sub-domains are recursively delegated to organizations or individuals who usually rent these sub-domains from the corresponding authority.

Resolving domain names to IP addresses follows a similar hierarchical pattern. The authority responsible for a domain usually operates one or more domain name servers that map names to IP addresses for the corresponding domain. A name server may be asked for the IP address belonging to a specific domain. If the name server has a corresponding record, it returns the answer immediately. If not, it delegates the query to another name server that probably has the required information or may delegate the query towards the right direction. Delegation may proceed downwards, i.e., towards a name server that governs a subdomain or upwards, i.e., towards a name server that is responsible for the parent domain.

Note that name servers heavily rely on redundancy and caching, so that domain queries are often answered by a name server that is not even responsible for the domain in question. This reduces the load on name servers that govern popular domains.

2.3.3.3 HyperText Transfer Protocol

Another cornerstone of WWW technology is the HyperText Transfer Protocol (HTTP) [BLFG+99]. Since the birth of the WWW, this protocol has been used for communication between Web

clients and Web servers. The communication pattern is simple. The Web client requests a resource by sending an **HTTP request** to the server and the server answers with an **HTTP response**. HTTP is a text-based, human-readable protocol, thus both request and response may be easily interpreted. A simplistic HTTP request may be as short as depicted in Listing 2.1.

```
GET /index.html HTTP/1.1
Host: example.org
```

Listing 2.1: HTTP Request Example

The first line of an HTTP request begins with the specification of a **method** that defines what action should be executed by the Web server. In this example, a resource should be acquired, thus the request uses the "GET" method and provides the location and the name of the desired resource, which is "/index.html". Table 2.2 shows a selection of methods that may be used in an HTTP request.

GET	Retrieves a resource identified by the request.
POST	Sends data to the server in the context of the resource identified by the request.
PUT	Sends a resource to the server to be stored as the resource identified by the request.
DELETE	Requests to delete the resource on the server identified by the request.
HEAD	Request to get only the HTTP headers of a server response without the actual resource identified by the request.

Table 2.2: Common HTTP Request Methods

Note that at the end of the first line of the HTTP request the desired HTTP version is specified, in this case version 1.1. The first line of the request may be followed by additional lines, the so-called **request header fields**. The only mandatory header field in a request is the **host** field that specifies a domain name from which a resource is requested. Note that a blank line after the request headers is necessary to mark the end of the request headers section.

Similar to an HTTP request, an HTTP response has also a simple structure. A typical HTTP response is presented in Listing 2.2.

The first line of the response contains the currently used HTTP **version**, a **status code**, and a corresponding **reason phrase**. The status code is a 3-digit integer indicating the status of the response, whereas the reason phrase adds a short explanation of the status code intended for human readers.

After the first line of the HTTP response, a number of **response header fields** and **entity header fields** follow, that contain information about the Web server, the communication process, and the requested resource. In the current example, the *Server* response header field gives away that the host uses the Apache Web server running on the operating system CentOS and the *Content-Type* entity header field states the content type of the resource, which is "text/html" in this case. Note that header fields may be specified in an arbitrary order. Finally, after a blank line marking the end of the header field section, the **entity body** follows. The body section may contain text or binary information corresponding to the *Content-Type* field. The current example contains a simple HTML document.

```
HTTP/1.1 200 OK
Content-Type: text/html
Date: Mon, 19 May 2008 13:48:38 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 15 Nov 2005 13:24:10 GMT
Content-Length: 101
Connection: close

<HTML>
 <HEAD>
  <TITLE>Example Web Page</TITLE>
 </HEAD>
 <BODY>
  Example Content
 </BODY>
</HTML>
```

Listing 2.2: HTTP Response Example

Note that the HTTP 1.1 specification [BLFG+99] describes a number of header fields that may be used in HTTP request or response messages to manage and optimize the communication between Web client and Web server. This includes authentication, cookie handling, cache management, etc. The reader is referred to the specification for a detailed description.

2.3.3.4 HyperText Markup Language

Besides URIs and the HTTP, the **HyperText Markup Language (HTML)** is the third and last core technology of the WWW. HTML is the basic language for building Web pages and Web Applications. If a Web client requests a Web page from a Web server, the server sends a HTTP response with an entity body that contains the Web page encoded in HTML. The Web client, usually a Web browser, interprets the HTML source code of the Web page and renders it for the end user. The current HTML 4.01 specification [RLHJ99] and the upcoming HTML 5 W3C working draft [HH08] define a wide range of language elements for building Web pages. This section gives a brief overview of the most important features of HTML but, of course, does not cover the language's whole functionality.

HTML is an application of the Standard Generalized Markup Language (SGML) [Int86], which is a meta language for creating markup languages. Consequently, HTML is a markup language that provides content, structural and presentational information. An HTML document contains a number of (nested) **elements** that are denoted by **tags** and assign specific semantics to the corresponding fragment of the document. Tags are used by Web clients to interpret a document encoded in HTML and to render it in an appropriate fashion. Listing 2.3 shows a simple HTML document.

The list of allowed elements (tags), attributes, and nesting rules of an SGML application is defined by the Document Type Definition (DTD); accordingly, this is also the case for HTML. A correct HTML document begins with the DOCTYPE definition that specifies which DTD version is to be used to validate and interpret the document. Note that different DTDs have been defined for HTML 4.01 that allow a different set of tags to be used in a document. The

current example uses the "strict" DTD [URI08e] that omits language elements that serve only presentational purposes.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
      "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
   <HEAD>
      <TITLE>My first HTML document</TITLE>
      <LINK href="style.css" rel="stylesheet"
            type="text/css">
      <SCRIPT type="text/javascript"
              src="script.js"></SCRIPT>
   </HEAD>
   <BODY class="cool">
      <P>Hello world!</P>
      <A href="another.html">Another document.</A>
   </BODY>
</HTML>
```

Listing 2.3: Simple HTML Example

The basic structure of an HTML document is simple. The main element is the HTML element that is divided into two parts. The HEAD part mainly contains meta data, e.g., the title of the document provided by the TITLE tag. The BODY section contains the document's main content. Note that elements may be nested according to nesting rules defined in the DTD. The order and the hierarchy of elements define the structure of the HTML document.

Besides the already introduced elements, the language defines a set of elements that represent paragraphs, lists, tables, hypertext links, etc. Most elements are composed of a start tag, the content, and an end tag. The element's name, e.g., "TITLE", appears in the start tag and in the end tag. The content section of an element may contain text, other elements, or both.

HTML elements may have attributes that add specific semantics to the element. Attributes are provided as key-value pairs in the start tag of an HTML element. For example the *class* attribute of the BODY element with the value "cool" assigns a style instruction that is stored in an external style sheet to the corresponding element. The allowed set of attributes for each element is defined in the DTD.

Early versions of HTML allowed only rudimentary control over the presentation of HTML documents and rendering conventions were hard-coded into the Web browser. HTML 3.2 introduced a set of element attributes for specifying alignment, font sizes, and colors. However, this was not an ideal choice as content and presentational aspects were intermixed. To avoid that, HTML 4 and upcoming versions pursue another course. They rely on so-called Cascading Style Sheets (CSS) that specify all presentational aspects of an HTML document.

Style definitions are usually stored in a separate document, the style sheet, which may be linked to the HTML document via the LINK element in the HEAD section. Style definitions that are specified in the style sheet may be linked to an HTML element using the element's *class* attribute. Listing 2.3 links the style sheet "style.css" to the presented example document.

Perhaps the most important feature of HTML is its support for hypertext (see 2.2.1). To this end, the document's author may define a link between an HTML document and another Web

resource, which may be the same or another HTML document or an arbitrary resource on the Web. A link has two anchors and a direction. A hyperlink in HTML is defined with the A element. Note that the BODY section of the example HTML document in Listing 2.3 defines a link to a second document named "another.html".

Finally, HTML allows for scripting, which is basically the direct or indirect integration of program code into a HTML document which is executed at the client side, i.e., by the Web browser after the delivery of the HTML document from the server. Scripts may be executed by the browser when the document loads or after a specific event, e.g., the user moves the cursor over a specific area or activates a link. HTML's support for scripting is independent from the actual scripting language. It offers two ways for specifying scripting code for a document. Similarly to a style sheet, the code may be stored in an external file and referenced by a SCRIPT element in the HEAD section. Listing 2.3 shows an example linking to the script file named "script.js". Alternatively, the scripting code may be directly embedded into the HTML document. To this end, the program code may be specified as content of the SCRIPT tag and placed into the HEAD or BODY section of the document. Note that the scripting capability of HTML is an essential prerequisite for the development of Web Applications.

2.3.3.5 Sessions

A **session** captures the state of the interaction between a Web client and a Web server for a certain user for a certain period of time. An example scenario is the interaction of a customer with a Web shop. The customer is allowed to place products into a virtual shopping cart and after selecting all required products, he is asked to complete the order procedure, which usually includes providing personal information, etc. in a multi-page dialog. To capture the entire state of this interaction the Web shop application has to store a number of data items, e.g., which products have been placed by the user into the shopping cart and what information has the user already filled in during the order procedure.

However, the HTTP is a stateless protocol. Thus, after a single HTTP request-response dialog the Web server considers the communication to be finished. Any further requests are considered to be independent from any previous ones. Of course, this is not satisfactory because in the depicted scenario a number of requests from the Web client must be identifiable as part of the same communication dialog. Unfortunately, HTTP offers no solution for associating a number of HTTP requests as a cohesive unit.

To circumvent this deficiency, Web Applications have to explicitly support sessions. There is no standard way to implement sessions, thus Web application developers usually choose one of the following four solutions. First, all state values may be stored in cookies (see Section 2.3.3.6) at the client side. The Web server may use these values as required to determine or change the state of the interaction. This solution has the disadvantage that the size and the number of allowed cookies per site is restricted.

Second, an improved version of the first approach uses a unique identifier in a cookie at the client side and manages all the required session data in a server-side map. Thus, the Web server only has to acquire the unique identifier of a client to be able to look up required state values in it's own storage. Besides being free of some cookie limitations, this approach has a further important advantage. The Web server is able to use its own storage for handling state values, thus depending on the capabilities of this storage, it may also use complex data types. However, all solutions using cookies have a general drawback. The storage of cookies may be

restricted or even completely forbidden in the settings of the Web client. Many Web users have disabled cookies as they may be misused for advertising or other undesirable purposes.

Third, all state values may be included into every request specified as query parameters in the URL (see Section 2.3.3.1). This solution has the advantage that it is independent of cookie limitations, however it has some of the same drawbacks as the first solution. The length of a URL, thus the space to store state values, is usually limited. Although the 255 byte limit [BLFG+99] is no longer valid, most Web clients still have a built-in restriction regarding the allowed length of a URL. Additionally, the problem of using complex data types as state values still applies.

Finally, the most flexible solution is to include a unique identifier into every request and to manage state values on the server. This solution is analogous to the second approach without depending on cookies.

Ultimately, the concept of a session is an important paradigm that is crucial for Web Applications. Regardless of the complexity of a Web Application, the communication between the user and the application is usually more complex than a stateless request-response pattern. Therefore, session management is a basic challenge that must be supported by every Web Application. To this end, Web application development frameworks (see Section 2.3.5) always have built-in support to deal with sessions.

2.3.3.6 Cookies

A **cookie** is a piece of text information that a Web server may store at the client computer if the Web browser is configured to accept it. The Web browser sends cookie-data back to the Web server that set the corresponding cookie with every subsequent request. The cookie mechanism may be used for arbitrary purposes, however, there exist some common usage patterns. Cookies allow customization, i.e., a Web Application may store user settings permanently, e.g., store the favourite color scheme of a user for a particular Web site. Another popular use of cookies is user identification, i.e., a unique identifier is stored permanently to identify the current user. Finally, as described in the previous section, cookies are often used for session management.

According to the specification [KM00], the Web browser should accept at least 20 cookies from a domain and support at least a cookie size of 4 kilobyte. Of course, the size of a cookie depends on the usage pattern, however, most cookies have a size of a few hundred bytes.

Web servers and Web clients use HTTP request and response header fields to handle cookies. Using the example from Section 2.3.3.3, the response of a Web server may include a special header field to set the cookie at the client like shown in Listing 2.4.

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: id=1; path=/section;
```

Listing 2.4: Set Cookie Example

The *Set-Cookie* header field of this HTTP response instructs the Web browser to store the string “id=1”. Note that other header fields and the entity body of the HTTP response are omitted in Listing 2.4 for simplicity. The path setting defines that the cookie is relevant for all resources that are under the specified path, i.e., for all resources under the container named “section”. Accordingly, any subsequent requests from the client for a resource that resides in

the folder "section" include a corresponding HTTP request header field as depicted in Listing 2.5.

```
GET /section/document.html HTTP/1.1
Host: www.example.org
Cookie: id=1
```

Listing 2.5: Use Cookie Example

The *cookie* header field of the HTTP request in Listing 2.5 includes the cookie that was previously set by the Web server, i.e, the value "id=1" (see Listing 2.4). Note that if there are several cookies to return they are provided in the *cookie* field separated by semicolons. Also note that *Set-Cookie* header field in a HTTP response may contain besides the *path* parameter several further parameters that regard domain handling, expiration date, etc. that are omitted here for the sake of simplicity.

2.3.4 Web Architectures

Previous sections provided detailed definitions for all terms regarding Web applications and introduced all basic technologies that are relevant for implementing them. However, Web Applications are complex systems that cannot be planned and implemented in an ad-hoc manner. Thus, it is not advisable to begin the development of a Web Application by opening a text editor and starting to type the source code of the first page in HTML. Additionally, Web Application development includes many recurrent design patterns and tasks and Web Applications usually support common functionality and have many common components. Therefore, a developer should always utilize state-of-the-art software and hardware architectures, frameworks, and platforms that support Web application development. In this section, a general architecture for Web applications and a suitable architecture for an underlying platform are introduced.

2.3.4.1 Web Application Architecture

As already suggested, Web Applications can become complex systems that manage a considerable amount of data, support comprehensive business logic, and provide various ways of content presentation. To this end, this section introduces a three-layer architecture for Web Applications that is depicted in Figure 2.8.

The bottom layer of this architecture is the **Content Layer**. This layer includes all components that are responsible for the low-level management of the Web Application's data. Therefore, it usually supports all CRUD operations, namely to create, read, update, and delete data entities. Besides basic data access components, the content layer may offer more sophisticated data access interfaces, e.g., a data query language.

The middle layer of the depicted Web Application architecture is the **Operation Layer**. This layer handles most of the Web Application's business logic. Note that compared to traditional applications, Web Applications have a strong focus on navigational characteristics. Navigational features and patterns constitute an important part of a Web Application's business logic. Thus, the operation layer contains components that focus rather on higher-level data manipulation and implement arbitrary business logic algorithms.

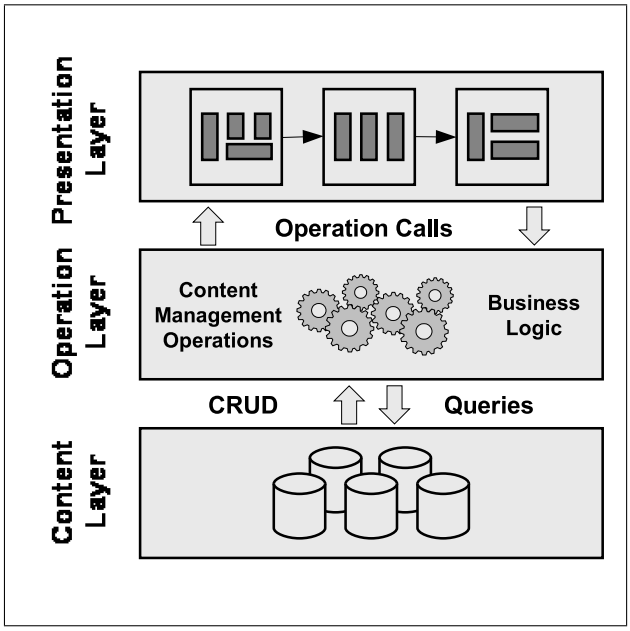

Figure 2.8: Three-Layer Web Application Architecture

The top layer of the presented architecture is the **Presentation Layer**. It contains all components that implement the Web Application's user interface. These components usually offer navigational and presentational functionality. Navigation components implement a range of navigational access structures, e.g., menus and indexes. Presentation components integrate the functionality of operation layer components into the Web Application's user interface. They show data that originates from components of the content layer or computed information that have been acquired from components of the operation layer.

2.3.4.2 The Client-Server Architectural Paradigm

Web Applications are typical client-server applications. The functionality of the Web Application enfolds during a dialog between the **client**, e.g., a Web browser, which is an application on the **client tier** and the **server**, e.g, a Web server, which is an application on the **server tier**. This simple two-tier architecture is depicted in Figure 2.9.

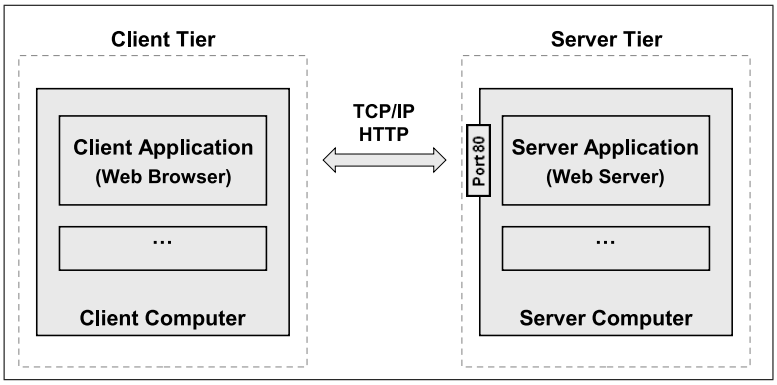

Figure 2.9: Client-Server Architecture

Naturally, the client-server architectural setting relies on the client-server communication

pattern. Consequently, it is always the client that starts the communication dialog. It requests a service from the server (HTTP request), which in case of a Web Application may be a single page of the user interface and the server responds (HTTP response) with the required resource (see also Section 2.3.3.3).

Traditionally, Web clients are **thin clients**, i.e., they implement only a small portion of the Web Application's business logic. However, since the introduction of the AJAX technology set [URI05], there is an increasing trend towards more application logic at the client side, i.e, towards **fat clients**. Using AJAX, the client does not have to wait for the response of the server to load a new page of the user interface but it may request resources in the background in an asynchronous manner, which leads to a much more smooth user experience. This new communication pattern allows to move a significant part of the application logic to the client side.

2.3.4.3 Web Application Platform Architecture

For simple Web sites, the basic client-server architecture, where the server part is fulfilled by a simple Web server, is appropriate. However, Web Applications are usually more complex systems. They often manage considerable amounts of data. For example, an average online store manages thousands of customers and a few orders of magnitude more products. Consequently, Web Applications also implement complex business logic. Therefore, Web Applications usually require a more complex technology configuration at the server side. Figure 2.10 depicts a four-tier architecture [JPMM04] that constitutes a more suitable environment for complex Web Applications.

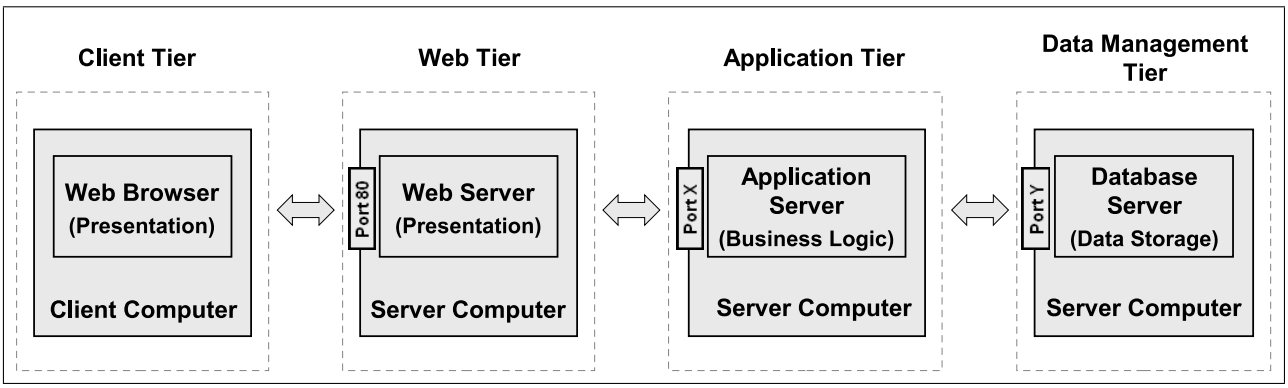

Figure 2.10: Four-Tier Web Application Plattform

The **client tier** is the same as in the simple client-server setting, which was introduced in the previous section. An application at the client tier may be a Web browser that communicates with a server. The first server component in this setting is a Web server that resides on the **Web tier**. In contrast to the simple client-server setting, the Web tier is not the only server tier and it is not responsible for executing business logic. Its task is rather to handle the presentation of resources and to manage the communication with the client tier. The execution of business logic is managed by an application server that resides on the **application tier**. An application server is a specialized software component that provides a runtime environment for applications in general and, of course, for Web Applications in particular. The application server communicates with the back-end storage, executes all business routines, and forwards

computed resources to the Web tier. The last tier of this architecture is the **data management tier**. Its purpose is to manage a Web application's data. On this tier, data storage is usually handled by a database management system.

2.3.4.4 Developing Web Applications For Specific Platforms

There are many different ways of developing a Web Application for a particular Web Application platform, i.e., to decide which layers of the Web Application are to be implemented for which tiers of the platform. Important factors that determine a specific configuration are of course the actual functionality of the Web Application, performance issues, or cost considerations.

For data-intensive Web Applications, the content layer may be implemented using a database management system and deployed to the data management tier of a Web application platform. On the other hand, if a Web Application does not have to handle much data, the content layer may be implemented as a simple file-based storage on the application tier thereby omitting the data management tier.

Components of a Web Application's operation layer are usually developed for the application tier of a Web Application platform, where business routines are executed in an application server. However, in many cases some of the business logic is moved to the client tier, e.g., implementing business routines in a client-side scripting language like JavaScript.

Dynamic presentation components of a Web Application's presentation layer usually computed on the application tier of the Web Application platform are automatically forwarded to the client tier. In contrast to that, static presentation components usually reside on the Web tier. However, in order to provide a highly responsive user interface, some of the presentation logic is developed directly for the client tier, i.e., coded in JavaScript.

Note that performance and cost considerations may determine the distribution of the tiers of a Web Application platform to one or more server computers. If high performance is required, each tier should be realized by one or more servers. On the other hand, if costs are to be minimized, one or more tiers can be realized on the same server.

2.3.5 Web Application Frameworks

The previous section introduced a general three-layer architecture for Web applications. However, it is considerably time consuming to design and implement this architecture for each new Web Application from scratch. Given the popularity of the Web it is not surprising that there is a myriad of Web Application development frameworks that provide the same or a similar architectural pattern. Most of these frameworks also bring along a pre-configured easy-to-install runtime platform or rely on a collection of standard applications, i.e., a standard Web server like Apache or the popular Tomcat application server.

A Web Application framework is a development environment and often also a runtime environment for Web Applications. It usually brings along a set of standard modules that support common tasks Web Applications have to tackle. Some of these common tasks include database connectivity, Web page template support, user and session management. The task of a database module is to support access to one or more database management systems. Web Applications often have to handle a considerable amount of data, thus the usage of an appropriate system, i.e., a DBMS, is advisable. Another common framework module is a template engine

that supports the generation of dynamic Web pages based on templates. Finally, many Web Applications support permanent user accounts and interact with users utilizing sessions. Accordingly, framework modules abstract from low-level tasks of user and session management (see Section 2.3.3.5).

A popular architectural pattern, that many Web Application frameworks are based on, is the Model-View-Controller (MVC) pattern that is shown in Figure 2.11.

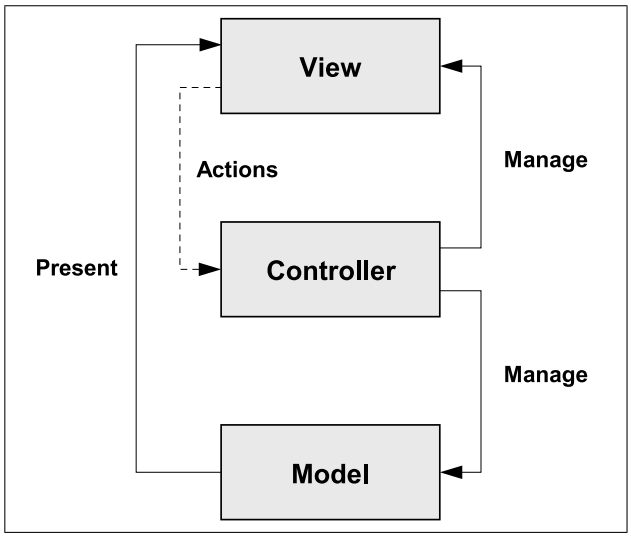

Figure 2.11: Model-View-Controller Architecture

The MVC architectural pattern is based on three main components: the model, the controller, and the view. The **model** is a domain-specific representation of the Web Application's content and it is not aware of the view or the controller. The **controller** is the heart of the system. It responds to user actions that originate from a view, possibly modifies the model, and presents further views to the user. Finally, the **view** (or views) represent(s) the user interface of the Web Application. It renders the model into appropriate user interface components.

Note that the MVC pattern is similar to the Web Application architecture introduced in Section 2.3.4.1, however, there are subtle differences. The MVC pattern is more technology-near and therefore less general in nature. For example, the controller is a central component that is responsible for the control of every user request. In contrast to that, the operation layer described in Section 2.3.4.1 is more general. It's task is to implement business logic operations that may be used, if required.

CHAPTER 3

Model-based Web Engineering

The previous chapter provided a general overview of Web applications. After a historical preamble, relevant terms regarding Web applications were explained and subsequently the most important technologies and terms for Web applications were introduced. This information builds the basis for topics of this chapter because knowing, what Web applications are, is a prerequisite for understanding how to develop them. The chapter begins with a short historical overview of the Web engineering research field. The second part gives a general introduction of the model-based Web engineering discipline. After that, the third part of this chapter explains what models are and how they may be used for Web application development. Finally, the last major part of this chapter introduces a list of model-driven Web application development methods that have been proposed over the last decade.

3.1 A Short History of Web Engineering

The history of Web engineering shows many interesting parallels to the history of the software engineering discipline and, of course, it has been greatly influenced by it. Most major trends, e.g., CASE tools, object-orientation, the use of formal languages, or the organization of the development process using standardized process models (e.g. the waterfall model [Roy87]), have influenced the research field of Web engineering.

At the dawn of the WWW, there was no need for an engineering approach to put information on the Web. Web sites were merely a collection of a few Web pages that were created by enthusiastic researchers using some text editor. However, the WWW became very popular very quickly and by the time the first Web search engines had emerged, the first methods for developing *hypermedia applications* were published [GPS93][ISB95][SR95b]. Note that hypermedia applications were not Web applications in the sense defined in Section 2.2.1. Hypermedia applications were purely informational systems without content management functionality. However, at that early time, researchers had already recognized that the development of hypermedia systems requires more expertise than just putting together a few Web pages.

By the end of the 1990s, a number of further development methods have emerged for hypermedia applications. Popular terms to characterize this new research discipline were “Web Site

Design" [FFK+98][AMM98], "Hypermedia Application Design" [SRB96], and "Hypermedia Engineering" [LH99]. The definition of Lowe and Hall [LH99] is reflected in Definition 5.

Definition 5 *`Hypermedia engineering`: the employment of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of hypermedia applications.*

Research approaches have incorporated many ideas from the software engineering discipline, e.g., object-orientation, formal languages, or graphical notations like UML [KHKR05]. Given the huge importance of models and the broad definition of the term *model* provided in Section 3.2.1, most Web engineering methods can be considered model-based. However, many methods have also defined a graphical notation to specify their models.

Generally, research approaches put their focus on three main aspects if it comes to modeling. The first aspect concerns the conceptual modeling of the application's data. A corresponding model is usually named "Conceptual Model" or "Data Model". The second aspect is modeling the navigation structure of the application with a "Hypertext Model" or a "Navigation Model". Finally, the last aspect concerns the presentation of content and is captured with a "Presentation Model" or a "User Interface Model".

The term "Web Engineering" was coined by Murugesan et al. [MDHG99] in 1999. The authors also provided a rather broad definition of the term, which is reflected in Definition 6.

Definition 6 *`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.*

By the end of the 1990s, it became clear that the development of Web sites and hypermedia applications were at a stage software development was at the time of the software crisis [LL07]. Most hypermedia applications were developed in an ad-hoc manner without relying on a structured development process or on proven development techniques. As a result, the quality of hypermedia applications was poor and development projects caused enormous costs and missed planned deadlines. The obvious analogy to the history of software engineering was quickly recognized. The terms "Hypermedia Crisis" [LH99] and "Web Crisis" [MDHG99] were coined.

Towards the end of the 1990s a steadily increasing number of Web sites have incorporated interactive features and evolved into Web applications. It became clear that the WWW was starting to evolve from a purely informational medium into an application medium [MD01]. This fact was also reflected in most research methods that started to use the terms "Web Application Design" [DIMG95][SR98] and "Web Engineering" [MD01]. Also Web engineering methods started to allow for data manipulation operations [BCFM00][Tur02] and incorporated them into existing models.

The evolution of Web applications into complex systems supporting sophisticated business processes had its impact on the Web engineering community. Many Web engineering methods started to incorporate modeling techniques and modeling elements to support business processes [KKCM04]. To this end, research methods pursue two main approaches. First, existing models are extended with model elements that support the modeling of business processes [BCCF03][SL03]. Second, new models are defined to capture the business process requirements [BCCF03][KKCM03].

One of the latest trends in Web engineering is the emerging support for Rich Internet Applications (RIAs) [BCFC06]. Web applications are becoming increasingly complex systems and

the request-response paradigm prevents traditional Web applications from having a fast and flexible user interface. Therefore, RIAs utilize the AJAX technology set to implement user interfaces that hide the communication process between client and server from the Web application's user. The result is a flexible and highly responsive user interface and user experience that is comparable to that of traditional software.

3.2 Using Models

The development of Web applications is a complex activity. Usually, there are many human and non-human resources involved in achieving the goals of a certain development project. The project usually involves many different activities that have to be executed and coordinated. To this end, Web application development and software development in general employ a large number of different models. This section describes the relevance of these models for Web application development. First, Section 3.2.1 describes the notion of a model and Section 3.2.2 introduces several approaches for model categorization. Second, Section 3.2.3 elaborates on the role of models for software development in general and Section 3.2.4 introduces a set of models for Web application development in particular. Finally, Section 3.2.6 describes the benefits of code generation from formal models.

3.2.1 What is a Model?

A model is an abstract representation of an object, a system, a concept or another model. Its aim is to describe the existing or planned original containing only attributes that are relevant for a specific task. Models are ubiquitous in every day life. Architects usually create a blue print before they authorize the construction of a new building and in many cases they also create three dimensional computer models or even build a small plastic model of the planned building.

These models serve different purposes but they also exhibit some common characteristics that are typical for models. There is always an *original* that is copied or is going to be created using the model. Thus, the model is just another representation of this original. However, a model always offers a certain amount of *abstraction* in respect to the original. Otherwise, it would not be a model but an exact copy. For example, a blue print of a building omits most of a building's characteristics, e.g., the materials or the color of the paint that are used to construct and decorate the building.

3.2.2 Model Categorization

Models may be categorized in many different ways. One distinction that is frequently made [LL07][Tab06][BS04] addresses the general purpose of the model. A *descriptive* model always represents something that already exists. For example, a toy train is a simplified copy of a real train. On the other hand, a *prescriptive* model serves as a plan for something that is to be created, e.g., a blue print of a building.

In many cases, a particular system is described using different complementary models. Consider a traffic light system at a road crossing which is a distributed system composed of several units (see [BS04]). This system may be described with different models that focus on different

aspects of the system. A *structural* model may define the location of all system components at the crossing. Whereas a *behavioral* model may specify the behavior of components as contained units and their interaction. As a matter of fact, system behavior may be even captured with several models. For example, a state/transition model may define states and valid transitions between states for a single traffic light. Additionally, an interaction model may specify messages that are sent between components and actions that are executed upon message reception.

Another aspect of differentiation is the actual physical form and, if required, the formal notation that is used to create the model. Both physical form and notation may vary considerably. Consider the blue print of a building. The formal notation is a complex graphical representation containing geometrical figures, numbers, and text. However, the physical representation of a blue print may exist on paper, stored on a hard drive of a computer, or just in the mind of the architect.

3.2.3 The Role of Models for Software Development

Models are used for two main purposes in software engineering. First, *process models* describe the development process itself. These models define principles, practices, and activities that are to be followed or executed during software development. Second, *software models* describe the subject of the development process, which is the software system to be developed. Section 3.3.3 describes the Web application development process and introduces different process models that may be applied for Web application development. This section deals with software models and describes how they may be employed for different development activities.

Software models are usually created as a plan for a software system that is to be built. Thus, they are clearly prescriptive models (see Section 3.2.2). However, in some cases, models are created from an existing software system. For example, many model-based software engineering tools support the automatic reverse engineering of models from source code. In these cases, we deal with descriptive models that document an existing system. Furthermore, software engineering principles dictate that models should always be kept consistent with the implemented software system, thus models that have been created with prescriptive intention may also be used in a descriptive manner. Software systems may become extremely large and complex. To address this complexity, it is advisable to capture both their structure and their behavior with a number of different models. Finally, a piece of software has no physical representation thus it makes no sense to build a physical model for it. Therefore, software models differ rather in formal notation than in physical representation. Popular formal notations for software modeling employ a combination of text, graphs, and diagrams. Subsequent paragraphs elaborate the role of models for different activities of the software development process.

As a first major activity of a software development project, requirements for the planned system have to be captured. This activity must be carried out with reasonable care because errors made in this early phase may cost a lot of resources to fix later on. Ultimately, if project managers and developers do not manage to capture the requirements of customers correctly, the project will probably fail. Being so important, there is a extra sub-discipline of software engineering addressing problems of this field called *Requirements Engineering*. The requirements analysis activity must produce answers to the following set of initial questions. What is the general business purpose of the system? What particular functionality do customers need? How will end-users interact with the system? Answers to these questions must be discovered in a dialog between customers and members of the development team and documented using a

set of models. These models should capture at least three different views of the system [Pre05]. First, the *information model* defines the information domain and characterizes the system's data flow. To this end, it is captured what input data is accepted, what data is stored permanently, and what output data is produced. Second, a *functional model* defines what functionality the system provides. Finally, a *behavioral model* captures how the system interacts with its environment, e.g., how it reacts to end-user input. Optimally, these models utilize a notation that is appropriate for communication with the customers thus the models may serve as specification of the system.

Besides requirements engineering, design is another vital activity of the software engineering process. Its goal is to transform customer requirements into technical design models. The effort necessary to execute this task depends of course on the complexity of the planned system and partially on the quality of the models that have been created during the requirements analysis phase. If requirements have been captured with imprecise natural language descriptions, the design effort may be considerable. Otherwise, if requirement analysis has produced more formalized models, e.g., use-case diagrams, activity diagrams and, statechart diagrams, the transition to corresponding design models may be much more smooth. Ultimately, design modeling ought to capture three different aspects of the system [Pre05]. First, an architecture model defines a system framework that structures system modules, which implement the system's functionality. Second, interface models define the system's interaction with its environment, e.g., depict graphical user interfaces. Finally, a component model captures the detailed design of system modules that are to be integrated into the system architecture.

Design is the last activity of the development process that produces models. All further development activities, e.g., implementation or maintenance use models for different purposes and adjust them to changes, if necessary. Design models are a blueprint of the software system to be built. The most obvious way to use this blueprint is to let it serve as a plan for creating the source code of the designed software system. To this end, developers have two alternatives to choose from. First, they may produce the source code manually. Second, if the design models employ an appropriate formal notation, developers may use a tool to partially or fully generate the source code of the system. However, most development tools support only an initial code generation step, after which the code must be adjusted manually. A code generation approach that manages to keep models and code synchronized is preferable to a solution that provides only partial code generation support.

Testing is of course an activity that must play a fundamental role in any serious development process. However, testing is not an activity that *ensures* the quality of the produced software. Software quality originates from solid design and proper implementation. However, testing may be applied to *detect* design flaws or errors in the source code. Ultimately, the elimination of identified defects leads to increased software quality. Models may be employed for testing in two ways. First, they may be used as documentation to easily identify components that must be tested. A comprehensive overview of the system architecture also allows to identify critical spots of the system, e.g., important internal and external interfaces. Second, models as a formal definition may be used to automatically generate test cases. Of course, generated code for testing must be manually adjusted. Creating reasonable test cases always requires a certain understanding of the components to be tested that must be provided by an expert.

Finally, models play an important role for the operation and maintenance phase of the software life cycle. Naturally, during this phase it is more easy to fix errors or add new components to the software if a comprehensive documentation is available. Of course, this is only the case

if the models are kept up-to-date with the implementation. However, this is a difficult task because it requires additional effort. Again, a development method that automatically ensures that models and code are kept synchronized is of great advantage.

3.2.4 Modeling Web applications

The previous section explained how modeling may be utilized for software development in general. Note that most presented aspects are also valid for Web application development. However, Web applications do have some unique characteristics (see Section 3.3.1) that introduce additional requirements, not only for the development process but also for the models and the modeling techniques.

In the Web engineering community, there is no consensus on a general process model that may be applied to Web application development [KPRR06]. Most experts agree that light-weight models are usually more suitable than their heavy-weight counterparts. This observation is described in detail in Section 3.3.3. However, the disunity is not restricted to process models. There is a considerable number of methods that have been proposed for Web application development over the last decade. Section 3.4 introduces a representative selection of methods, each of which present a somewhat different approach for developing Web applications. However, these methods also exhibit common characteristics. For example, many of these methods pursue a model-based development approach. The focus of this section is on the description of models that may be applied for Web application development. To this end, the following sub-sections explain which model types may be used for which activities of the development process. However, first a general categorization of models is provided.

Kappel et al. have introduced a simple model for capturing the scope of modeling activities for Web application development [KPRR06] (see Figure 3.1). The model defines the three dimensions *Levels*, *Aspects*, and *Phases*.

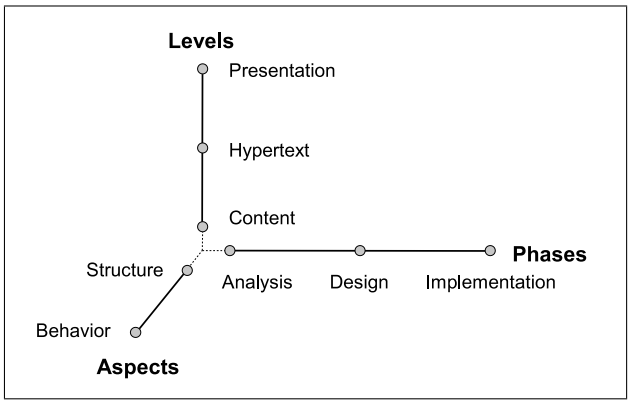

Figure 3.1: Dimensions of Modeling

The **Levels** dimension defines three levels that correspond to different aspects of Web application modeling. The *Content* level refers to models that capture the information domain and the business logic of the Web application. The *Hypertext* level refers to models that specify navigational characteristics of the application. Finally, the *Presentation* level defines the layout and final appearance of the Web application's user interface.

Of course, models that correspond to these levels may additionally have an extra focus on a certain aspect of a particular level. Therefore, the **Aspects** dimension splits each level into the

two aspects *Structure* and *Behavior*. Correspondingly, models of each level may have a focus on structure, e.g., a class diagram may capture the content structure of an application or on behavior, e.g., a state diagram may define the set of allowed states for one or more content objects.

Finally, the **Phases** dimension captures activities of the Web application development process. Correspondingly, prominent activities of the process, e.g., *Analysis*, *Design*, or *Implementation* are depicted as units of this dimension. Ultimately, this dimension amending the *Levels* and *Aspects* dimensions suggest that each model of a certain level or aspect may be created or used in different phases of the development process.

The following sub-sections provide a brief description of the three levels of modeling introduced in Figure 3.1. Note that an additional level that explicitly captures business logic operations is introduced in Section 4.4.

3.2.4.1 Content Modeling

Web applications are traditionally content-based applications. A vast majority of applications in the WWW today still focus on the presentation of content, e.g., text documents, images, videos, etc. To this end, most applications utilize a back-end system that is capable of managing large data volumes. Therefore, content modeling is an essential activity for developing Web applications. Without a proper content model it may prove difficult to build a Web application that satisfies the informational needs of potential user groups.

Content models capture content requirements of the Web application. Thus, models that belong to this category are usually created during the requirements analysis phase. As the *Aspects* dimension of Figure 3.1 indicates, content modeling deals with the structure and the behavior of the Web application.

The content structure of a Web application is usually captured with a *class diagram*. Primarily, a class diagram defines the object types that the Web application is going to support. For example, if during the requirements analysis phase for the Book Portal (see Section 2.2.3) developers identify the need for storing book authors, the class diagram of the application will surely contain the corresponding `Book` and `Author` classes. Additionally, the class diagram may define associations between content classes to reflect relationships between real-world objects. For example, the fact that authors *write* books may be expressed in the content model with a corresponding association between the `Author` and `Book` classes. Finally, class diagrams usually allow to group content classes into packages that correspond to modules of the Web application. There are several graphical notations that are appropriate to define the content structure of a Web application. Popular notations are the Entity-Relationship Diagram or the UML Class Diagram.

The second aspect considering the content of a Web application is modeling its behavior. Actually, content does not really *behave*, people do. Thus, the real question is, which operations may be executed by users of the system for which content objects and how these operations affect the state of content objects. These questions may be answered with several different models. First, operations that are related to a certain content object may be specified in the corresponding class of a class diagram. Basic operations simply manage the attributes of content objects, e.g., an operation *setTitle* of a book object may allow to update the title of a book in the content storage. Complex operations may combine several basic operations or may additionally implement custom business logic routines. Second, a *state diagram* may define a set of

states that are allowed for a certain content object. For example, the state *new* for a book object may indicate that the corresponding book is new on the market. Additionally, this diagram type specifies a set of transitions that bring an object from one state into another. A transition may be triggered by an operation and it may depend on different conditions. For example, an operation *toRegular* may change the state of a book to *regular* if the book has been in the state *new* for at least six month. Finally, the "behavior" of content may also be captured with a *sequence diagram*. This diagram type captures small usage scenarios and maps operations that are part of the scenario to content classes thereby creating a connection between the behavior and the structure of content.

3.2.4.2 Hypertext Modeling

The most significant characteristic of Web applications compared to traditional software applications is the strong reliance on hypertext. Usually, each page of the Web application's user interface contains a significant number of hyperlinks and the user may follow an arbitrary path along the hypertext structure of the Web application. This highly non-linear navigation has its advantages but also its drawbacks. One possible risk is that, if navigation paths are not carefully designed, the user may get quickly disoriented or even execute some actions unintentionally. To avoid this, the navigation structure and access to operations managing content must be designed properly. To this end, hypertext modeling addresses two different aspects. First, the modeling of the hypertext structure, also called navigation structure. Second, the modeling of navigation behavior, which defines the behavior of the application in response to navigation actions executed by the user.

Web application development methods usually employ a custom graphical notation for the definition of the *navigation structure*. Naturally, the graphical notation is in most cases a simple graph with nodes and edges. Nodes usually correspond to pages of the user interface and edges represent a navigation step between two pages. Additionally, the navigation structure model may employ a set of *access structures* that specify common navigational patterns. Access structure examples are a *menu* that contains a simple set of central hyperlinks, an *index* that allows the selection of an item from a set of similar items, and a *guided tour*, which guides the user through a predefined path usually visiting a set of similar nodes.

The second aspect of navigation modeling is the definition of navigation behavior. A user of a simple Web site is not able to interact with the content of the site. Thus, if it comes to Web site design, modeling just the navigation structure is sufficient. In contrast to that, Web applications provide different ways for the user to interact with content. In many cases, the user is allowed to search, create, or modify the content of the Web application. To this end, possible user actions are integrated into the navigation structure of the application. Thus, if the user activates a particular action during navigation, the Web application *reacts* correspondingly. This *navigation behavior* may take different forms. First, if the user navigates from one page to another, the Web application may pass along some parameters from the source to the target page. Subsequently, the application may use the parameter values to dynamically construct the target page. This is a rather passive interaction because in most cases the user does not even recognize the parameter transfer. Second, during a navigation step, the user may activate a certain operation that changes the content of the application. This is a more direct interaction that should be clearly indicated on the user interface of the Web application. Ultimately, user actions may have different scopes and may serve different purposes. An action

may be atomic or part of an interaction process. To this end, methods for Web application development employ a range of different proprietary model elements to express navigation behavior.

3.2.4.3 Presentation Modeling

Besides modeling content and navigation the third main activity supported by most Web application development methods is presentation modeling. The presentation model defines the user interface of the Web application. Accordingly, it is vital that this modeling activity is executed with appropriate care. No matter how accurate the other models are, the Web application's user is going to have great difficulties if the user interface does not provide a clear and intuitive view of the Web application's content and its navigation structure.

Similarly to the specification of content and navigation, presentation models have to take into account two aspects. The *presentation structure* that defines the layout of the user interface and *presentation behavior* that specifies user interface elements that allow users to interact with the application.

Models that define the structure of the user interface usually rely on a set of standard model elements that may be used repeatedly to construct the user interface. A central element of user interface modeling is a *page*. A page is usually populated with further elements that may be composite or atomic. Composite elements may contain subelements and create a substructure on the page. For example, most Web pages are divided into a navigation area and a display area. Thus, a structure model element that allows to define different *areas* on a page may become handy for user interface modeling. Atomic elements simply display a portion of the Web application's content or reveal a certain part of the navigation structure. To this end, the presentation model usually employs different *listings* and *object views* for displaying content and provides *links*, *menus*, or *breadcrumbs* to present navigation.

Another concern of presentation modeling is the specification of presentation behavior. As a matter of fact, this task includes three different aspects. The first one is access to content behavior. One task of the presentation model is to provide appropriate views for the user over the Web application's content. As described in Section 3.2.4.1, content behavior may be accessed and controlled through content management operations. Consequently, if the user should be able to manage the Web application's content, the presentation model must contain elements that provide access to these operations. Furthermore, there may be a correlation between the Web application's content and the way it is presented, thus a change of content may directly affect presentation behavior.

The second aspect that must be considered about presentation behavior is its relation to navigation behavior. Besides presenting content, another task of the presentation model is to provide a clear and intuitive view of the Web application's navigation structure. As described in Section 3.2.4.2, navigation links may have different semantics and accordingly their activation may have different results. Hence, the presentation model should support a set of different user interface elements that correspond to navigation links with different semantics. Similar to the relationship between content and presentation in many cases, there may be a correlation between navigation behavior and presentation behavior. A simple example is the transport of parameter values between two pages during a navigation step. The fact that the navigation step is a special one, i.e., it transports parameter values, is a characteristic of navigation behavior. However, if the presentation of the target page changes according to parameter values, this

reaction is a manifestation of presentation behavior.

Finally, presentation behavior may be completely independent of the Web application's content or navigational behavior. First, many Web applications allow the user to customize the appearance and the behavior of the user interface. To this end, the user may select from a set of options to configure an individual user profile for the interface. Furthermore, many Web applications support user groups, in which case for each group an individually pre-configured user interface profile may exist.

Besides defining structure and behavior, presentation modeling comprises a final task, which is rarely supported by Web application development methods, namely the modeling of the actual visual appearance of the user interface. Creating a certain look of the user interface (color scheme, font types, custom graphics, etc.) is a task for a graphic designer and usually not for a programmer. This task involves a certain amount of artistic creativity and providing modeling support for the actual process of creating the required look may prove difficult. However, a task that can be tackled is support for the management of style definitions, once they have been created by graphic designers.

3.2.5 Model Weaving

A common problem in model-based software engineering is that the models of choice, which are used to describe the same system from different points of view, are only loosely coupled. In this context, loosely coupled means that elements of different models that have some kind of a semantic connection are not bound to each other in a formal way. An example that demonstrates this problem is a simple user interface component of the Book Portal application that allows the portal's user to search for books. Let's assume that the user interface specification of the application includes the definition of a search page with a search form and a submit button. Lets also assume that another model, which specifies the application's operations defines a search operation that actually executes the search. The question is how to connect the elements of these two different models to each other. This question is not trivial because there is a range of aspects that have to be considered for such a connection. First, the user interface component has to be mapped to the appropriate operation. Second, if the operation requires parameters, they have to be mapped to the fields of the corresponding Web form. Finally, the operation's result has to be presented in some form, thus, it has to be mapped to an appropriate component of the user interface, e.g., a presentation component on a result page.

This example demonstrates that in some cases semantic connections between different models can become rather complex, thus if the models are intended to be used for code generation, these connections have to be formalized. The process of connecting different models in a formal way is called *model weaving*.

Many software engineering resources do not discuss model weaving explicitly [Som07][LL07] [SV05][CE00]. They assume that models are used as a blueprint for manual implementation, or as input for a code generator tool that creates separate portions of the implementation for each model which have to be manually integrated at the code level.

Pressman and Lowe [PL09] investigate the relationships between different models for Web application development from different perspectives. Examining the requirements gathering perspective they provide a list of questions that the developer should ask in order to identify connections between the Web application's content and functionality and in order to investigate how content is presented on the Web application's user interface. They do not provide a

self-developed solution for weaving different models. Instead, they refer to the Web application Extension for UML (WAE) approach [Con03] for connecting content and functionality and to the WebML for integrating functionality to the user interface. Unfortunately, the WAE employs a graphical notation that is rather high-level and cannot be used for direct code generation and the WebML integrates content management operations directly into the user interface specification instead of using a weaving approach. For a detailed discussion of the WebML approach, the user is referred to Section 3.4.4.

A model-weaving approach for Web applications is presented by Cicchetti et al. [CDRP06] [CDR08], who propose the utilization of two explicit weaving models that connect a data model to a composition model and a composition model to a navigation model using UML as graphical notation. Unfortunately, the approach has several disadvantages. First, it proposes two extra models merely for model-weaving purposes. Thus, the weaving models repeat model elements from other models and introduce a lot of redundancy. Second, mappings that are used in the weaving models appear to be rather coarse grained. Content entities are assigned to navigation nodes and navigation nodes are assigned to Web pages. A fine-grained specification of the user interface is not possible. Finally, this approach does not consider content management operations at all, which is nowadays a must for any Web engineering solution. Note that the flashWeb approach presented in this work avoids all these problems.

3.2.6 Generating Code

As described in Section 3.3.3.2, Web application developers are usually under time pressure during the development process. The importance and the competitive nature of the WWW force development teams to work in short development cycles and to deal with constantly changing requirements. Under these circumstances, any techniques that speed up development are more than welcome. One viable option to achieve this goal is to employ code generation technology.

Code generation relies on a formal specification of the system for that the implementation is to be generated. This specification usually contains several models that employ a set of standard elements to capture certain aspects of the system. As described in the previous section, methods for Web application development usually employ models to capture three different views of a Web application: content, navigation, and presentation. Depending on how powerful and well-connected the models are, a partial or complete generation of the implementation may be possible.

Partial code generation is a goal that may be achieved with most methods for Web application development (see Section 3.4). These methods usually employ several graphical models to capture different aspects of the Web application. Most of these graphical models may be expressed with an appropriate representation (e.g. XML) that may serve as input for a generator tool, which is capable of producing the source code of the application. However, most Web application development methods utilize models that cover only a certain part of the Web application's functionality. Additionally, in most cases the models are more or less isolated and semantic connections between them are not clear or not expressed in a formal manner. As a result only a certain part of the implementation can be generated.

The partial generation of the Web application may seem promising at first glance, however, there are many problems attached to it. The first and most important deficit is that the partial code must be supplemented by hand. For example, if a business logic operation should be

executed after the user activates a certain user interface element, the actual code for the call must be inserted into the generated source code. Generally, if the models that are used for code generation are not properly integrated, the connections must be manually supplemented after the generation process.

Another problem that arises from this first disadvantage and from the fact that Web applications are usually developed incrementally is the continuous need for the synchronization of models with generated source code. Partial code generation may be reasonable if there is a single development iteration. In this case, the models are initially used to generate a partial implementation and the missing parts are added manually. However, if the development process includes many iterations some new problems arise. Besides the described integration effort, developers have to make sure that newly generated components do not interfere with manually created ones. New iterations include new requirements that result in changed models and in changed generated code. Manually implemented components may be overwritten or invalidated by additionally generated code. An example is the simple renaming of a content model component, which often occurs if requirements change. All references to the component in the custom part of the implementation use the old name thus being invalid. Ultimately, the employment of partial code generation in a development process with a high number of increments is problematic. One has to keep track of any changes of the models between two increments very precisely. Subsequently, every piece of code, that has been created manually and is affected by the change of the models, has to be updated. This additional documentation effort may prove extremely time-consuming thus the advantage of the partial code generation approach may shrink or even disappear completely.

One alleged solution for keeping graphical models and generated code synchronized is the approach to re-engineer models from generated code. This feature is provided by some modeling tools for traditional software development. These tools analyze the source code of an application and create a graphical model from it. Theoretically, this approach may be used for each development increment of a Web application to re-engineer all models from the source code, then modify the models according to the requirements for the given increment, and finally, to re-generate the implementation. However, there are two major problems that make this approach impractical. First, re-engineering graphical models from source code is possible for some models, e.g., a class diagram of a content model, however, it is very difficult or even impossible for some others, e.g., a presentation model. Second, the layouting of graphical models after the re-engineering step has to be done manually. Automatic layouting algorithms are not able to reconstruct the layouting information (positioning, grouping of model elements) that is usually created by a modeling expert. This task may be very time-consuming for large models. Ultimately, partial code generation brings very little help for Web application development.

Compared to approaches that allow partial code generation, Web application development methods that manage to generate a fully functional Web application are clearly superior. Full code generation may be achieved if the models and extension facilities of the approach manage to specify the complete functionality of the application. Basically, there are two ways to achieve that. As a first alternative, the development method may employ models that capture the entire functionality of the Web application. This approach is impractical, because it requires models that are powerful enough to specify arbitrary functionality. However, models employed by Web application development methods are usually designed to specify common functionality. To capture unique features ,e.g., custom algorithms, the models lack the expressive power of

a programming language. Therefore, the development approach must provide an extension facility that allows for the specification of custom program code. Basically, there are two ways to accomplish this extensibility. First, the models may have custom elements that allow to specify custom program code. In this case, custom code is stored with the models and can be inserted into the source code during the generation process. Second, the generated source code may contain extensible areas that may be used to insert custom code into the implementation manually. Both approaches have advantages but some disadvantages as well. Storing program code in a model breaks its language independence. Thus, if the model is to be used to generate implementations for several target platforms, custom elements must support the specification of program code in all corresponding target languages. However, the developer has a very good overview of custom code and the code is better integrated into the models. Inserting the program code into the implementation after the generation step requires certain designated areas in the code where extensions may be added. The developer has to be careful not to leave these areas to avoid unintentional overwriting of custom code by the generator. However, inserting the code *in place*, i.e, where it belongs, has the advantage that the developer does not have to copy the code into the model. Ultimately, full code generation capability of a development method may speed up the development process considerably and works well with an iterative process model.

The gap between partial and full code generation is considerable. Theoretically, the extension mechanisms described previously, which allow to insert custom program code into the implementation, could be employed for an approach that supports only partial code generation. However, they are only practicable for small amounts of program code. Most model-based methods for Web application development are very far from supporting full code generation. They usually fail to properly integrate their models, which is a prerequisite for a functional Web application that has been generated automatically. As mentioned before, most development approaches provide at least models that capture the content, the navigation structure, and the presentation of the Web application. These models are expressed with different diagrams and can be used to generate a certain part of the application. For example, a code generator may produce the implementation of an object from a *class* component of a content model. The implementation may also define some operations that can be used to control the behavior of objects of this type. But how do these operations relate to other parts of the application? Which navigation steps may execute which operations? How does the user interface react to the execution? These questions stay unanswered as most Web application development methods fail to define semantic connections between content behavior, navigation behavior, and presentation behavior. The solution to these problems is model weaving. Models have to be designed to allow for integration with other models. This is a central aspect of the flashWeb approach presented in Section 4.

3.3 Engineering Web Applications

As established in Section 3.1, the development of Web engineering methods have been influenced by advances of the software engineering discipline. However, Web applications have some unique characteristics that must be accounted for. Therefore, existing models and development processes for traditional software are not always adequate to cover the needs of Web application development. To fill in this gap, existing methods and models have to be extended

or complemented by new ones.

This section provides an overview of the process of Web application development. First, Section 3.3.1 highlights some differences between traditional software and Web applications and points out consequences for the process of Web application development. Second, Section 3.3.2 introduces the software life cycle, a model describing different stages of a Web application during its development and application. Finally, Section 3.3.3 discusses whether standard software engineering processes are adequate for Web application development.

3.3.1 Conventional Software vs. Web Applications

Web applications are different from traditional software applications in many ways. Most unique features of Web applications can be directly derived from the nature of the WWW, which is an ubiquitous, constantly and rapidly evolving medium supporting hyperlinked content and applications. Subsequently, a selection of the most important characteristics of Web applications (see [KPRR06]) is presented.

Most Web applications strongly rely on hypermedia (see Section 2.2.1). This is natural as Web applications have evolved from hypermedia applications and most of them still heavily utilize hyperlinked content. Using hyperlinks creates a complex, non-linear navigation structure through the Web application's content. An obvious advantage of such a navigation structure is the great flexibility and speed for the user. However, there are some disadvantages, too. First, hypermedia may cause cognitive overload for the user because it is difficult to keep in mind all *semantic jumps* during the browsing process. Second, the navigation graph may be very complex thus the user may feel quickly disoriented. Traditional software applications usually do not offer navigation structures at this level of complexity.

Web applications are usually content-driven software systems. Primarily, the WWW is a medium for publishing digital information. Consequently, Web sites and Web applications usually have a strong focus on publishing information via documents. Therefore, Web applications are often described as document-centric. Furthermore, the presented content is always a key factor for attracting and satisfying users of the system. The expectation of users regarding the quality and topicality of content is often very high. In many cases, documents and articles that are published through the Web application are created by skilled or even professional writers. This focus on content is seldom among traditional software applications.

Additionally, the presentation of content plays a very important role for the success of a Web site or a Web application. If a user is convenient with the look & feel of the system, he is going to use it further. This willingness depends often on the fact whether an application uses stylish presentation components and up-to-date presentation techniques. Therefore, the development of a Web application involves not only system architects and programmers but in many cases also graphic designers, who ensure that the application gets a unique touch. Although usability and look & feel are also important aspects for developing traditional software components, in many cases there is a stronger focus on functionality compared to presentation.

Another unique characteristic of Web applications is their integration into a global application space, the WWW. Traditional software products are usually self-contained, stand-alone programs. Of course they may communicate with other applications in various ways but the core functionality of each program is well defined and is usually packaged into one or more executables that are executed on a single computer. In contrast to that, the execution of a Web application may involve many different server and client machines. Different server machines

may provide different parts of the application and a Web application may integrate the functionality of other Web applications (see mesh-ups); thus, the boundaries between applications are hard to define. As a result, Web application developers have to possibly consider a range of technical characteristics, e.g., APIs, authorization, or accounting of numerous existing systems.

The WWW as a platform for Web applications is a medium which is globally available around the clock. This ubiquity and constant availability brings along some chances but also some challenges for Web application developers. On the one hand, location and time information about the user may be utilized to provide location-aware and time-specific content and services. On the other hand, in a global context it is difficult to customize the application for a certain target group because of the heterogeneous social background of the Web application's users. Most Web applications are developed for the general public, thus making assumptions about age, nationality, and other cultural aspects of potential users is virtually impossible. In traditional software development, the target group for an application is usually more concrete.

Besides the social context Web applications are also determined by restrictions of a technological context. The WWW uses the infrastructure of the Internet, thus all technical aspects, e.g., bandwidth, server load, or features of an access device, must be considered. The main difficulty is the fact that these characteristics are mostly not known in advance. A potential user of the Web application may be connected to the Internet by cable and using a high-end desktop computer or he may want to access the WWW via a mobile phone, which has completely different characteristics regarding connection speed and screen resolution. Therefore, the Web application developer has to define an acceptable range for each parameter, within which the Web application functions normally. The Web application may even explicitly support different target platforms providing different user interfaces that consider the characteristics of the corresponding platform. However, support for different platforms is a significant development decision and has to be addressed while planning the development process.

Finally, the rapid development of the WWW is a general factor that keeps Web application development methods, processes, and tools at the state of constant change. First, the importance of the WWW as a global information medium exerts enormous competitive pressure on organizations and enterprises to deploy and maintain appropriate Web applications that support their diverse business needs. Consequently, requirements for Web applications are constantly changing and information has to be made available as fast as possible. Of course, this pressure is propagated to Web application development teams and manifests itself in constantly shortening development cycles and in time-consuming maintenance. A further necessity that originates from the rapid development of the WWW is the adaptation of new technologies (e.g. AJAX) allowing Web applications to remain competitive.

Besides all differences, there are also obvious similarities between Web applications and traditional software applications which are not discussed here in detail. However, there is an interesting trend in Web application development that is worth mentioning. Since the beginning of the new millennium, Rich Internet Applications (RIAs) are getting increasingly popular. This category of Web applications utilize the AJAX technology set to build user interfaces, which are comparable to user interfaces of traditional software applications, regarding the richness and responsiveness of the interface.

3.3.2 Web Application Life Cycle

During its development a Web application runs through different phases. In each phase, the focus is usually on one specific activity. For example, all development processes, which may be applied for Web application development, allow for an *implementation phase* in which components of the Web application are being programmed. Of course, it depends on the development process whether phases take place consecutively or whether there is a stronger focus on some of the phases compared to others. Furthermore, the process may also determine whether a certain phase may take place once or several times. Regardless of a certain development process, a Web application is at any given point in one or more of the following phases.

The `Requirements Analysis Phase` subsumes activities that are concerned capturing requirements for a planned Web application. Important activities in this phase are the identification, documentation, and validation of requirements. As a matter of fact, requirements identification is the most difficult part because customers usually find it difficult to precisely define their needs and expectations. Therefore, requirements have to be identified during an intensive dialog between developers and customers. Furthermore, compared to traditional software development, requirements analysis for Web applications is slightly more difficult. This is due to the fact that there is little knowledge about the target audience of the planned Web application (see Section 3.3.1). Another problem is the multi-disciplinarity of experts participating in the development process, e.g., writers, graphic designers, and programmers, who usually use different concepts and terms. Therefore, creating a coherent set of requirements that is well understood by all participants may prove difficult.

In the `Design Phase` requirements are used to create a technical plan for the required Web application. The main activities in this phase include the decomposition of problems into manageable units that lead to modules of the system, the definition of interfaces for module communication, and the definition of an architecture, which provides a framework for the conceived modules. Again, Web application development is somewhat different from the development of traditional software applications, because of the dominance of the client-server architectural setting. In most cases, the architecture of a Web application orients itself towards the client-server paradigm and defines business logic modules for the server side as well as for the client side. However, this orientation towards the underlying infrastructure is not mandatory. Architectures that abstract from such aspects (see Section 2.3.4.1) can be the better choice. Note that in the design phase graphical models may be employed to boost productivity (see Section 3.2).

In the `Implementation Phase`, the design is used to create the source code of the Web application. To this end, an appropriate programming language is selected that supports the requirements of the customers and the development team. In many cases, additionally to the programming language a Web application framework is utilized that provides a range of useful modules required by most Web applications. Note that some Web application frameworks suggest a certain architectural type, e.g., frameworks based on the MVC paradigm (see Section 2.3.5), whereas other frameworks allow for an arbitrary design. Therefore, in some cases the choice of a certain framework should already be made in the design phase. Depending on the usage of models and CASE tools the implementation phase may vary regarding required time and effort. Especially, the implementation phase may be minimized if the development team employs a model-based engineering method which is supported by a CASE tool capable of partially or fully generating the Web application's source code.

Testing, which is conducted in the `Test Phase` of the Web application development is the most important development activity for ensuring the proper quality of a Web application. Important activities during this phase include testing functional requirements, usability, performance, and security. However, testing Web applications is different from testing traditional software applications because of three main reasons. First, the content-driven nature of Web applications and the importance of content presentation requires a strong focus on these aspects. Second, the WWW as a platform for Web applications brings along a set of liabilities, e.g., unpredictable bandwidth or server loads and different end devices. Finally, uncertainties about the actual user group regarding age, experience, or cultural background of users makes it difficult to develop appropriate test procedures.

As mentioned before, one challenge of Web application testing is to check whether the presented content satisfies certain quality requirements. This is a time-consuming activity, as the content usually contains a fair amount of natural text, which can only be checked automatically to a certain extent. Another aspect unique to Web application testing is the integrity of the hyperlink structure. Current WWW technology does not support the automatic checking of hyperlink integrity thus the target of a given hyperlink may be removed at any time resulting in a broken link. However, there exist Web application frameworks that support automatic hyperlink checking and the integrity of the hyperlink structure may also be validated using automatic tools using crawling technology.

Further aspects that play an important role in Web application testing is usability and presentation. These two aspects are strongly related. On the one hand, a Web application should meet certain usability criteria, e.g., possess an intuitive navigation structure. On the other hand, each competitive Web application needs a compelling presentation, i.e., comply to certain aesthetic requirements. Of course, testing usability and presentation is pretty difficult. This is due to the fact that these aspects may be judged completely differently by any two individuals.

Last but not least, testing Web application performance is usually more challenging than testing the performance of traditional software applications. The additional complexity results from the technical diversity of the Internet and from the unpredictable usage profile of the Web application. Web application users may connect to the Internet via slow or fast connections, they may use end devices with varying computing power, and they may choose from a set of different Web browsers. Therefore, the Web application must be tested for multiple target environments with completely different performance characteristics. Finally, Web applications are multi-user applications. They may be accessed by thousands of users concurrently. Therefore, the Web application and the underlying infrastructure must be tested for different loads including extremes like sudden usage peaks.

The `Operation and Maintanance Phase` of Web applications is characterised by constant change. In contrast to traditional software applications, new versions of a Web application are published more frequently. This fact is due to the enormous competitive pressure that dominates the WWW. Thus, besides the mandatory activity of fixing errors, this phase is usually characterised by ongoing improvement of the Web application. The final phase of the Web application life cycle is the `Retirement Phase`, in which the application is removed from its runtime environment and gets archived.

Note that depending on the complexity of the Web application and the applied development process, the phases of the Web application development life cycle may occur in different constellations. However, most of the time they do not take place sequentially. Thus, the term *life cycle*, suggesting that the phases are ordered in a linear fashion, is somewhat misleading.

Actually, different phases may be emphasized or considered less important, they may overlap and even reoccur.

Heavy-weight software development processes, like the *Waterfall Model* [Roy87][Pre05] usually have a strong emphasis on requirements analysis and design. They employ a set of formalized documents capturing these aspects which are time-consuming to create. In contrast to that light-weight, agile development processes like *Extreme Programming (XP)* [Bec04][Pre05] focus on the implementation phase and try to avoid the overhead of creating large formalized documents.

As already suggested, different phases of the Web application life cycle seldom take place sequentially and may overlap. This has several reasons. First, the rapid evolution of Internet technology and the competitive pressure in the WWW ensure that requirements for a Web application change constantly, i.e., also during development. Second, testing is a central activity that is crucial for the success of the development project, thus it should overlap with other phases, e.g., design and implementation. Finally, as a response to competitive pressure and short development cycles, it is common practice to develop several versions of the Web application in parallel. In this case, phases that take place concurrently are natural. For example, the design and implementation phases of a basic version may be ongoing during the requirements analysis phase of an extended version.

Finally, development processes that advocate iterative development, e.g., the *Spiral Model* [LL07], build on reoccurring phases of the Web application life cycle. In such cases, the Web application is developed in several cycles, each of which includes the same set of phases, e.g., requirement analysis, design, implementation, and testing. Cycles are repeated until the Web application is mature.

3.3.3 Web Application Development Process

In Section 3.3.1 differences between traditional software applications and Web applications have been elaborated. These differences point out the need for a development process that takes into account the specifics of Web applications. Unfortunately, there are no established process models for Web application development. Web application development projects usually rely on a process model for traditional software development or do not employ a process model at all. Only a small percentage of projects employ a specialized development method that is tailored to Web application development [LF05].

The focus of this work is not to define a process model for Web application development but to introduce a development method that can be used in conjunction with any process model. To this end, the following sub-sections describe some characteristics of the Web application development process. Section 3.3.3.1 introduces the term *development method*. Second, Section 3.3.3.2 introduces a set of characteristics that are typical for Web application development. These characteristics are vital for the choice of the right process model. Third, Section 3.3.3.3 discusses the suitability of heavy-weight and light-weight process models for Web application development.

Note that Section 6.3 contains a discussion of the flashWeb method's support for different development process models. The discussion is based on the terms and explanations of this section.

3.3.3.1 Methods for Web Application Development

The term *development process* subsumes all activities that concern the development of an application. The development process usually follows a *process model*, which is a plan that defines which development activities are to be executed and which resources (human and non-human) are to be used to achieve the goals of the development project. The process model usually defines the order of activities and in many cases a set of milestones that are to be reached at some point during development. Many process models define certain roles that are embodied by one or more members of the development team and are assigned to appropriate activities. After all, the main purpose of a process model is to organize and coordinate the development process.

In contrast to a process model, a *development method* is not concerned with the organization of the overall development process but provides detailed recommendations, instructions, and tools that help to achieve well-defined development goals. For example, a development method may suggest that the application should be developed using object-oriented concepts, a certain graphical modeling notation to capture the design of the Web application, and a certain CASE-Tool that supports the creation of graphical models. Thus the main goal of a development method is to provide techniques and tools for the execution of certain development activities, e.g., the creation of development artifacts like source code and documentation. An optimal development method should be flexible enough to be used in conjunction with different process models. However, this is not always the case as sometimes it is hard to determine the boundary between a process model and a certain development method. For example Extreme Programming postulates that source code should be developed in pairs. The actual *idea* that any given piece of code may only be created or altered if two developers simultaneously work on it, is clearly part of a development method. However, the *decision* that this method is the only allowed way to work on source code in the project is part of the process model [KPRR06].

3.3.3.2 Characteristics of Web Application Development

The WWW and particularly Web applications are getting increasingly important for the success of any enterprise. This is especially true for companies that directly rely on the WWW as a medium for business transactions. Therefore, the success of Web development projects is in many cases vital for the success of an enterprise. Accordingly, the choice of an appropriate process model that takes into account the specifics of Web application development is imperative. This section introduces several typical factors [KPRR06][Pre05] that determine Web application development.

Immediacy and Continuous Evolution Due to the importance of the WWW as a communication and application medium and the fierce competition in the WWW, enterprises are forced to launch or update their Web applications as soon as possible. This necessity of *immediate presence* on the WWW puts development crews under a lot of pressure, as they are forced to work in *short development cycles*. To make things worse the rapid development of content and technology and the necessary adaptation to social and cultural impacts in the WWW cause a *frequent change of requirements*. As a result, the development process is rather *communication intensive* as customers need to often articulate their additional requirements, which results in an increased

communication effort for the development team as team members have to synchronize their activities frequently.

Incremental and Parallel Development The enormous time pressure and frequently changing requirements call for a process model that supports *incremental and parallel development*. The incremental approach ensures that the Web application is developed in several increments. Usually, the first version of the Web application is an architectural prototype, which contains the main building blocks of the application but does not provide detailed functionality. The second increment may be a first publishable version of the application and further versions concentrate on providing the missing detailed functionality. Incremental development supports not only changing requirements but also a short time-to-market period. First, early versions of the application are usually general enough to support requirements that emerge late in the development process. Thus, there is a good chance that the new functionality can be included into the application at low cost. Second, vital functionality of the Web application may be already included into early versions, thus, a first version may be published early in the development process.

A further aspect that characterizes Web application development is the parallel execution of activities. On the one hand, parallelism is natural when it comes to developing Web applications because of the diverse disciplines that are involved in the development process. User interface developers may work together with graphic designers to create the user interface of the Web application, whereas data modeling experts and programmers may work on core business logic. On the other hand, parallelism may be introduced artificially into the development process to get the necessary work done faster. To this end, several small groups can work on similar problems to increase development speed. Sometimes it is even the case that different versions of the application are developed in parallel.

Reuse and Integration Web applications usually provide a common basic set of features. For example, many Web applications provide user authentication, employ sessions for the management of user interaction, or maintain user profiles that support user groups with different requirements. Additionally, there may exist a set of further functionalities that are enterprise specific and are necessary only for Web applications of a certain enterprise. Anyway, common features should be packaged into modules that can be reused in several development projects. Furthermore, it is advantageous to utilize a Web application framework that brings along ready-to-use modules for common functionality. Through reuse of components the development effort may be minimized.

Another aspect that is typical for Web applications is the integration of existing data or the integration of the enterprise's legacy systems. In the early days of the WWW, first large Web sites were merely read Web front-ends to some enterprise database. Over time, these Web sites have evolved into Web applications that not only support read access to enterprise data but also allow data modification. Although databases still play a very important role for Web application development, data integration is just one side of the coin. The rapid development of the WWW has brought along technologies that enable Web developers to build powerful Web user interfaces and to integrate virtually any kind of enterprise legacy system into the Web. During the integration process of a legacy system, a certain part of the system's functionality may be reprogrammed and integrated into the Web application, whereas other parts

remain in the legacy system to allow the Web application to serve as a wrapper to such legacy functionality.

Adapting to Complexity Web applications are rapidly evolving systems. As already established, the competitive pressure in the WWW forces developers to work in short development cycles and the problem of frequently changing requirements is answered by incremental development. However, permanently arising additional requirements let the Web application grow continuously. Consequently, the complexity of the system increases steadily. As a matter of fact, a Web application that starts with a minimalistic first version, may develop into a formidable system with a large feature set. This evolving complexity affects several development activities. First, the design of the system may require modification as the system architecture must support the additional requirements. To this end, additional system modules have to be inserted into the design or existing ones have to be modified or merged. Second, the implementation of the Web application may need some refactoring as a result of the changed design. Third, the deployment strategy for the Web application may need adaptation as the altered implementation or the need to support more simultaneous users may require an improved infrastructure. After all, the development process model must support the changing complexity level of the Web application. It must allow reoccurring development activities and must be able to manage human and non-human resources in a flexible manner.

3.3.3.3 Heavy-weigth vs. Light-weight Processes Models

Until the 1990s, traditional (heavy-weight) development process models dominated the field of software engineering. Process models of this category, e.g., the Waterfall Model or the Rational Unified Process (RUP) [JBR99] emphasize the importance of planning and documentation. The development process is usually divided into several phases, which are scheduled at the beginning of the development process. Each phase has a specific focus and a set of goals that are to be reached before the transition into the next phase is allowed. The end of a phase is usually marked by a milestone, which defines the goals of the phase, i.e., specifies which artifacts (documentation, source code) have to be brought to which state. After passing through all phases, the product has been developed and the development process ends.

For example, the RUP defines the four phases *inception*, *elaboration*, *construction*, and *transition*. In the *inception* phase a vision of the product is developed and basic usage patterns of the application are captured with a use case model. Also, a simplistic architecture of the system is drafted in this phase. Additionally, most important risks of the process are identified and a rough schedule for the project is created. The *elaboration* phase concentrates on completing the use case model and designing and implementing the basic system architecture. At the end of the phase, the project manager should be able to plan all necessary activities and estimate the required resources for the project. The *construction* phase is the most resource-intensive phase of the RUP. The focus is on implementing all requirements that have been captured with use cases. Basically, the architecture that has been implemented in previous phases is filled with modules that correspond to all customer requirements. Last but not least, the goal of the *transition* phase is to test and transfer the product to the customer. To this end, a small number of beta-testers are asked to test the system. Defects that are revealed during beta-test are scheduled for correction.

Heavy-weight development process models are primarily suitable for the development of

large critical systems, which are developed over several years. However, they may be the wrong choice if it comes to the development of medium or small systems. In such cases, a strictly defined process and the numerous documentation artifacts that are required by a heavy-weight process model may constitute an unnecessary overhead. Since the end of the 1990s, a number of agile (light-weight) process models and development methods have been proposed that try to circumvent the drawbacks of heavy-weight approaches.

Agile approaches follow a set of principles that have been articulated by advocates of agile software development in a manifesto [URI08i]. The manifesto formulates some basic principles that differentiate agile process models and methods from traditional approaches. First, it states that *individuals and interactions* are more important than processes and tools. Second, *working software* is valued more than comprehensive documentation. Third, *customer collaboration* is preferred over contract negotiation. Finally, *responding to change* is regarded superior to following the plan. Ultimately, as these principles suggest agile approaches try to get rid of a strict and rigid development plan and the burden of extensive documentation and concentrate on individuals and collaboration to be able to respond to changing requirements.

Perhaps the most well-known agile software development approach is Extreme Programming (XP) [Bec04]. This approach postulates the four core values *communication*, *simplicity*, *feedback*, and *courage*. The first value, *communication*, emphasizes that a direct dialog between developers and customers and also among developers is considered more important than writing and sharing documents. *Simplicity* is paramount if it comes to designing the system or writing code. Simple solutions are easier to create, understand, and maintain. The third core value, *feedback*, is the main activity of quality assurance in XP. Continuous customer feedback ensures that the product corresponds to customer requirements. Last but not least, *courage* is the fourth core value and postulates that developers should not be afraid of trying new ideas and unconventional solutions.

Additionally to the core values, XP defines about a dozen of good practices that should be applied during development. As an excerpt, three of these practices are introduced here. The first example is *test-driven development*. In a development process that follows the XP model tests are developed and implemented before actual code for the product is written. All tests are evaluated each time new functionality is added to the system. The second example is *collective code ownership*. This means that all developers of the project are responsible for the complete code. Consequently, every developer may work on any part of the system at any time. The advantage of this approach is increased development speed as any programmer of the team may fix a given problem or extend an arbitrary part of the system. The final example of XP practices that are mentioned here is probably the most well-known characteristic of this approach, which is *pair programming*. This practice suggests that at any given time any two programmers of the team may create or alter any piece of code of the system, but never alone. This approach has several advantages. First, it helps to achieve the goal of collective code ownership, as at least two programmers are familiar with a given part of the system. Second, it facilitates and encourages communication, which is a core value of XP. Finally, it helps to improve the performance of single developers as they may learn a great deal from their colleagues.

Of course, XP does not merely define some core values and good practices but also defines activities like design, testing, or coding, and suggests a workflow of activities for each iteration of the project. For a full description of the method, the reader is referred to [Bec04] and [Pre05].

3.4 Model-based Web Engineering Methods

Over the last decade, a variety of methods for Web application development have been proposed. This section introduces a comprehensive and representative selection of these methods. The Book Portal example from Section 2.2.3 is used to illustrate how graphical models are employed by these methods to define a Web application. Note that these methods are not fully-fledged Web application development process models (see Section 3.3.3.1). Usually, they describe how different development activities are to be executed and provide different models for Web application design. However, there are a lot of aspects, e.g., project management, resource planing, communication handling, etc., that have to be considered by a development process model, which these methods fail to address.

3.4.1 Relationship Management Methodology (RMM)

The Relationship Management Methodology (RMM) [ISB95] [DIMG95][IKK97][IKK98] is a methodology for the design of hypermedia systems. It focuses on the modeling of complex information domains and provides graphical models and guidelines for the construction of Web information systems. As the name of this methodology suggests, RMM focuses on the management of relationships between information entities. The initial work on the RMM was published in 1995 [ISB95] and, at that time, Web applications as we know them today were non-existent. Consequently, the RMM supports the development of informational systems, i.e., Web sites that do not support content management functionality. The RMM also provides a CASE tool [DIMG95] that supports creating the graphical models.

The RMM suggests a set of development steps that may be carried out sequentially throughout a Web application development project. The first step is the analysis of informational and navigational requirements, however, this step is not explicitly supported by the approach, i.e., RMM does not provide a model or any guidelines for this activity. After *Requirements Analysis*, RMM defines three core development steps that are supported with graphical models and detailed instructions. The *E-R Design* step utilizes a standard Entity-Relationship Diagram [Che76] to model the information domain that underlies the Web site to be built. The *Entity Design* step employs so-called *m-slices* to define complex views over information entities. Finally, the *Navigation Design* step uses the *Relationship Management Data Model* that specifies a set of modeling primitives for defining the navigation structure of the information system. Additional to these core modeling activities, RMM suggests three further development steps that concentrate on the actual implementation of the system utilizing the core RMM models. The *User-Interface Screen Design* step defines concrete user interface elements to implement the views that have been created in the *Entity Design* step. In the *Construction* step, the information system is implemented. Finally, as the name suggests, in the *Testing and Evaluation* step the implemented Web site is tested and evaluated. Note that the RMM methodology concentrates on the three core activities mentioned before, which are explained in detail in the following sections. Further information about the development process proposed by RMM may be found in [IKK98].

3.4.1.1 E-R Design

The RMM employs a standard Entity-Relationship diagram to model the information domain of the system. This graphical notation was selected because it is a well understood and doc-

umented model that is primarily used to define the data model of relational databases, which often serve as storage backend behind Web information systems. Figure 3.2 depicts the Entity-Relationship diagram for the Book Portal using the graphical notation employed by RMM.

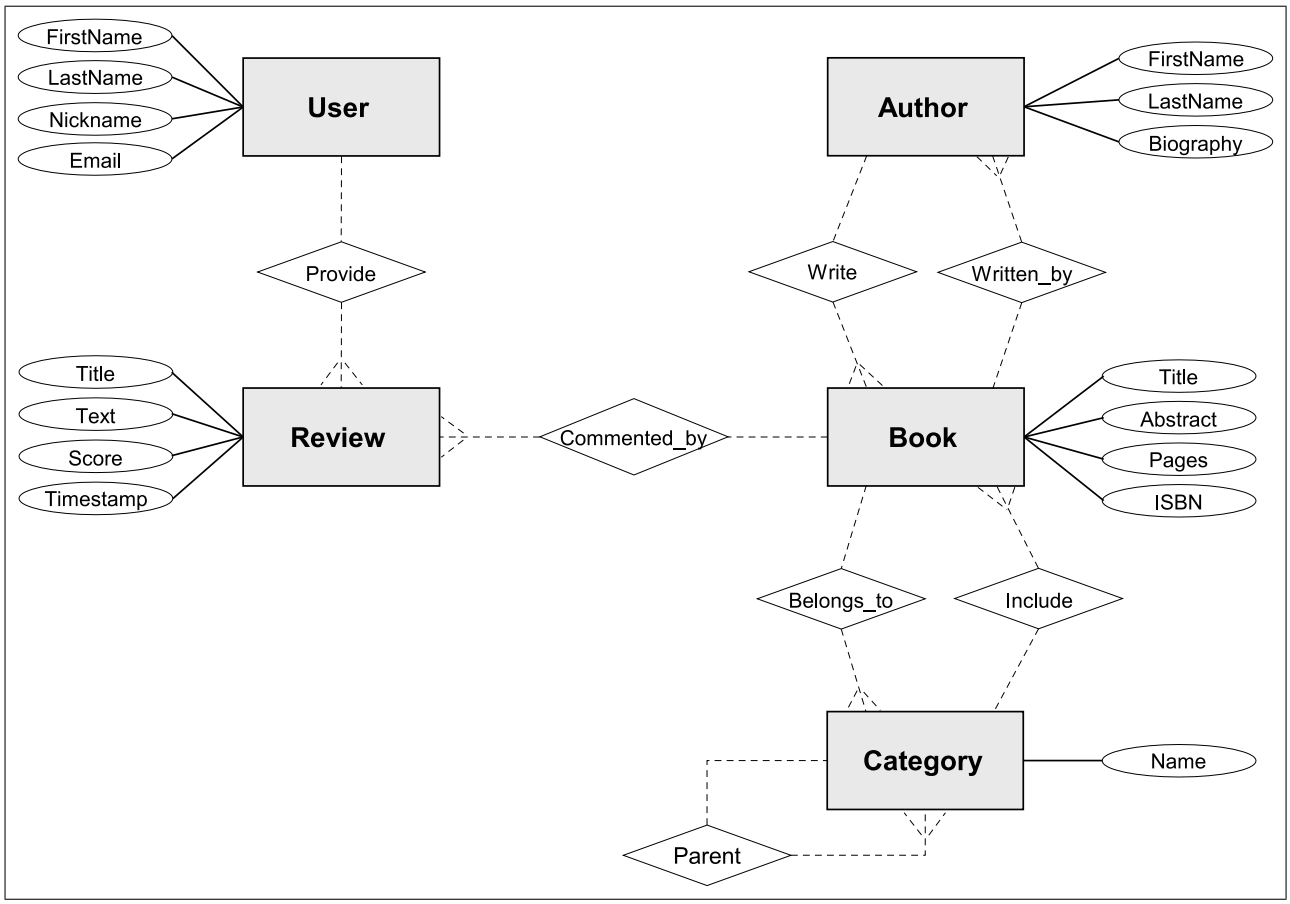

Figure 3.2: E-R Diagram of the Book Portal for RMM

Entity-Relationship modeling is well documented in various resources [Che76][SKS06][Dat04], thus, subsequently only specialties of the RMM are highlighted without explaining the E-R modeling approach in general. The RMM employs only a subset of the functionality that is provided by E-R modeling, thereby unnecessarily limiting the capabilities of the approach. It does not allow specialization or multi-valued attributes, which are important to reduce the complexity of the E-R model. Note, for example, that the `User` and the `Author` entity types both possess the *FirstName* and *LastName* attributes, thus it would be appropriate to capture these attributes with a `Person` entity type and use the *is-a* specialization relationship to define specialized person entities. Also note that RMM requires to split all many-to-many relationships into two one-to-many relationships to prepare the E-R model for further processing. Relationships between entity types are used in navigation modeling to derive the navigation structure of the Web site. This may be observed looking at the *Write* and *Written_by* one-to-many relationships between the `Author` and `Book` entity types that may have been modeled with a single many-to-many relationship.

Another fact that limits the expressive power of the RMM approach is the lack of data types in E-R modeling. Development methods that support data types may introduce this additional information early into the design process. Finally, E-R modeling does not support attributes

that describe relationships. This is a disadvantage compared to some object-oriented modeling notations, e.g., UML. For example, it would be more appropriate to specify the *CommentDate* attribute as an attribute of the *Commented_by* relationship than as an attribute of the `Review` entity type. Afterall the *CommentDate* attribute should denote the date on that a specific review was associated with a specific book.

Also note that the E-R notation used by RMM is somewhat different from the standard notation defined by Chen [Che76]. The RMM method denotes one-to-many relationships between two entity types with a dashed edge that opens up into three lines at the “many" end of the relationship.

3.4.1.2 Entity Design

The aim of the *Entity Design* step is to define views over one or more entity types and to use these views to display information on Web pages. The original RMM proposal [ISB95] introduces the notion of a *slice* that is a grouping of selected attributes of a single entity type, i.e., a special view of the entity type focusing on certain aspects. For each entity type, an arbitrary number of slices can be defined and connected by so called *structural links*. These slice definitions are implemented as interlinked Web pages that allow the user to browse different views of a certain entity. However, this original proposal proved to be too limited for Web site design. One obvious problem is that Web pages that correspond to slices are rather simple and cannot contain any access structures leading to other entities. Furthermore, this approach does not allow to present information from different entities on the same Web page.

An extension of the original proposal introduces the notion of so called *m-slices* [IKK97] that replace *slices* in RMM to provide a more powerful way for the definition of views over entities. The graphical notation of m-slices is explained by means of the `Book` page from the Book Portal example (see Section 2.2.3.3). Figures 3.3 and 3.4 provide m-slices that are necessary to capture the information that is shown on the `Book` page.

An m-slice is a collection of attributes, access structures, and other m-slices that allow the recursive building of arbitrary views over one or more entity types. The graphical notation of an m-slice contains an entity component (rectangle with rounded corners) and a slice component. The entity component specifies the name of a basis entity type (top left corner) and the slice element provides the name of the m-slice (at the bottom) that is used to identify an m-slice if it is included in another m-slice. Information that originates from the basis entity is placed into the intersection of the entity and the slice. Example a) in Figure 3.3 defines a simple view over the `Category` entity type. The *Name* attribute of the `Category` entity type is placed into the intersection of the entity and the slice because it is an attribute of this entity type. This simple m-slice contains the name of a `Category` entity. Example b) is a comparably simple definition of an m-slice that provides the name of an `Author` entity composed of the *FirstName* and the *LastName* attributes. As mentioned before an m-slice may contain other m-slices and also navigation access structures. Example c) defines a view over the `Review` entity type that contains the review’s title and also the name of the user that created the review. To this end, this m-slice includes the *Title* attribute of the `Review` entity type and the *User Name* m-slice. Note that the *User Name* m-slice is placed into the slice area that does not intersect with the area of the entity. This denotes that this m-slice contains information originating from an entity other than the basis entity, i.e., this m-slice shows information from a `User` entity and not from a `Review` entity. The *User Name* m-slice is connected with a solid edge to the basis entity. The label of this

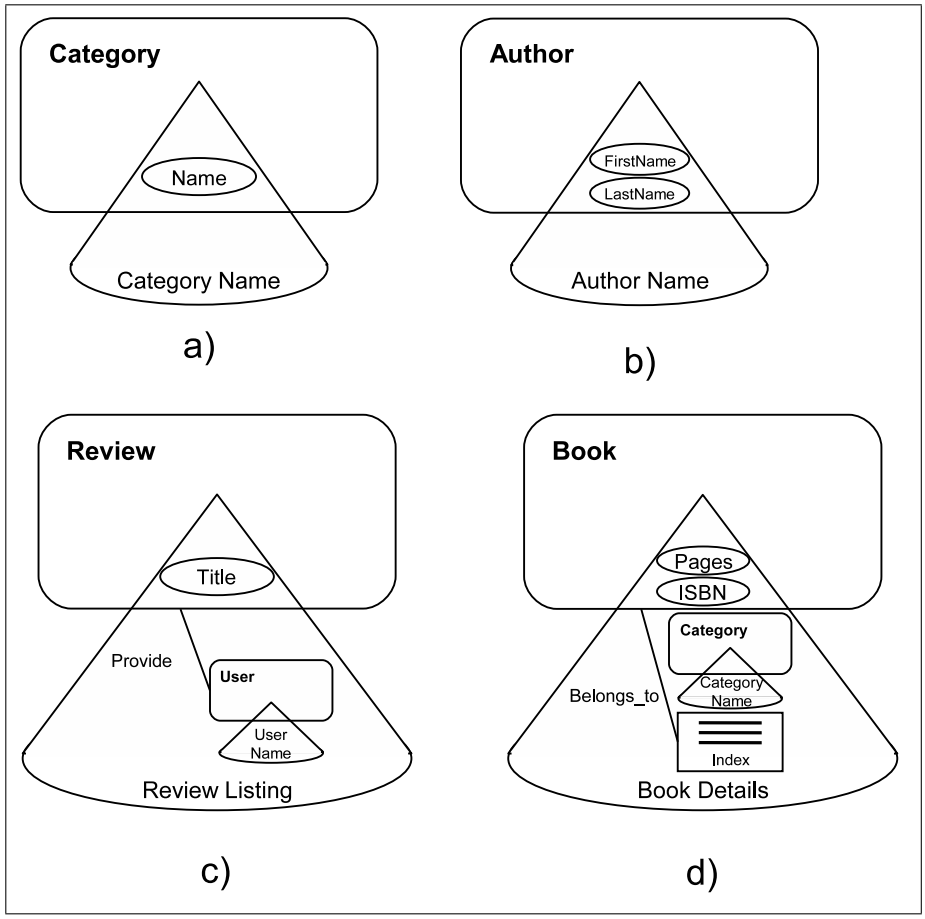

Figure 3.3: Basic m-slice Examples for the `Book` Page

edge indicates that the `User` entity is accociated to the `Review` entity by the *Provides* relationship. Note that the *User Name* m-slice itself is not shown in Figure 3.3, however, it can be built similarly to the *Author Name* m-slice. Finally, example d) depicts the *Book Details* m-slice that, as the name suggests, provides details about a `Book` entity. To this end, this m-slice contains the *Pages* and *ISBN* attributes of a book and also an index of the book's categories. Note that the category index uses the *Category Name* m-slice that is depicted in example a) and that the edge between the category index and the entity refers to the *Belongs_to* relationship between the `Book` and the `Category` entities. Figure 3.3 shows four m-slice examples with increasing complexity that define information units for the `Book` page of the Book Portal example. Figure 3.4 presents an m-slice that defines the information content of the entire `Book` page.

The `Book` page (see Section 2.2.3.3) shows complex information about a `Book` entity including simple attributes, different views of related entities and several navigation structures. To this end, the *Book View* m-slice employs some of the previously defined m-slices from Figure 3.3. First, the *Title* and *Abstract* attributes of the `Book` are included as these attributes are to be directly shown on a corresponding page. Second, the *Book Details* m-slice is included to provide detailed information about the book. Third, an author index is specified using the *Author Name* m-slice and the *Written_by* relationship between the `Book` and the `Author` entities. Finally, a review index is defined using the *Review Listing* m-slice and the *Commented_by* relationship between the `Book` and the `Review` entities.

Note that the *Entity Design* step of RMM may specify for each entity an arbitrary number of

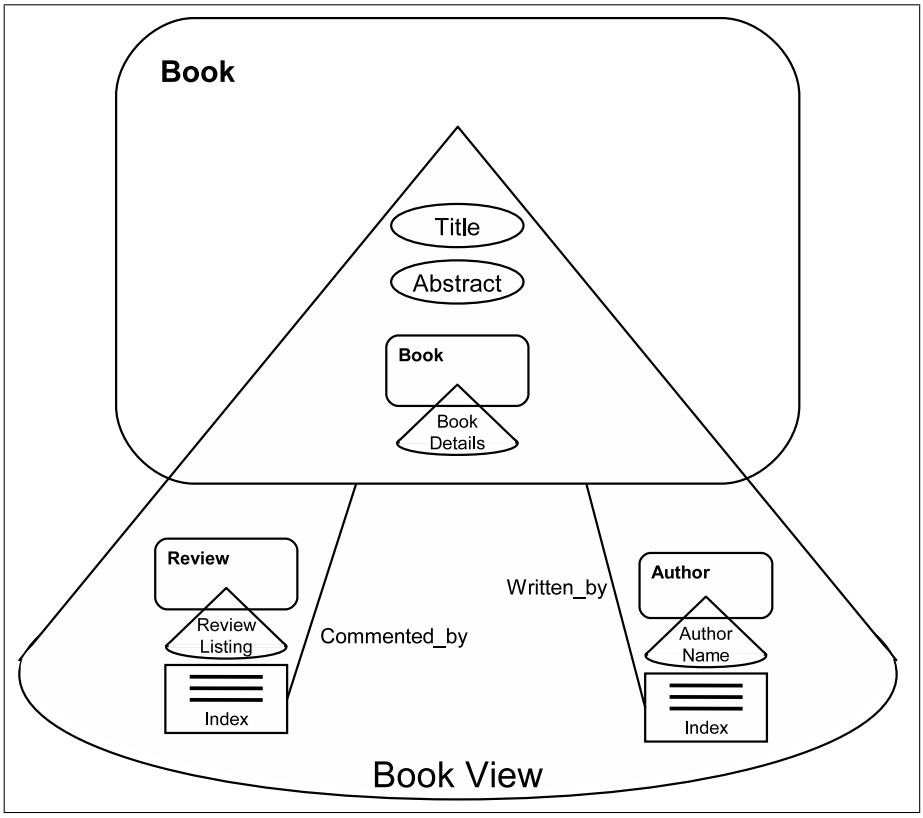

Figure 3.4: Design of the `Book View` with RMM

m-slices that define information units with different complexities. Each m-slice is defined only once and may be used in an arbitrary number of other m-slices. Any m-slice (simple or complex) may be used to create a page of the Web site that is modeled with RMM. Note however that m-slices merely define what information is to be shown and not how this information is to be presented.

3.4.1.3 Navigation Design

The aim of the *Navigation Design* step of RMM is to define the Web site's navigation structure. Navigation design with RMM strongly relies on the content model created in the *E-R Design* step of the development process. Navigation paths may be only created corresponding to relationships between entities or to navigate between different views of a single entity. For example, the navigation model may define access structures between a `Book` and `Author` entities only if the content model includes a relationship between these two entity types.

The RMM provides a set of graphical modeling primitives that may be combined to build a navigation model. The most basic elements for navigation modeling are the *Unidirectional Link* and the *Bidirectional Link*, which are used to specify navigation paths between different user interface pages defined by m-slices. An unidirectional link creates a navigation step between two pages that can be traversed only in one direction. In contrast to that, the bidirectional link allows navigation in both directions. Of course, RMM also allows for the definition of more powerful access structures. The *Conditional Index* specifies an index to a selected set of entities of the same type. A condition may refer to a relationship between two entities, for example, to create an index of authors on a book page. A *Conditional Guided Tour* defines a

guided tour that analog to the index may restrict the set of presented entities with a condition. Finally, a *Conditional Indexed Guided Tour* is an element that combines an index and a guided tour to create an access structure that allows to navigate to selected entities and also to traverse entities in a linear fashion. The condition may refer to a relationship just like in the case of an index. Note that a condition may also be empty so that the corresponding navigation structures may allow access to the complete set of entities of a certain type. The last model element for defining navigation is the *Grouping* element, which allows to group an arbitrary number of the previously introduced access structures. Figure 3.5 depicts a so called *Relationship Management Diagram* that defines the navigation structure of the Book Portal example application using the described modeling elements.

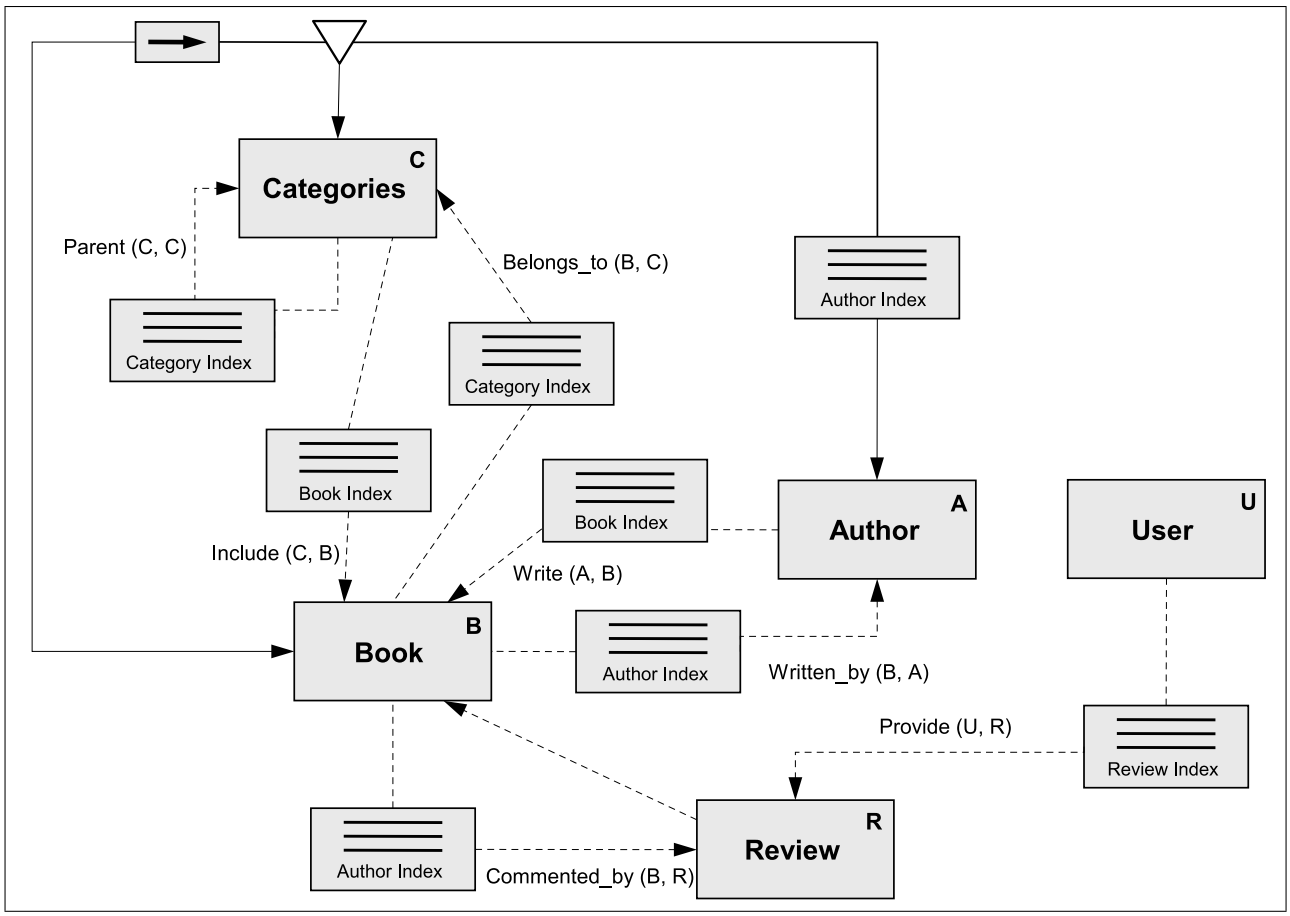

Figure 3.5: Relationship Management Diagram of the Book Portal for RMM

The main entry point to the navigation structure of the Book Portal Web site is defined by a *Grouping* (upside-down triangle), which includes a link to categories, an index of authors, as well as a guided tour (small rectangle with an arrow) of books. Additionally, the diagram defines a set of indexes that are derived from relationships between entities. For example, there are the *Write* and *Written_by* relationships between the `Book` and `Author` entities that are expressed with corresponding indexes. Note that these indexes utilize conditions that refer to the corresponding relationships in order to restrict entities that can be visited using the indexes. An index that does not specify a condition is, for example, the index of authors included in the *Grouping* element.

Unfortunately, the limited expressive power of RMM for navigation design does not allow to

define the complete Book Portal example. The relationship management diagram may contain only entities just like the E-R diagram for content modeling and does not employ the notion of a user interface page. Allowed access structures are derived from relationships between entities, thus the definition of custom access structures is not possible. There are several parts of the Book Portal example that cannot be expressed with an relationship management diagram. First, the definition of user interface pages that do not correspond to an entity type is not possible. Thus the `Authors` or the `Search` pages cannot be modeled. This results in an overloaded navigation design because the developer does not have the freedom to place indexes on custom pages. Correspondingly, a possible implementation for the navigation structure depicted in Figure 3.5 must place the author index on the front page instead of on an extra `Authors` page. Second, as pages of an implementation always correspond to an entity, it is not possible to place a guided tour on an arbitrary page. Note that the Book Portal example has a guided tour of popular books on the front page which cannot be modeled with RMM. Third, RMM does not provide the notion of a menu, thus, it is not possible to define a menu that appears on multiple pages of the user interface.

3.4.1.4 Conclusions

As also recognized in [GHSL02], the expressive power of RMM for modeling Web sites is rather limited. It is suitable to model a simple Web site that is derived in a straight-forward manner from an Entity-Relationship diagram. However, it does not support custom pages or custom navigation structures. The RMM was conceived for the design of Web sites, thus of course, it does not support the design of Web applications with content management functionality. However, as one of the first approaches for the design of Web information systems, the RMM has inspired many subsequent methods for Web application development.

3.4.2 Object-Oriented Hypermedia Design Method (OOHDM)

The Object-Oriented Hypermedia Design Method (OOHDM) [SR95a][SRB96][SR98] [RPSO08] is an approach for modeling Web applications relying on object-oriented design principles. First resources describing the OOHDM were published in the mid 1990s. The original motivation for the approach was to support new requirements for Web applications emerging at that time, including the need to support services, new navigation structures, and sophisticated user interface elements. Since then this approach has been maintained constantly and it still appears as one of the major methods for Web application development in recent publications [RPSO08].

The OOHDM approach is built around a few cornerstone statements. First, it postulates that navigation classes are *views* of conceptual classes. This means that the navigation model is independent from the conceptual model and that the designer may decide which units of the conceptual model play a role in navigation design. Second, the OOHDM utilizes appropriate abstractions to organize the navigation space. Third, it separates the tasks of navigation design and user interface design. Finally, the OOHDM argues that there are some design decisions, which have to be made during the implementation of the Web application.

The OOHDM provides a set of graphical models and describes how these models may be used for certain development activities. The main activities of OOHDM are *Requirements Gathering*, *Conceptual Design*, *Navigational Design*, *Abstract Interface Design*, and *Implementation*.

3.4.2.1 Requirements Gathering

This activity is a recent addition to the OOHDM [RPSO08]. Early descriptions of the approach [SRB96][SR98] did not cover requirements analysis for Web application development. The *Requirements Gathering* activity aims at specifying the interaction of users with the Web application employing use cases. To this end, the OOHDM uses the so-called *User Interaction Diagram* [VSDS00] that provides a graphical notation for capturing user interaction more precisely than a UML *Use Case Diagram*. Figure 3.6 depicts a *User Interaction Diagram* for a use case in which the user of the Book Portal searches for a book and then navigates to one of the book's authors or to a book review. Note that a detailed description of the graphical notation of this diagram type may be found in [VSDS00].

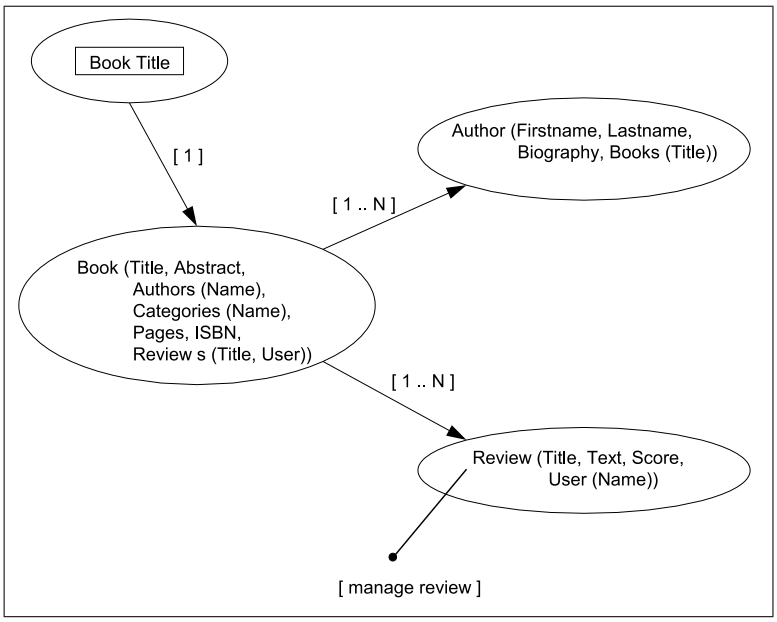

Figure 3.6: User Interaction Diagram Example for OOHDM

In this use case, a user of the Book Portal tries to locate a book in the portal by the book's title. Thus, the first state of the interaction process is the user providing the title of the book to the system. This user input is denoted by the rectangle in the first ellipse in the top left corner. If the user provided the title correctly, the interaction process reaches the second state and the system provides detailed book information. The ellipse in the middle specifies exactly, what information about the book is provided, e.g., a title, an abstract, a list of authors, or a list of reviews. Note that transitions between states are illustrated with directed edges between the source state and the target state. Transition edges may have labels that specify the number of options along a transition. In this use case, the user is required to specify the exact name of a book, thus the system may provide a single choice for the user, i.e., to select the book. After the system delivered information about the book, the user may proceed to look at the authors of the book or examine some of the book's reviews. These choices are indicated by the last two states of this interaction process. Both states may be reached through transitions that are labeled with [1..N] multiplicities, because in both cases there may be several options for the user to choose from. The top right ellipse denotes the interaction state that is reached if the user decides to view information about one of the book's authors. In this case, the system provides the author's name, biography, and a list of the author's books. The bottom right ellipse defines

a state in that the system shows information about a book review, e.g., the title, the review text, a score, the name of the user that issued the review, and the review date. Additionally, the last state defines the *manage review* operation that may be executed by the user.

Note that depending on the size of the Web application a considerable number of diagrams may be necessary to capture the entire user interaction process. Also the number of necessary diagrams depends on the number of interaction paths that are included in the diagrams. One the one hand, the presented diagram could include additional states, e.g., states in which information about categories or users are presented. On the other hand, the state in which information about a book author is presented could have been left out to concentrate on books and reviews. Furthermore, if the application is to be used by different user groups it may be necessary to specify a set of diagrams for each user group. Ultimately, this type of diagram is very useful to capture the requirements of a Web application because it may be used to derive information units that are to be included into other models. For example, it is apparent that a content model of the Web application implementing the depicted interaction process should at least support book, author, and review objects. Furthermore, it is to assume that the navigation model of the application will provide a book page from that the user may navigate to pages that present author and review information.

3.4.2.2 Conceptual Design

The original graphical notation of OOHDM for conceptual design [SR98] was a UML-near notation providing the usual modeling concepts of a class diagram, i.e., classes, attributes, relationships between classes, etc. However, at some point in time, the OOHDM switched to employ a UML class diagram for conceptual modeling. Figure 3.7 shows a UML class diagram of the Book Portal application.

The UML is a well known modeling language for application design and class diagrams are very well documented elsewhere [KHKR05], thus there is no need for a detailed discussion of the graphical notation. However, as a UML class diagram is more powerful compared to an Entity-Relationship diagram, it is worth to mention a few differences using the Book Portal example. First, the UML provides the concept of data types and allows the definition of multiplicities for attributes. Observe the *FirstNames* attribute of the `Person` class that has the *String* data type and the multiplicity of one-or-more. Second, the UML class diagram supports association classes that may contain attributes describing relationships between objects. An example is the *CommentDate* attribute of the `Comment` association class between the `Review` and `Publication` classes. Finally, the OOHDM employs the notion of *operations*, which are actions that may be executed by the Web application's user. The OOHDM specifies operations in UML-conform manner in the operation compartment of the class element. Observe, for example, the *manageReview()* operation in the `Review` class. Ultimately, a UML class diagram provides sufficient expressive power to define the content model of a Web application and there are also other methods for Web application development that employ UML for this task (see the UWE approach in Section 3.4.3).

3.4.2.3 Navigation Design

Navigation design is a critical step of the OOHDM for the Web application development process. The navigation model is considered a view of the conceptual model and defines which information units (conceptual classes) are relevant for presentation to the Web application's

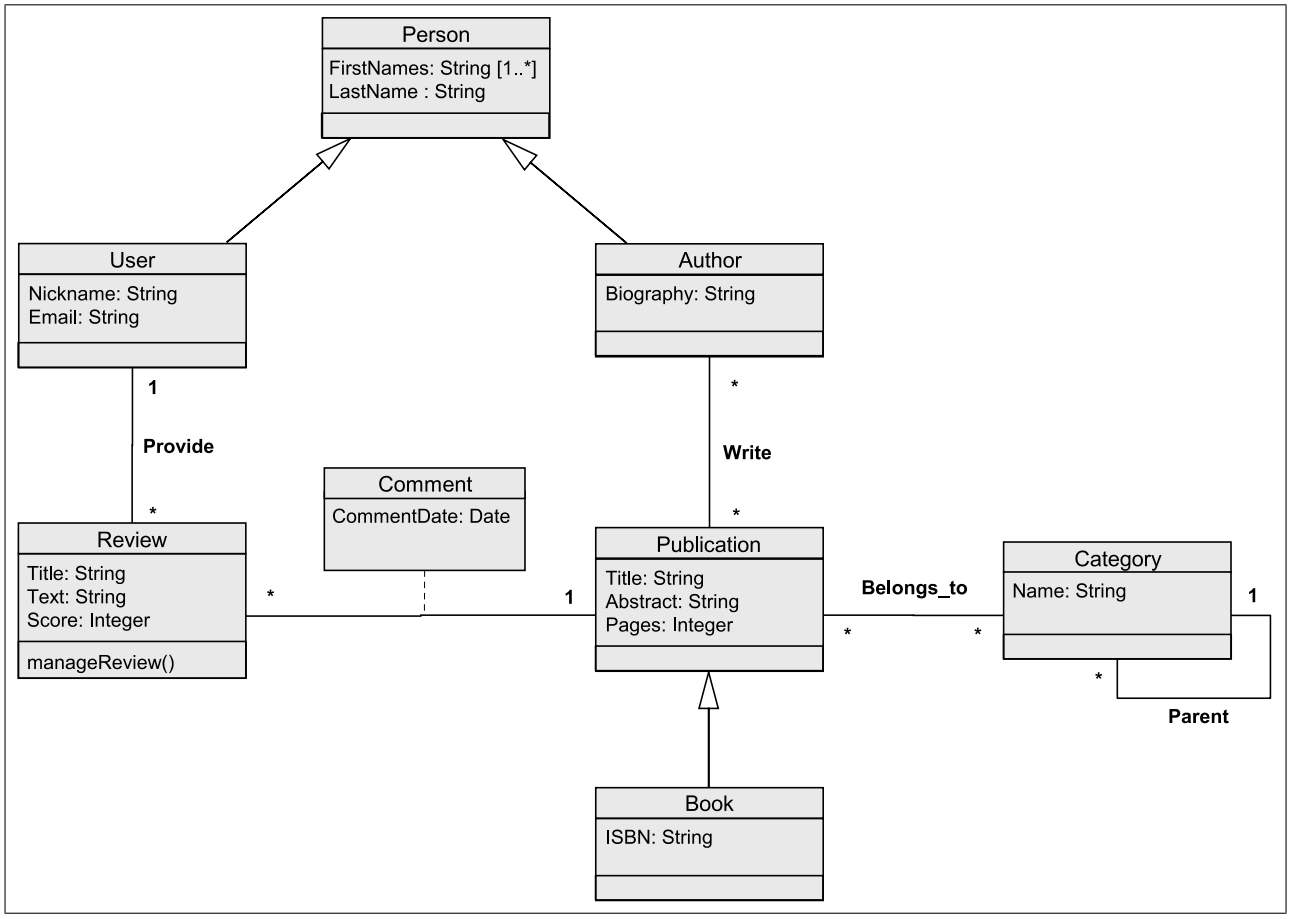

Figure 3.7: Conceptual Model of the Book Portal for OOHDM

user and which navigation structures are appropriate to interconnect these information units. To this end, the OOHDM employs two different diagram types for navigation design and a SQL-like query language that may be used in conjunction with the diagrams for the exact specification of views and access structures.

Navigation Class Diagram The first diagram that the OOHDM employs for navigation modeling is the *Navigation Class Diagram*. It defines navigation nodes that are views over conceptual classes and access structures that interconnect navigation nodes. To this end, this diagram identifies which classes of the conceptual model are relevant for navigation, which attributes of a conceptual class are to be included for presentation and which relationships between classes are to be converted into access structures between navigational nodes. Figure 3.8 depicts a *Navigational Class Diagram* of the Book Portal application.

The graphical notation of this diagram is similar to the notation of the *Conceptual Model*. However, the specified navigation edges and the index definitions prohibit the notation from being UML-compliant. The diagram includes all conceptual classes of the Book Portal that are relevant for navigation. For example, it contains the `Book` navigation class that specifies what information is to be shown on a navigation node that presents a book object. To this end, the class contains the *Title*, *Abstract*, *Pages*, and *ISBN* attributes. Additionally, the `Book` navigation class specifies the *Categories* and *Authors* indexes as well as a *Review* listing. Besides indexes and listings, the navigational model may also define simple links between objects. Observe the

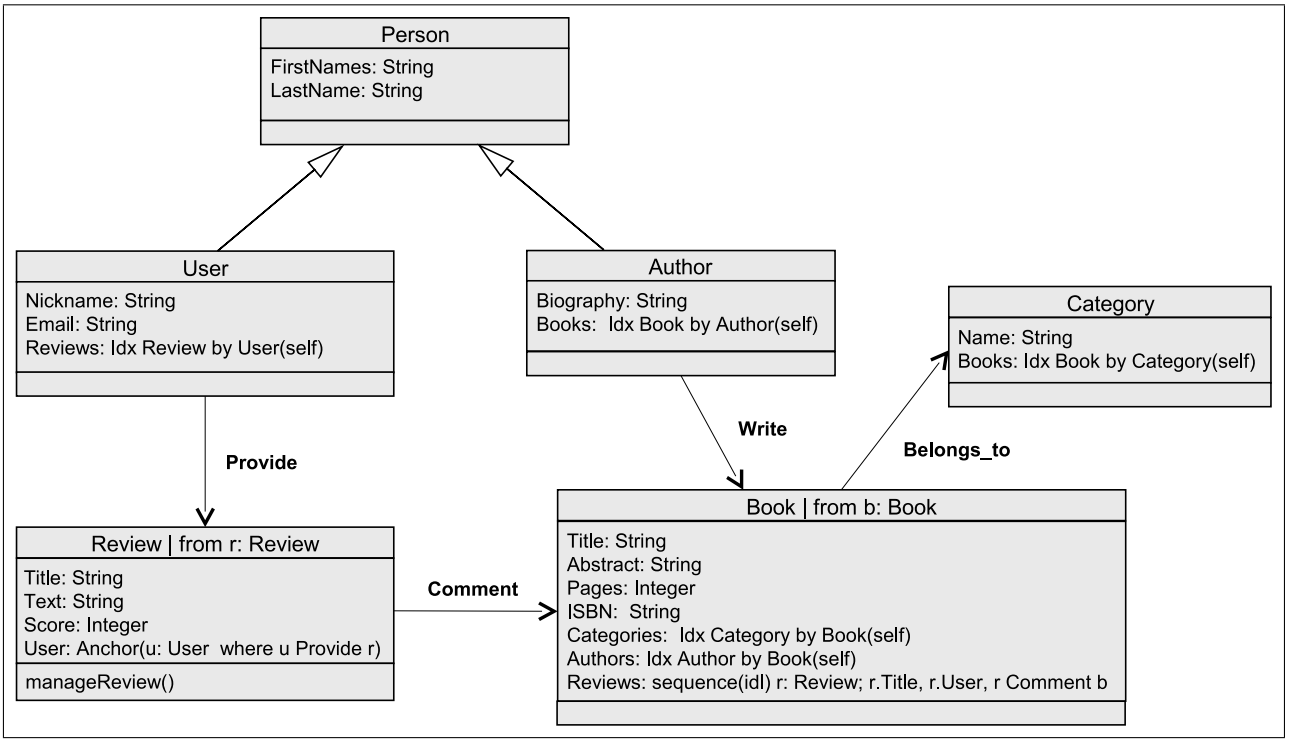

Figure 3.8: Navigational Class Diagram of the Book Portal for OOHDM

`Review` navigation class that provides a link to the `User` class. To this end, the *User* attribute defines an anchor utilizing the SQL-like query language of the OOHDM. The expression "u: User where u Provide r" identifies a `User` object that is related to the `Review` object by the *Provide* relationship. Note that navigation steps are also illustrated with edges between navigation classes. For example, the edge between `User` and `Review` classes indicate that there is a navigation step between these two nodes and that it is based on the *Provide* relationship. However, this information is redundant as the `User` class already defines a review index. Also note that the *Navigational Class Diagram* supports specialization. Analog to the *Conceptual Model*, the `User` and `Author` classes are specializations of the `Person` class. Also analogous is the specification of the *manageReview()* operation in the `Review` class. The operation is included into the navigation model because it is relevant for user interaction.

Navigation Context Diagram The second diagram of the OOHDM for navigation modeling is the *Navigation Context Diagram*. This diagram relies on the notion of a *Navigation Context*, which is the context in that a set of objects are presented to the Web application's user. The notion of navigation contexts are best explained by example. A simple navigation context of the Book Portal application may be "Authors of a book". In this context, the user is usually presented a view of an `Author` object that additionally to author information also contains user interface elements that allow navigation to the next and previous authors. Another context that may be employed by the example application is "All Authors" which, as the name suggests, includes the complete set of authors. In this context, it may be desirable to provide an alternate presentation of an `Author` object. For example, if an author is visited in the context "Authors of a book", it is reasonable to include a link "Back to the Book" into the author view. In contrast to that if an author is visited in the context "All Authors", an appropriate link may be "Back to

the Author Index". Also navigating to the next or to the previous author in different navigation contexts may generate different results. This is obvious as the number and ordering of objects in different contexts may differ greatly.

Ultimately, a *Navigation Context* defines a context-sensitive presentation of objects and context-sensitive navigational characteristics for object sets. To provide a custom object presentation, each navigation context defines a view for the corresponding object in form of a so-called *In-Context* navigation class. The notation of such a navigation class is identical to those of the *Navigation Class Diagram* and may define attributes, access structures, and operations for an object. The navigational characteristic of an object set is determined by a navigation context in two ways. First, the context identifies those objects that may be visited in the context. Second, the context defines the order in that objects may be visited. A *Navigation Context Diagram* includes all navigation contexts of an application and defines additional access structures (menus, indexes, etc.) that provide access to the contexts. Figure 3.9 depicts all navigation contexts of the Book Portal example application.

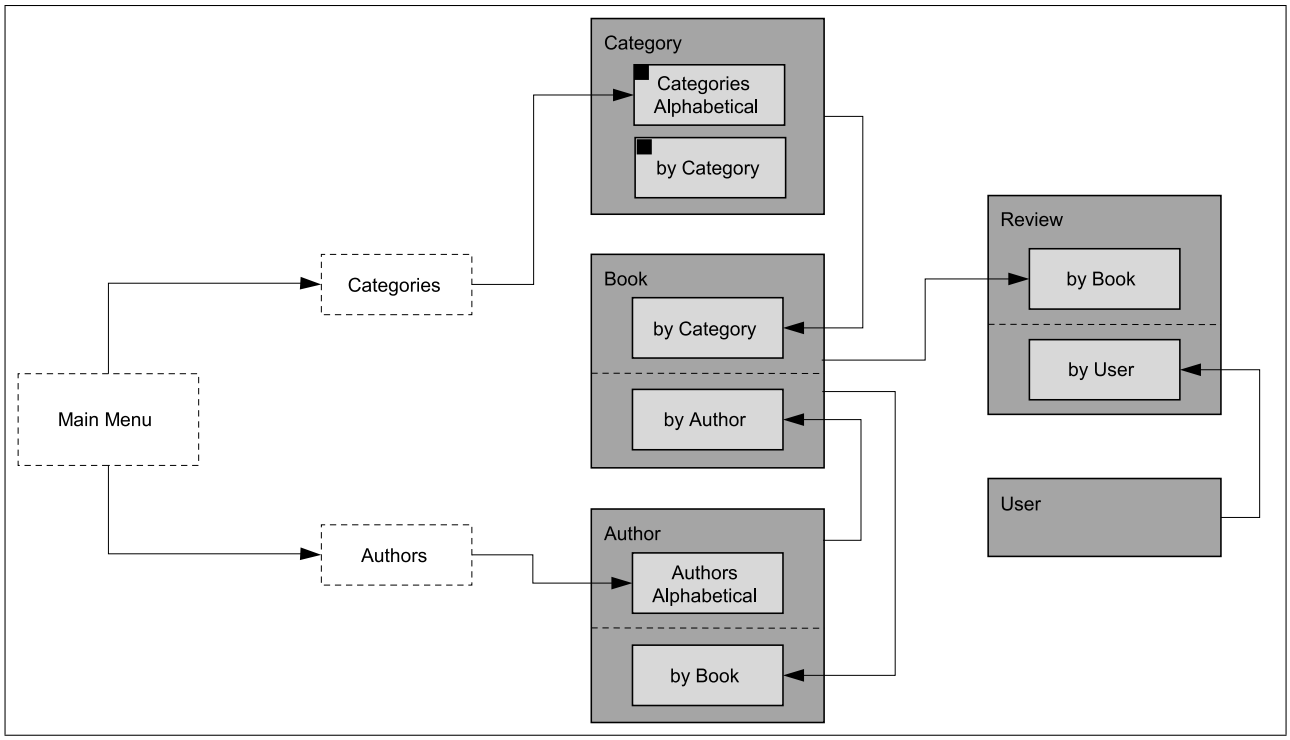

Figure 3.9: Navigation Context Diagram of the Book Portal for OOHDM

This diagram employs a proprietary graphical notation for capturing navigation contexts and access structures. Menus and indexes are denoted with simple rectangles having dashed edges. Navigation contexts (light grey background) are grouped by type (dark grey background). A small black box in the top left corner of a rectangle denoting a navigation context indicates that the corresponding navigation context includes an index. Navigation contexts are separated by a dashed line if it is not allowed to switch from one context to another without leaving the current context.

The *Navigation Context Diagram* of the Book Portal defines a main menu with two entries that lead to the *Categories* and *Authors* indexes, respectively. From the *Categories* index the user may navigate to the *Categories Alphabetical* navigation context, which belongs to the *Category* group.

This group also contains a *by Category* context because categories are nested and categories may be browsed by categories. Note that there is no dashed line between the two navigation contexts of the *Category* group. This indicates that it is possible to switch between these two contexts at any time. The small black boxes indicate that these contexts employ indexes. A possible implementation for this navigation construct is a tree-like navigation structure that shows categories in alphabetical order and for each category a sub-tree provides sub-categories.

Navigation edges between groups and navigation contexts in Figure 3.9 indicate possible navigation steps for the user of the Book Portal application. For example, the navigation edge pointing from the *Book* group to the *by Book* context of the *Author* group indicates that the user may navigate from all contexts of the *Book* group to this context. Besides the *by Book* context the *Author* group also contains the *Authors Alphabetical* context. However, these contexts are separated with a dashed line, which indicates that the user cannot switch from one of these contexts to the other. Of course, if the user of the Book Portal is in the *by Book* context, then he looks at information about an author in context of a book. The context contains all authors of the corresponding book and the user may navigate to the previous or next author. In this context, it makes no sense to switch to the *Authors Alphabetical* context that includes the complete list of authors.

3.4.2.4 Abstract Interface Design

The final design activity of the OOHDM for Web application development is the definition of the Web application's user interface. According to the OOHDM, the aim of this activity is to cover three major tasks. First, it should define the appearance of user interface elements. Second, it should specify which elements activate navigation. Finally, it should define which elements are responsible for providing further functionality of the Web application, e.g., data input by the user or the activation of operations. The OOHDM postulates that these aspects must be captured with a model and must be considered in the design phase and not merely during implementation. Furthermore, the approach requires that the user interface is defined at an abstract level making the model independent from the actual implementation. To this end, early publications on the OOHDM [SRB96][SR98] employed the notion of *Abstract Data Objects* and *Abstract Data Views*. Abstract data objects are elements of the navigation model, e.g., navigation classes, menus, or indexes. Abstract data views are actually abstract user interface elements, e.g., buttons, text fields, etc. These abstract elements may be combined to create more complex components of the user interface. For example, several text fields and a button may be combined to create a Web form. However, the OOHDM does not define a classification or a concrete graphical notation for abstract data views. It merely employs simple sketches to indicate a possible layout of user interface elements. Figure 3.10 depicts the layout of the `Book` page of the Book Portal application.

The figure provides a rather high-level definition of the *Book* interface page. It indicates that a corresponding page of the implementation includes some of a book's attributes, e.g., the title or the abstract. Additionally, it defines some areas that may display information related to the book, e.g., reviews or information about authors. Finally, it suggests some interface elements that allow navigation to other pages of the user interface, e.g., to a page about authors or to a page that allows to submit a review.

A current publication on OOHDM [RPSO08] introduces a taxonomy of abstract interface widgets that may be used to create a more detailed design of the user interface. The taxonomy

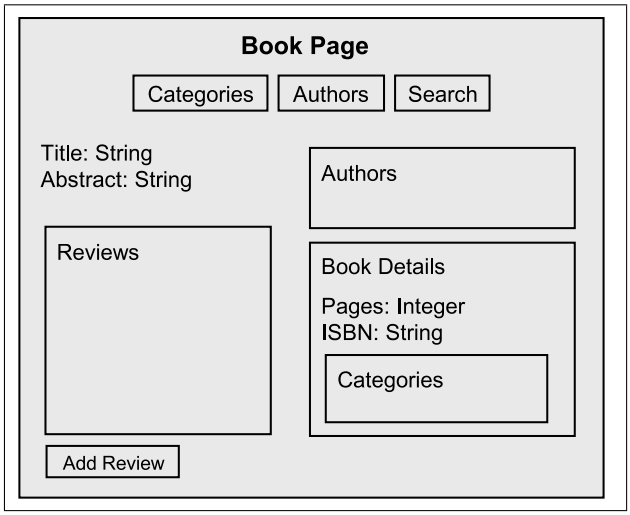

Figure 3.10: Abstract Interface Design of the `Book` Page for OOHDM

defines three types of widgets. First, it includes the *ElementExhibitor* widget, which is used to denote an interface component that presents content to the Web application's user. Second, it includes the *SimpleActivator* widget, which specifies an interface component that reacts to user interaction. Finally, it defines the *VariableCapturer* widget and a set of its specializations, which denote interface components that capture user input. Additionaly, to these widget types the taxonomy includes the *CompositeInterfaceElement* widget that allows to arbitrarily combine the previously introduced widgets. The OOHDM argues that these abstract interface widgets must be mapped to elements of the navigation models and to concrete components of a chosen implementation framework. The approach uses the OWL [URI04b] to achieve the latter, however, fails to provide a solution for the former.

3.4.2.5 Implementation

The last activity of the OOHDM is the implementation of the Web application. Unfortunately, the OOHDM is not supported by a CASE tool that is able to generate an implementation from its graphical models. During the development of the OOHDM, several approaches have been proposed to *support* the implementation phase. Early publications [SRB96][SR98] suggested to implement the Web application using a relational database management system (RDBMS) for data storage, and HTML and CGI scripting for constructing the user interface without providing any detailed guidelines. This was of course a great disadvantage of the method at that time, because the implementation of a Web application utilizing the mentioned technologies based on OOHDM models was very time consuming for several reasons. First, the OOHDM employs an object-oriented data model which has to be mapped to the relational data model of an RDBMS. Object-relational mapping is a time-consuming and error-prone development activity. Second, the OOHDM employs the concept of navigation contexts for defining the navigation structure of the Web application. Although the navigation context is an interesting design concept, it adds another level of complexity to the implementation.

In 1999, the OOHDM-Web implementation environment [SPM99] was introduced to support the implementation phase of the OOHDM. This environment employed the Lua scripting language and a set of templates to construct the Web application's user interface. The sys-

tem provided three Web interfaces. The first one allowed the developer to create the navigation schema and the interface templates according to the existing conceptual schema of the underlying RDBMS. The developer could use the second interface to create instance data corresponding to the previously defined schemata. The third interface presented the actual Web application to the end user. Unfortunately, the OOHDM-Web environment did not support the OOHDM's graphical models but provided form-based Web interfaces for all modeling tasks.

In [LS03][SSML04][NS06], the OOHDM was continuously extended to produce the Semantic Hypermedia Design Method (SHDM). During its evolution, the SHDM has employed a range of different languages and frameworks. First, the models known from the OOHDM were expressed with the Resource Description Framework (RDF) [URI04a] and DAML+OIL [URI01] and the Java programming language was utilized for implementation [LS03]. Afterwards, the approach switched to the Web Ontology Language (OWL) [URI04b] and the Sesame RDF Framework [URI08k] for storing models and data [SSML04]. Finally, in [NS06] the HyperDe modeling and implementation environment was introduced. This environment is based on the Ruby on Rails Framework [TH07][URI08j] and still employs the Sesame framework for data storage. HyperDe also relies on a proprietary Domain Specific Language (DSL) based on the Ruby programming language. HyperDe allows to create and modify the design models and application data using the DSL. Although the SHDM uses a set of new technologies the focus of the approach is still the same as that of the OOHDM. The advantage of SHDM is that it is capable of producing an implementation, however, to this end all graphical models have to be manually recreated in a notation (e.g. the HyperDe DSL) that is appropriate for the implementation environment.

3.4.2.6 Conclusions

The OOHDM is well suitable for the design of Web information systems as its models for requirements analysis, conceptual modeling, and navigation design allow to capture most of their relevant aspects. However, the approach has two major deficits. First, it does not support the development of Web applications that provide content management functionality. Second, the OOHDM does not provide a CAWE tool that supports the creation of graphical models and is able to generate a functional implementation. The advantages and disadvantages of the approach are summarized subsequently comparing it to the RMM.

The OOHDM employs a UML class diagram to create the *Conceptual Design* of the Web application. Compared to the RMM, which employs an Entity-Relationship diagram for content modeling, the OODHM's approach is superior. However, it makes little use of the greatest advantage of class diagrams, which is modeling behavior with operations. As a matter of fact, the OOHDM does not deal with the detailed design of application behavior. It is typical for an OODHM conceptual design to specify behavior on a rather abstract level. Take for example the *manageReview()* operation of the `Review` class in Figure 3.7. Usually, the management of a content object is more complicated than just defining a corresponding operation. It includes control over object attributes, object relationships, and possibly over attributes of relationships.

The OOHDM provides two diagrams for modeling navigation concerns of Web applications. The *Navigation Class Diagram* has roughly the same expressive power as *m-slices* of the RMM. However, the OOHDM's notation is more compact, consistent with the content model, and more intuitive for designers. Unfortunately, both methods rely strongly on their content models for deriving nodes of the application's navigation structure. They do not allow to define

custom nodes that do not directly correspond to a content model class. The *Navigation Context Diagram* in OOHDM introduces the concept of a *Navigation Context*, which defines how the user may iterate over different objects sets and how each object is presented in a certain context. This concept strongly relies on classes and relationships between classes that are defined in the OOHDM's conceptual schema as these are used to derive the definition of element sets and different access structures, e.g., menus and indexes. Unfortunately, the *Navigation Context Diagram* does not allow to define custom navigation nodes or navigation structures either. Thus, the OOHDM is not capable of modeling the complete navigation structure of the Book Portal application. Note that the `Search` page or pages supporting the management of reviews, e.g., the `Add Review` and `Manage Review` pages cannot be modeled with OOHDM.

The *Abstract Interface Design* activity of the OOHDM defines the user interface of the Web application at an abstract level. To this end, the method uses abstract interface elements that may be used to compose each page of the user interface. Abstract interface elements are mapped to navigation nodes and to concrete user interface widgets of an implementation framework. This activity is named *User-Interface Screen Design* in the RMM. Although the RMM identifies this activity as a unique step of the development process, in contrast to the OOHDM, it does not provide any models or guidelines for executing it. Unfortunately, the OOHDM gets imprecise when it comes to user interface design. Early publications on the OODHM introduce a simplistic graphical notation that sketches the layouting of each user interface page. However, this graphical notation does not provide any mappings to navigation nodes. The SHDM, which is an extension of the OOHDM, employs the RDF and the OWL to specify textual mappings instead of a graphical notation. On the one hand, the OODHM way to model the user interface is too imprecise to be useful for a direct implementation. On the other hand, the SHDM abandons graphical modeling which is a great disadvantage of the method.

Finally, the OOHDM's and the SHDM's support for automatically creating the implementation is insufficient. Neither the OOHDM nor the SHDM provides a CAWE tool that is capable of creating graphical models and generating a fully functional implementation. Both implementation environments that support these methods dispense with graphical modeling. The OOHDM-Web environment provides different Web interfaces, whereas the HyperDe environment employs a domain-specific language to programmatically create models and instance data.

3.4.3 UWE

The UML-based Web Engineering (UWE) method for Web application development was introduced towards the end of the 1990s [BKM99][HK00]. It is based on previous Web engineering methods, e.g., the RMM and the OOHDM, but in contrast to them, it employs the UML for all design activities of the Web application development process. Besides standard UML diagrams, e.g., use case diagram or class diagram, the UWE method extends the graphical notation of the UML with a UML profile. Originally, the method supported three design activities [BKM99]. First, modeling the application domain with a *Content Model*. Second, modeling navigational characteristics of the Web application with a *Navigation Space Model* and a *Navigation Structure Model*. Finally, the modeling of the user interface with a *Presentation Model*. Over time, the UWE process has been extended to support requirements analysis with use case diagrams and activity diagrams [HK00][KK02a] with additional focus on business processes [KKCM03][KKZH04]. Subsequent sections describe each design activity of the UWE.

3.4.3.1 Requrirement Analysis

The UWE method started to support the *Requirement Analysis* phase of the Web application development process shortly after the method had appeared [HK00]. The method suggests to specify requirements for the Web application with a standard UML *Use Case Diagram* [HK00] [KK02a]. This diagram captures a piece of the Web application's behavior by connecting users to certain functionalities of the application. To this end, a use case diagram employs two core concepts. First, an *actor* is a user, who interacts with the system in some way, e.g., a customer or an administrator. Of course, the notion of an *actor* does not identify a single user instance but rather a group of users. Second, a *use case* specifies certain functionality of the system that is linked to one or more actors. Use cases are connected to actors with simple *associations*, which denote that the functionality captured by the use case is to be provided for the corresponding actor. It is also possible to define relationships between use cases. The *«include»* relationship between two use cases denotes that one use case *includes* the other, i.e., the included use case is always executed along with the including use case. Additionally, the *«extend»* relationship between use cases denotes that a use case *extends* another one, i.e, the extending use case may be executed with the extended use case. Finally, it is allowed to define specialization/generalization relationships between use cases or actors, respectively. A specialized actor inherits all use cases that are associated to its super-actor.

In [KKZH04], a custom *«navigation»* keyword is introduced to mark use cases that include navigation activity by an actor. As navigation design is a key aspect in Web application development, this extension to the use case diagram is advantageous. It allows to denote system functionality with navigational characteristics early in the development process. Use cases that are not marked with the *«navigation»* keyword define business logic that is not related to navigation.

The UML defines several possible graphical notations for a use case diagram. A commonly used notation denotes actors with a stick figure icon and use cases with an ellipse. Associations between actors and use cases are denoted with a solid line and relationships between use cases with dashed arrows. Names of use cases are written inside the ellipsis and all other names of keywords are positioned above or below the corresponding graphical element. Figure 3.11 depicts a partial use case diagram of the Book Portal application.

Figure 3.11 shows the `User` actor that is associated with the two main use cases *Find Book* and *View Book*. Both use cases are navigation relevant, because the user of the Web application is supposed to carry out one or more navigation steps while using the corresponding functionality. The Book Portal allows the user to find books through different channels. He may use keyword search, browse through categories or through the list of authors. Corresponding use cases are defined as extensions of the `Find Book` use case. The `View Book` use case includes the `View Reviews` use case because reviews of a book are to be presented together with book information. Both use cases are navigation relevant, which is indicated with the *«navigation»* keyword. Finally, the `Logged In User` is a specialization of the `User` actor and is associated with two additional use cases. The `Add Review` and `Manage Review` use cases deal with the management of reviews. However, this functionality is restricted to logged-in users only, thus the corresponding use cases are not associated with the `User` actor.

Of course, use case diagrams are not appropriate to create an in-depth requirements specification of a Web application. Although the designer is free to define use cases at different abstraction levels, the expressive power of a use case diagram is weak, if it comes to specify-

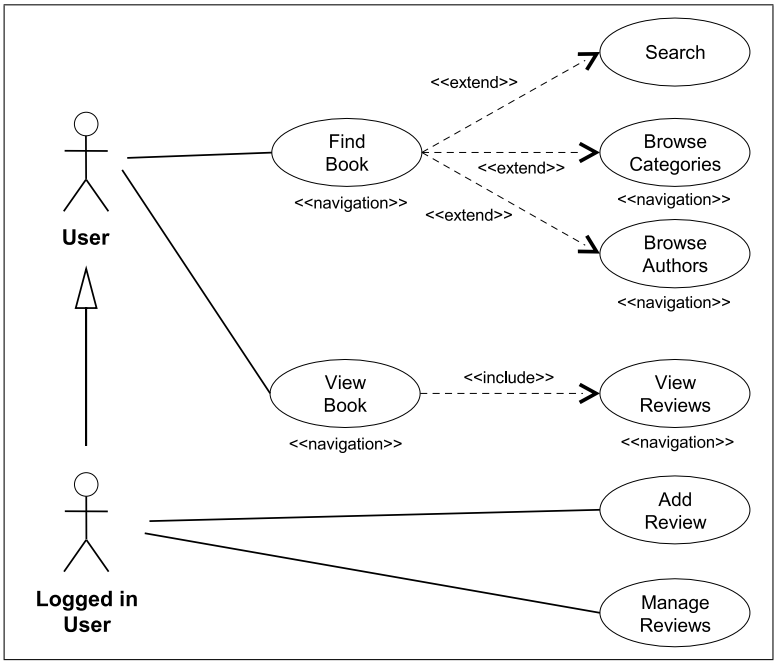

Figure 3.11: Partial Use Case Diagram of the Book Portal for UWE

ing relationships between use cases and in general between user actions. For example, certain activities are to be executed in a specific order or they may depend on various conditions. For example, it is obvious that a customer of an online store may only proceed to checkout if he has already placed some products into the shopping cart. The necessity to further refine the requirements specification was also recognized by the creators of UWE. In [KZE06], the UWE method is extended to capture requirements in a more detailed manner utilizing UML *Activity Diagrams*. Furthermore, the UWE UML profile defines three *action* stereotypes that may be used in activity diagrams to denote Web-application-specific actions. First, the *«browse»* action indicates that the Web application's user is required to *browse*, i.e., to navigate between different pages of the user interface. This action is marked with an arrow icon. Second, the *«search»* action denotes that the user is provided a search form, with which he may execute a search. This action is illustrated with a question mark icon. Finally, the *«user transaction»* action specifies a transaction that changes the Web application's content and is indicated with a bidirectional arrow icon. The UWE method proposes to create one or more activity diagrams for each use case of a requirements specification. In Figure 3.11, the partial use case diagram of the Book Portal application included the `Add Review` use case. Figure 3.12 depicts an activity diagram for this use case.

The Book Portal application allows registered users to create book reviews that may be read by other users of the portal. To this end, a logged-in user may navigate from a book page to a Web form that allows him to create a review. This process is depicted in Figure 3.12 with an activity diagram. The process allows the user to create zero or more reviews for a book and includes four activities. First, the user may view information about a book, for example, on a book page. Second, he may activate a button *Add Review* on the book page to be forwarded to an appropriate Web form. Third, the user may provide the review data. Finally, if the input data was correct, the review is created and the user may look at the book page containing the new review or abandon the process. Note that the first two actions include browsing activity

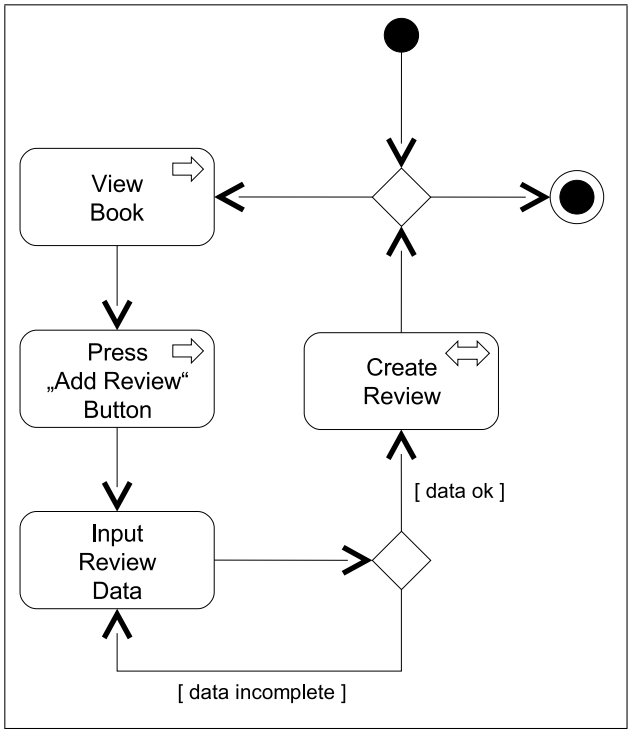

Figure 3.12: Activity Diagram of the `Add Review` Activity for UWE

by the user, thus they are marked with an arrow icon. The last is a transactional action that creates a new data object, thus this action is marked with a bidirectional arrow.

3.4.3.2 Content Modeling

The UWE method employs the UML for all design activities. This is also the case for modeling the Web application's content. For this task, the UWE method utilizes a standard UML class diagram. In contrast to the OOHDM, which adopted full UML compliance at some point in the method's maturing process, the UWE method has applied a UML class diagram for content modeling from the beginning. It utilizes the options of a UML class diagram fully. Figure 3.13 depicts the *Content Model* of the Book Portal application for the UWE method.

The syntax of a UML class diagram is well known and is described in detail elsewhere [KHKR05]. Also, Figure 3.13 showing the content model for the UWE approach is very similar to the corresponding model of the OOHDM, which is shown in Figure 3.7. Accordingly, the class diagram is described in Section 3.4.2.2. However, there are two subtle differences between the two methods. First, the UWE method utilizes the expressive power of the class diagram to a somewhat larger extent. For example, it defines role names for associations. Correspondingly, the *Provide* association between the `User` and `Review` classes is also characterized with the role names *ReviewProvider* at the user end and *ReviewItem* at the review end of the association. Second, the UWE method defines operations at a lower level of abstraction than the OOHDM. For example, it is characteristic for the UWE approach to define operations like *addReview()* or *deleteReview()* for the `Review` class rather than a more general operation *manageReview()* (see Figure 3.7), which is to cover all aspects that are related to the management of a content object. Ultimately, the UML class diagram is well suited to capture the content model of a Web application.

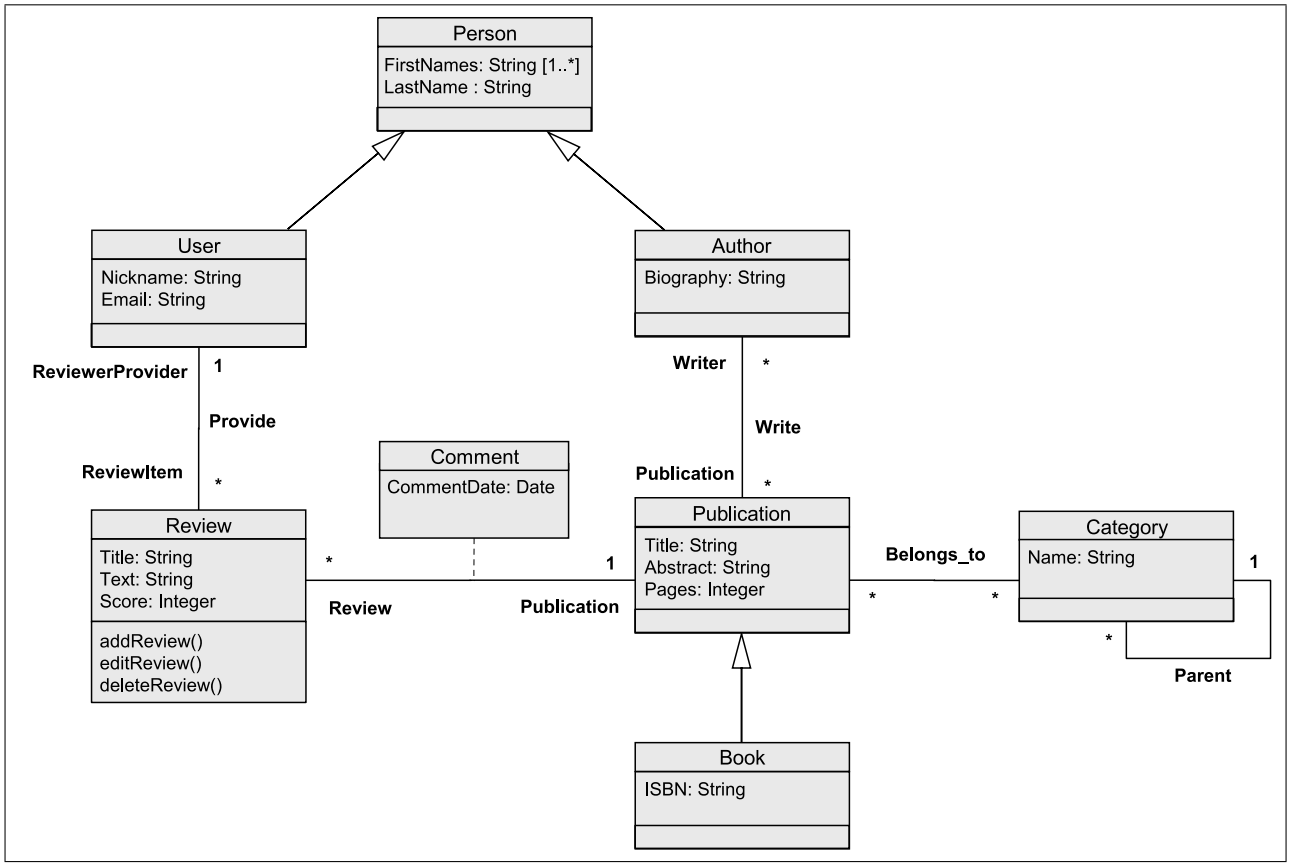

Figure 3.13: Content Model of the Book Portal for UWE

3.4.3.3 Navigation Modeling

The UWE approach provides two models that consider the navigational characteristics of a Web application [HK00][KK02a]. The *Navigation Space Model* defines which objects are relevant for navigation and may be visited by the Web application's user. The *Navigation Structure Model* specifies how navigation objects may be reached by the user and connects them with different access structures, e.g., menus and indexes. The first model is directly derived from the *Content Model* of the UWE method, whereas the second model, which defines the navigation structure of the Web application, utilizes components of the first model, i.e., depends on objects of the navigation space. Both models employ a set of stereotypes that extend the standard UML notation.

Navigation Space Model The *Navigation Space Model* defines objects that are relevant for navigation and connects them directly with navigation links. To this end, the model offers two concepts, which are modeled with UML stereotypes. First, the «*navigation class*» stereotype defines an object that is a node in the navigation space of the Web application. Navigation classes correspond directly to classes of the *Content Model*. Each content model class that is relevant for navigation is included in the navigation space and becomes a navigation class that contains all attributes of the corresponding content model class. Additionally, content model classes that are left out of the navigation space may be included as *derived attributes* of a navigation class. In such a case, an OCL [War03] constraint defines which content model class is to be included as a derived attribute. Second, the «*direct navigability*» stereotype defines associations between

navigation classes and indicates that there is a direct navigation link between corresponding navigation objects. Note that these associations are derived directly from associations between content model classes but they define navigation paths, thus they possess different semantics. According to the navigational requirements of the Web application, these associations may be unidirectional or bidirectional. Figure 3.14 depicts the *Navigation Space Model* of the Book Portal application.

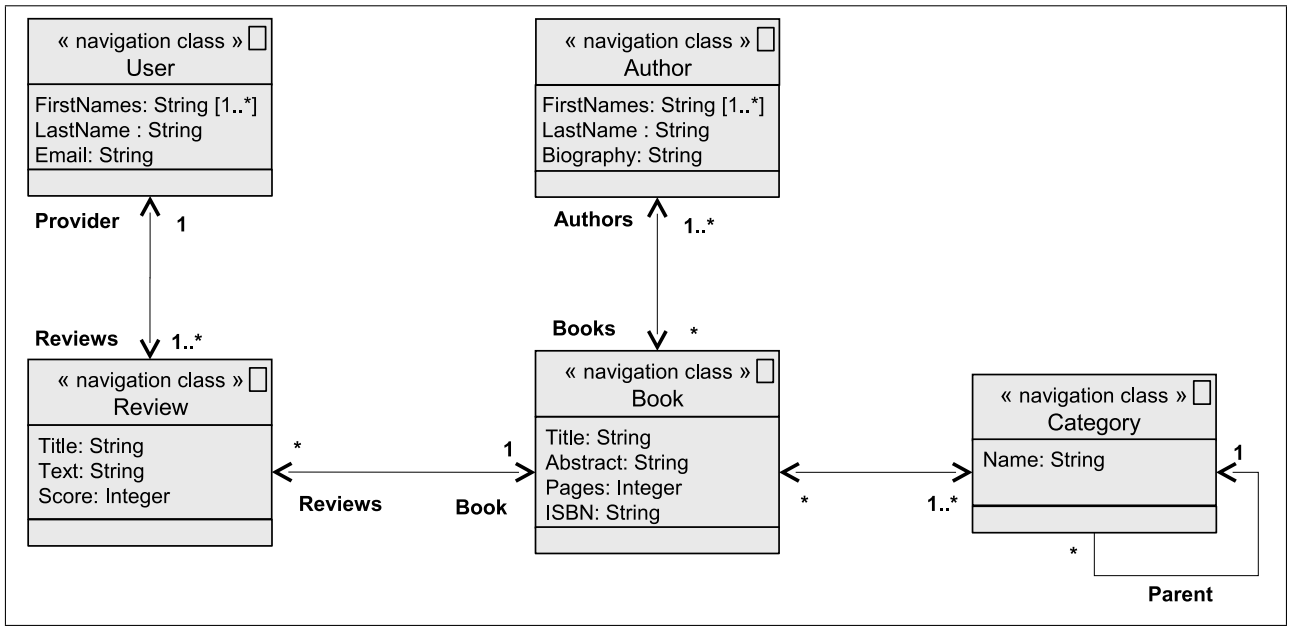

Figure 3.14: Navigation Space Model of the Book Portal for UWE

The syntax of this model is very similar to a standard UML class diagram. However, there are two minor differences, which correspond to the two basic concepts of the UWE approach for defining the navigation space of the Web application. The *«navigation class»* stereotype is depicted as a stereotyped class and is additionally marked with a small box at the right-hand side of the class header. This can be observed at all five navigation classes in Figure 3.14. Note that this navigation space diagram example does not employ specialization/generalization, thus all classes additionally contain attributes of their super-classes. Also note that this navigation space model contains all classes of the corresponding content model, thus there are no classes to be represented as derived attributes. However, the designer could decide not to include the `Authors` class into the navigation space and to represent this object as a derived attribute *Authors* of the `Book` navigation class.

The *«direct navigability»* stereotype is a stereotyped association and is shown as a directed arrow between classes. An example is the bidirectional association between the `Author` and `Book` classes. The role name at the end of an association indicates which role a specific navigation class plays in regard to the other class. The multiplicities indicate how many objects of a certain type are related to another object at the opposite side of the association. For example, the *Authors* label and the 1..* multiplicity of the association between the `Book` and `Author` classes specify that there may be one or more navigation links from a book to authors in the navigation space and that the role of the linked objects is to be the *Authors*, which is of course trivial in this case.

Navigation Structure Model The *Navigation Structure Model* of the UWE method captures the navigation structure of the Web application. It extends navigation nodes and direct navigation associations between nodes that have been defined in the *Navigation Space Model*. The aim of the model is to specify how navigation objects may be visited by the Web application's user. To this end, the model enhances the navigation space with common access structures, e.g., indexes, menus, and guided tours.

The initial version of this model introduced in [BKM99] was strongly influenced by the OOHDM's *Navigation Context Diagram*. It is based on the notion of navigation contexts, which were introduced by the OOHDM (see Section 3.4.2.3). This version of the model utilizes three stereotypes to define different navigation contexts. First, a simple *«navigation context»* stereotype defines the complete set of navigation objects of a certain type. Second, the *«grouped context»* stereotype specifies a partitioning of navigation objects by a certain grouping condition. An example for a grouped context is the partitioning of books by authors. The books of an author build a single partition and all partitions (each belonging to a single author) compose the grouped context. Finally, the *«filtered context»* stereotype defines a set of navigation objects that have been selected from all objects of a certain type depending on a particular condition. An example of a filtered context is a list of books having more than 1000 pages. All contexts belonging to a certain class are gathered into a *package*. Additionally, access structures like menus and indexes are used to connect navigation contexts. However, this approach being a UML copy of the OOHDM's *Navigation Context Diagram* brings along all disadvantages of the model. The most evident problem is that the concentration on navigation contexts and their grouping into packages produces a diagram that does not depict well the navigation structure of a Web application. Perhaps, this was the reason for the UWE method to abandon this approach.

In [HK00], a new *Navigation Structure Model* was introduced that has been used since then for modeling the navigation structure of Web applications with the UWE method. This model is similar to the one that was proposed by the RMM for navigation design (see Section 3.4.1.3), however, it employs a UML notation and a set of stereotypes for defining common access structures. The *«menu»* stereotype specifies a menu which corresponds to the commonly used *menu* access structure. A menu item may be connected to all other elements of the model. The *«index»* stereotype specifies an index to a set of navigation objects of a certain type that have been selected by some criteria. An index is always connected to a navigation class. The *«guided tour»* stereotype corresponds to the commonly used concept of guiding the user through a set of navigation objects that similarly to an index have been selected by certain criteria. Finally, the *«query»* stereotype specifies a query by the Web application's user, i.e., the user may specify some input data that is used to produce a certain result, which may be, for example, an index of navigation objects. Figure 3.15 depicts the *Navigation Structure Model* of the Book Portal application.

The entry point to the Book Portal application is modeled with the *Main Menu*. The menu entries lead to the main areas of the application. The *Categories* and *Authors* entries point to indexes of categories and authors, respectively. An *«index»* stereotype is depicted with a rectangle with three horizontal lines inside. Note that for each index an ordering is specified. For example, authors are listed by their names and categories by their names and by their parent categories. The multiplicity at the end of the navigation association pointing to an index is always 1 because there is always a single index. In contrast to that, the multiplicity at the end of the association between an index and a navigation class is usually 1..*, because the index has

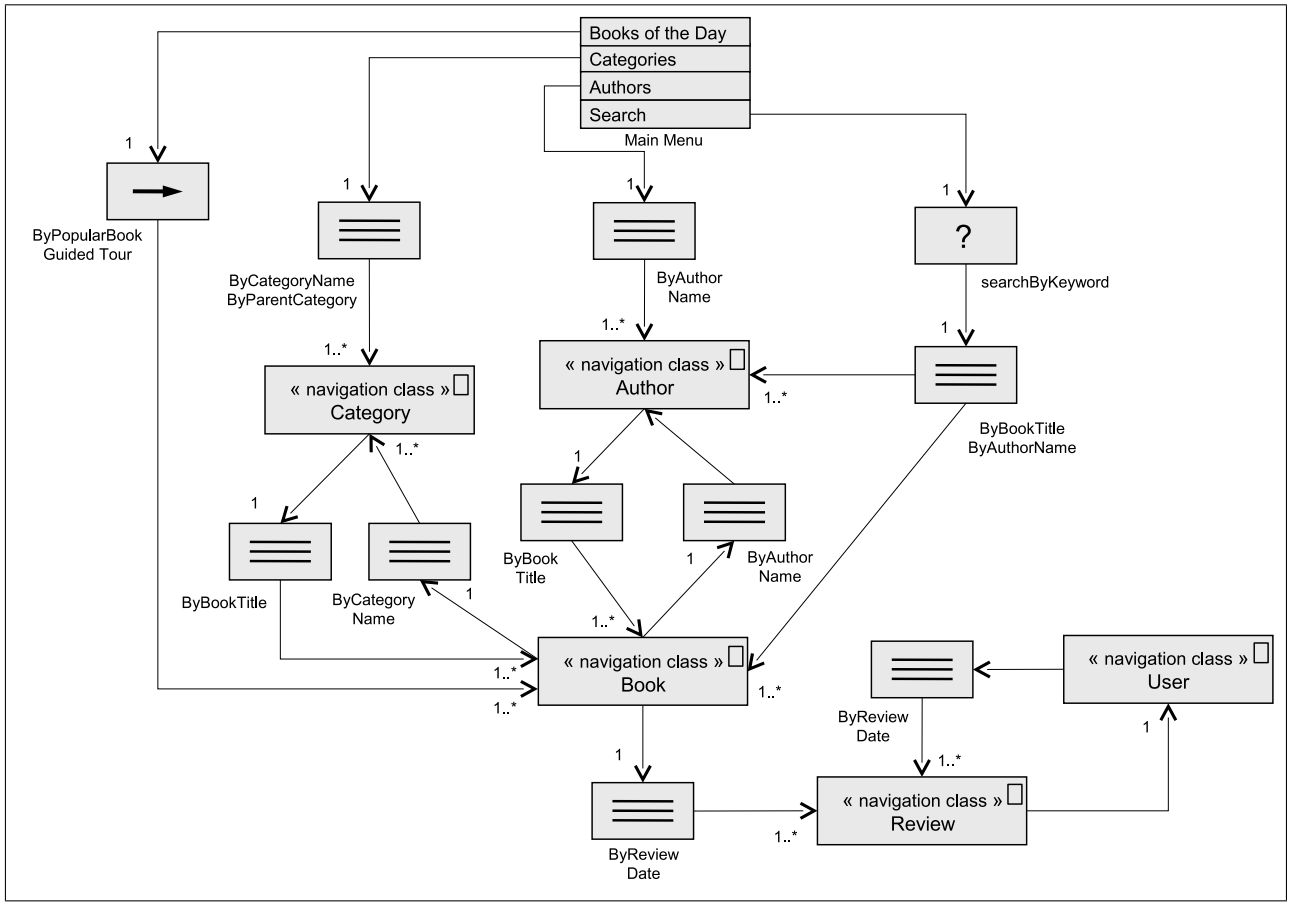

Figure 3.15: Navigation Structure Model of the Book Portal for UWE

usually one or more entries, thus it points to one or more navigation objects.

As already mentioned, the *Navigation Structure Model* extends the *Navigation Space Model*. Navigation classes that are defined in the first model are also in the second one. However, for the sake of simplicity a simplified notation is employed. A *«navigation class»* stereotype is depicted without the list of its attributes. Note that all navigation paths defined in the navigation space are present in the navigation structure model. However, they are extended with appropriate access structures. Note, for example, that the simple navigation paths between the `Book` and `Author` navigation classes are now modeled with indexes.

The *Search* entry of the main menu points to a query. A *«query»* stereotype is shown as a box with a question mark inside. The result of the query is an index that shows books ordered by their title and authors ordered by their name. Note that the *«query»* stereotype is a model element that is not provided by the RMM or the OOHDM, thus the UWE is the first in the list of introduced Web engineering methods that is capable of modeling the search feature of the Book Portal application.

The *Books of the Day* entry of the main menu is not supposed to be a menu entry at all. The Book Portal application shows a guided tour of popular books on the portal page. A *«guided tour»* stereotype is depicted with a rectangle and an arrow inside it. As the UWE approach does not support the modeling of user interface pages, the guided tour has to be modeled standalone and must be referred to from the main menu.

3.4.3.4 Business Process Modeling

In [KKCM03][KKZH04], the UWE method is extended to consider business processes. However, the approach presented is not really a mature solution to support business process modeling, i.e., it does not allow to create complex workflows and it does not employ a powerful workflow language. It is rather an acknowledgement that Web applications are not merely about presenting content through simple navigation structures. Consequently, the UWE method provides a few stereotypes that define application-logic-relevant components, which may be inserted into the navigation model of the Web application and employs UML activity diagrams with a stronger focus on application behavior.

The first extension of the method for modeling application logic are two stereotypes, which may be used to enhance the *Navigation Stucture Model* of the Web application. The «*process node*» stereotype may be used to define new nodes in the navigation structure that specify application logic components and do not correspond to any content model classes. Note that previously the UWE method allowed to create only navigation nodes that were directly derived from a content model class. An example regarding the Book Portal application may be the *Create Review* process node, which may provide functionality for creating a book review. Additionally to process nodes, the «*process link*» association stereotype may be used to connect them to common navigation nodes. Consequently, a process link may be used to connect the *Create Review* process node to the `Book` navigation node.

The second approach of the UWE method to support business processes, is utilizing UML activity diagrams that have a stronger focus on application logic. However, the UWE method already utilizes this diagram type in the requirements analysis phase, thus this extension is more of a paradigm shift than a new modeling approach. Note that the example activity diagram in Figure 3.12 has already a focus on application logic. It defines a process to create a piece of content, i.e., a book review, which may be very well a part of the business process implemented by the Web application.

3.4.3.5 Presentation Modeling

The last design activity of the UWE method is presentation modeling. To this end, the method provides a *Presentation Model*, which defines how the navigation structure of the Web application is exposed to the user. The model provides for each navigation node and access structure of the navigation model an appropriate presentation.

The aim of the presentation model is to define the structural organization of user interface elements rather than their visual appearance. To achieve that, the UWE method provides a set of atomic user interface elements that may be combined to build more complex composite user interface structures. A basic building block of the model is the «*frameset*» stereotype. It may be used to define arbitrarily nested composition structures as a frameset may contain other framesets and also any other element of the model. Another basic element is the «*presentation class*» stereotype. It is used to define the presentation of an entire navigation node and may also contain subelements, however no framesets or other presentation classes. Additionally to a frameset and a presentation class, the model provides some further elements that specify the presentation of content, access structures, and interactive user interface components. For example, self-explanatory stereotypes that represent content are the «*text*», «*image*», or «*video*» stereotypes. The «*collection*» stereotype may be used to define a list of elements of the same type, e.g., a list of images. Access structures may be represented with the «*anchor*» or «*anchored*

collection» stereotypes. The former is to be used if a single link to a certain navigation node is required. The latter corresponds to an index and defines links to a set of navigation objects of the same type. Finally, the «*form*» and «*button*» stereotypes may be used to include further interactive components into the user interface. A form allows the user to provide input data and a button represents an interface component, which may be used to initiate a navigation step, to execute an operation or both. Figure 3.16 depicts the presentation specification of the `Book` navigation class with the `Book` presentation class.

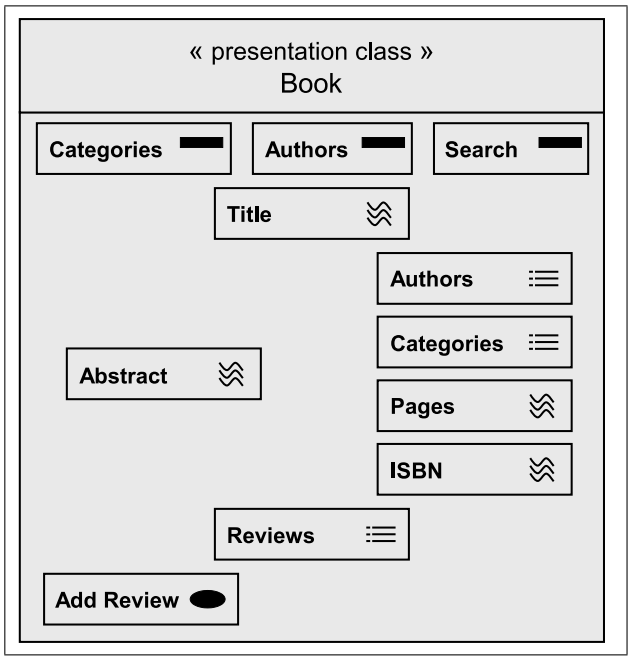

Figure 3.16: The Book Presentation Class for UWE

The presentation of a `Book` object is rather self-explanatory. The layout is defined with a `Book` «*presentation class*» stereotype. The presentation class includes a set of user interface elements that specify information about a book and access structures to other navigation nodes of the Web application. At the top of the layout, the main menu includes the *Categories*, *Authors*, and *Search* entries. Detail information about a book, e.g., the title, the abstract, or the number of pages, is specified with «*text*» stereotypes. Indexes of other objects, e.g., book authors, categories, and book reviews, are indicated with «*anchored collection*» stereotypes. Finally, the *Add Review* «*button*» stereotype specifies a button that leads to a section of the Web application, which provides the functionality to create a book review.

3.4.3.6 Tool Support

During the development of the UWE method, there have been several efforts to provide tool support for the method. The aim of the intended support was twofold. First, the tool had to provide features for creating and storing the graphical models. Second, it was supposed to support the generation of an implementation. Whereas the first task could be tackled with the ArgoUWE CASE tool [KKMZ03], the second one has been never achieved.

The first approach to generate an implementation from UWE models was the UWEXML framework [KK02b]. It utilized a code generator tool that received different UWE models, which were rendered as XML files. From these XML definitions, the tool produced a partial

implementation for the Cocoon XML publishing framework [URI08a] that was embedded into a J2EE setting. While the tool was able to produce a content storage, it was not suited to generate a fully functional user interface of the Web application. Besides the inability to produce a fully functional implementation, the approach had the disadvantage that the models had to be created in XML. Thus the graphical notations proposed by the UWE method had to be transformed into an XML representation manually.

To avoid this problem, the open-source ArgoUML modeling tool has been extended to support the graphical models of the UWE method. The extended tool is ArgoUWE [KKMZ03] and facilitates the creation of all diagrams of the UWE method. To this end, the ArgoUWE uses standard diagrams of ArgoUML, e.g., use case diagram, class diagram, etc., and extends them with UWE-specific stereotypes. An interesting feature of ArgoUWE is its capability to automatically derive navigation models from the content model. This is possible because there is a strong relationship between these models. Unfortunately, ArgoUWE is not capable of generating an implementation of the Web application, thus its main purpose is to facilitate requirements analysis and design through graphical modeling.

3.4.3.7 Conclusions

The UWE method is very similar to Web engineering approaches that had existed long before the method was proposed. In particular, it integrates several characteristics of the RMM and the OOHDM, which have been introduced in sections 3.4.1 and 3.4.2, respectively. However, the UWE method also brings along some new concepts and techniques for Web application modeling. Its main advantage compared to previous approaches for Web application development is that it relies on the UML as a graphical notation for all design activities. The UML provides a standardized and therefore familiar way for developers to model Web applications. One major deficit of the method (and of UML in general) is that different models capture different aspects of the Web application at different abstraction levels without specifying sufficient semantic connections to other models. Subsequently, advantages and disadvantages of the UWE method are discussed in detail comparing the method to the OOHDM.

Like the OOHDM, the UWE method did not support the *Requirement Analysis* phase of the Web application development process from the beginning. Both methods introduced this activity at some point during the methods' evolution process. The UWE method proposes the utilization of UML use case diagrams and UML activity diagrams for this purpose. A use case diagram specifies actors and usage scenarios that define in what ways a user may interact with the Web application. Unfortunately, use case diagrams are usually far too abstract to be useful for capturing detailed requirements or to be used to derive characteristics of the Web application regarding its content or navigation structure. An activity diagram specifies a set of actions that may characterize the behavior of the Web application more precisely than a use case diagram. As a matter of fact, the UWE method proposes to create one or more activity diagrams for each use case. However, the suitability of activity diagrams for directly deriving design models from them depends strongly on the developers ability to find an appropriate abstraction level for actions. For example, in an e-commerce setting, the action *Buy Product* may be too abstract, whereas the action *Look at Product Name* may be too specific. In contrast to the UWE method's approach, the OOHDM's *User Interaction Diagram* (see Section 3.4.2.1) seems to be more appropriate to capture a first draft of the requirements specification. It defines a set of user interaction states and, for each state, it specifies what user input is expected and

what information to the user is provided. This model is more suitable to indicate content and navigational requirements of the Web application.

Like the OOHDM, the *Content Model* of the UWE method is defined with a standard UML class diagram. However, it is to mention that the OOHDM switched to UML-support for content modeling after the UWE method was proposed, thus this is a feature that has been borrowed by the OOHDM from the UWE method and not vice versa. Ultimately, a UML class diagram is well suitable to specify the informational need of the application domain. The same or similar notations are employed by many Web application development methods. However, the UWE method seems to better exploit the capabilities of the class diagram. It uses more features, e.g., role names for associations, and it defines more concrete operations.

Navigation design is supported by the UWE method with two models. The *Navigation Space Model* is very similar to the OOHDM's *Navigation Class Diagram*, however, it provides less features. The UWE method keeps this model very simple. Content model classes that are relevant for navigation are included into the navigation model and become navigation classes, which represent nodes of the navigation space. Associations between content model classes are used to derive direct navigation paths between navigation nodes. In contrast to the OOHDM, the UWE method does not use navigation classes to define a filtering of attributes that are relevant enough to be presented on the user interface. Additionally, it is not clear if this model supports specialization/generalization. Unfortunately, all publications about the UWE method provide very simplistic examples without inheritance hierarchies, if it comes to navigation modeling.

The second model of the UWE method for navigation design is the *Navigation Structure Model*. This model is based on the navigation space model and extends it with a number of common access structures, e.g, menus, indexes and guided tours. Note that this model is very similar to the RMM's *Relationship Management Diagram* (see Section3.4.1.3). Also note that similarly to the RMM and the OOHDM, the UWE method only allows to define navigation nodes that directly correspond to content model classes. The specification of a custom navigation node that represents arbitrary information or correspond to a custom set of content classes is not possible. For example, none of these methods are capable of modeling the start page of the Book Portal application. This navigation node does not correspond directly to any content model class, however, it is an important member of the navigation space.

The UWE method employs the *Presentation Model* to define the presentation of navigation nodes and access structures that have been specified during navigation design. This model provides a set of presentation elements that may be used to construct the parts of the user interface piece by piece. Again, this model is very similar to the OOHDM's *Abstract Interface Design*. However, similarly to navigation design, it does not allow for the definition of custom components, e.g, user interface pages with arbitrary elements.

3.4.4 WebML

Like all other Web engineering methods that were introduced towards the end of the 1990s, the initial focus of the Web Modeling Language (WebML) [CFB00] was on modeling data-intensive Web sites. Since then the method has been extended to support data-entry operations [BCFM00], business processes [Bra06] [BCFM06], and semantic Web applications [Bra07b]. WebML and the WebRatio [URI08b] commercial toolkit are being constantly developed by a considerable number or researchers and developers. Accordingly, the WebML is the most comprehensive approach for Web application development originating from the Web engineering

research field.

The WebML has been inspired by previous Web engineering methods, especially the RMM and the OOHDM. Similarly to these methods, the WebML provides different models to capture different aspects of the Web application. The *Data Model* defines the Web application's content structure with an Entity-Relationship diagram. The *Hypertext Model* subsumes two further models and expresses them in a single diagram, which utilizes the proprietary WebML notation. First, the *Composition Model* specifies pages and different presentation units that yield the basic structure of the Web application's user interface. Second, the *Navigation Model* defines different access structures and connects pages and presentation units with each other, thereby specifying the Web application's navigation structure. Finally, the *Presentation Model* defines the actual visual appearance of user interface elements.

Note that in contrast to other Web engineering methods, the WebML pursues a different paradigm for combining the content model, the navigation model, and the Web application's user interface model. Previously introduced methods employed the notion of a navigation space, in which navigation nodes were defined as more or less flexible views over content objects and navigation paths corresponded strictly to relationships between these objects. Additionally, the aim of presentation modeling was to specify a visual representation of navigation nodes without the possibility to introduce any custom user interface elements. In contrast to that, the WebML does not restrict user interface modeling to the concept of navigation nodes. It utilizes the concept of user interface pages that are initially independent from content or navigation concerns and may be constructed piece by piece through combining a set of basic user interface elements, which may refer to arbitrary content objects or specify arbitrary navigation structures.

As stated in [RPSO08], the WebML method favors an iterative development process. For each iteration, the method proposes the following set of phases. First, the aim of the *Requirements Analysis* phase is to gather informational and functional requirements for the desired Web application. However, the method does not provide any models or guidelines for this activity. Second, the *Conceptual Modeling* phase includes the creation of the *Data Model* and the *Hypertext Model*. The data model is derived from the informational requirements and defined with an Entity-Relationship diagram. Hypertext modeling includes the definition of user interface pages, main presentation units, and basic access structures, which may be extended and refined in subsequent iterations. Third, in the *Implementation* phase the actual Web application is constructed using an appropriate technological infrastructure. Finally, in the *Testing and Evaluation* phase the quality of the application is evaluated. Depending on the result of this phase, the development process enters a new development cycle, is deployed at the customer's infrastructure, or both.

3.4.4.1 Data Model

Similarly to the RMM, the WebML employs an Entitiy-Relationship diagram to capture the *Data Model* of the Web application. However, it utilizes a larger subset of E-R features than the RMM and it provides a custom graphical notation. The model specifies entity types and attributes, which are characterized by data types. This is a WebML-specific extension, because data types are usually not of concern in E-R modeling. Besides standard attributes, an entity type may also specify *derived* attributes, which represent computed values. The WebML also marks for each entity a set of attributes that identify an entity unambiguously, i.e., a key. Enti-

ties may be connected with relationships, which are using a consistent naming scheme for role names. For example, the relationship between the `Book` and `Author` entity types is characterized by the role names *BookToAuthor* and *AuthorToBook* corresponding to both directions of the relationship. This naming scheme is used in other models to identify relationships between entity sets. Additionally, multiplicities may be used to specify how many partners a certain entity may have considering a certain relationship. Note that in contrast to the RMM, the WebML also allows specialization/generalization hierarchies. A small disadvantage of the E-R modeling notation that is used by the WebML is its inability to specify describing attributes for relationships. The WebML suggests to create an extra entity that represents the relationship and contains the corresponding attributes. Figure 3.17 depicts the data model of the Book Portal application.

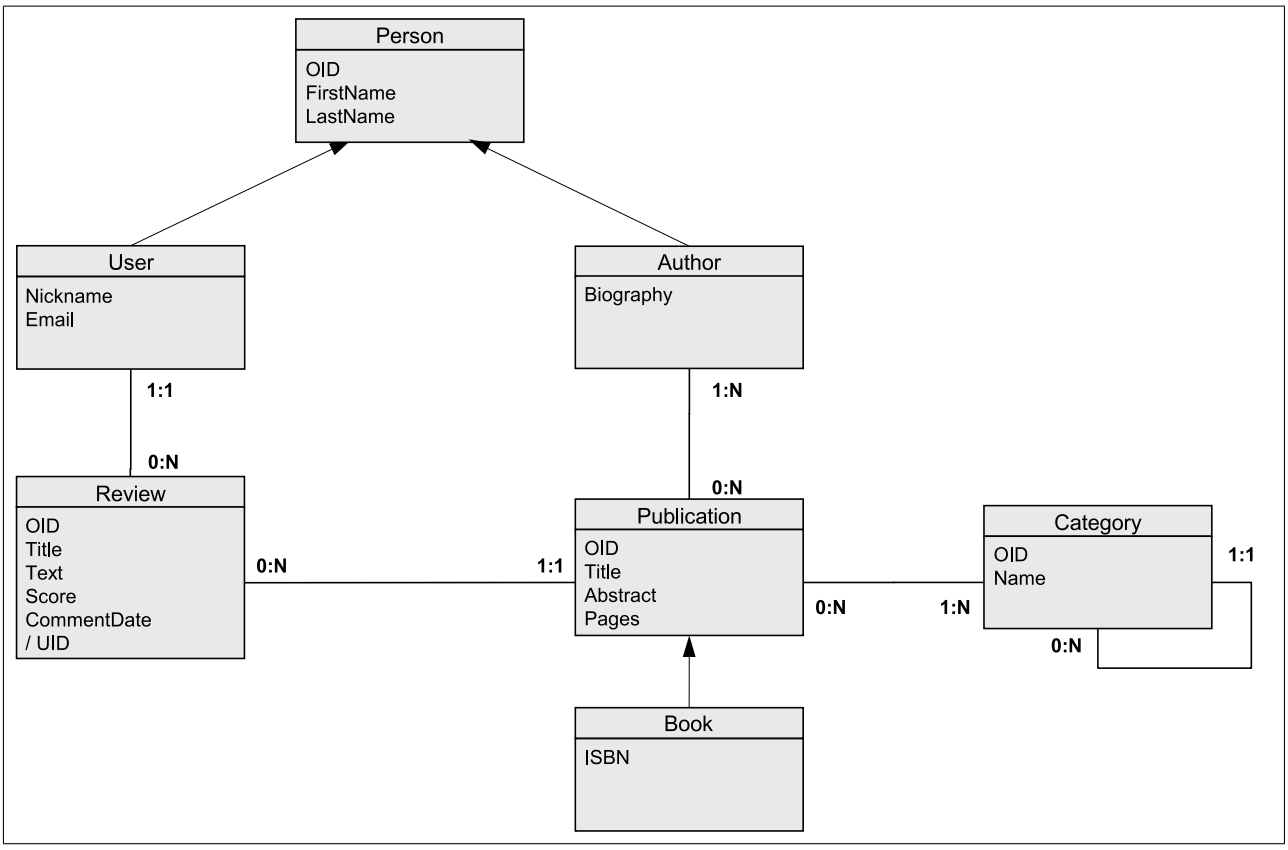

Figure 3.17: Data Model of the Book Portal for WebML

The WebML uses a custom graphical notation to define the E-R model of a Web application. It is different from the original E-R notation [Che76], especially regarding how it depicts attributes. It places them inside the box that represents an entity type making the notation very similar to the syntax of a UML class diagram. Besides some minor differences, this model is very similar to the content model of other Web application development methods, e.g., the OOHDM (see Figure 3.7) and the UWE method (see Figure 3.13). The most obvious difference is that the *CommentDate* attribute that actually belongs to the *Comment* relationship is modeled as an attribute of the *Review* entity. The WebML suggests to model attributes that characterize a relationship with an extra entity between the original entities of the relationship. However, this is a bad solution because it results in very complicated content management patterns, if it comes to creating corresponding relationship instances. The UML provides the concept of

an association class which is in contrast to the WebML's E-R notation well suitable to define relationship attributes. Note that, although the WebML uses the notion of attribute data types, these are not represented in the graphical notation of the data model.

3.4.4.2 Hypertext Model

The aim of WebML's *Hypertext Model* is to define the Web application's user interface, which usually heavily relies on hypertext and hypermedia. To this end, the method provides a model and a proprietary graphical notation, which may be used to specify two different aspects of the user interface. First, how user interface pages are composed of elementary content units. This is called the *Composition Model*. Second, how content units and pages are interconnected with navigation links and more complex access structures. This is called the *Navigation Model*. Although the WebML states that both aspects are covered by the hypertext model, it does not try to separate these aspects explicitly.

The WebML method applies four concepts to structure the hypertext model. A *Site View* is a complete hypertext that represents a certain aspect of the Web application or corresponds to certain requirements. For example, a designer may identify different user groups and define for each group a site view, which is in that case a customized user interface that considers all requirements of that user group. Site views may be split up into disjoint areas using the *Area* concept. An area groups pages, which build a coherent unit of the user interface, e.g., represent information about the same topic or provide a certain functionality of the Web application. *Page* components represent nodes of the user inteface and may contain an arbitrary number of content units. Finally, a *Content Unit* shows information about one or more entities of the data model.

Content Units The WebML provides five elementary content units. The *Data Unit* represents information about an entity and allows the selection of those attributes that are relevant for presentation. The *Multidata Unit* represents information about multiple entities. It also allows to select relevant attributes. The *Index Unit* presents descriptive keys of a set of entities and allows the selection of one or more entities. Note that this definition of an index is different from the definitions used by other Web engineering methods, because usually only the selection of a single entity is allowed. Besides the basic index, that allows the selection of a single entity out of a linear list of entities, the WebML provides two further indexes. First, the *Hierarchical Index* represents a nested, tree-like index structure with an arbitrary number of nesting levels. Second, the *Multichoice Index* allows the selection of several entities from a linear entity list. The *Scroller Unit* represents navigation elements that allow to browse through a set of entities. For this unit, a so-called *block factor* may be defined that specifies how many entities are presented to the user at the same time. Note that a scroller unit alone does not represent any content. It has to be connected to some other element, e.g., a data unit or a multidata unit that shows information about the entities, which were selected by the scroller unit. Finally, the *Entry Unit* represents a form that may be used by the Web application's user to provide input data.

Besides the entry unit, all other content units have to be characterized by a *source* and a *selector*. The source identifies the type of entities, which should be presented by the corresponding content unit. The selector provides a condition that selects a subset of entities of the source entity type. A selector may use basic arithmetic and Boolean operators to combine entity attributes, references to relationships, constants, and parameters. A selector that includes

parameters is called a *parametric selector*. A parameter to the selector of a content unit is usually provided by a navigation link. For example, a multidata unit may show books of the certain author. To this end, the multidata unit may specify the "Book" type as the source and the parametric selector "AuthorToBook(OID)". The *OID* of the author may originate from a navigation link that, for example, allows navigating from an author entity to the list of the author's books. The textual definition of the source and the selector are provided below the icon of a content unit in the graphical notation of the hypertext model.

Content units that represent information about one or more entities, e.g., a data unit or an index unit allow to specify a set of entity attributes that are relevant for presentation. An enumeration of selected attribute names may be added to the textual definition of the unit. Content units that show several entities of a certain type, e.g., the multi-data unit and the index unit may also define the order of entities. The order may depend on one or more attributes of the corresponding entity type and may have the direction ascending or descending.

Links Content units mostly represent information about one or more entities of the Web application's content. To define the actual navigation structure of the application, the WebML employs the notion of *links*. Links connect content units and pages with each other and specify how the user may navigate between them. Links are also able to transport information between components. The WebML defines three different link types. A simple *Link* connects content units and pages with each other. The source and the destination of a link may be content units on the same or on different pages. If the source of the link is a content unit, then the link may transport an arbitrary number of parameters to the target. Depending on whether the source content unit represents a single or multiple entities, the link may transport in each parameter a single value, e.g., the value of a certain attribute of the entity, or a set of values, i.e., the values of a certain attribute of all entities. The graphical notation of a link is a solid, directed arrow. The parameter specification includes the name of the parameter and the name of the selected attribute and is displayed as an arrow label.

The second link type is a specialization of a simple link. The *Automatic Link* is utilized to connect two content units on the same page and to automatically transport an initial value if the user visits the page. This makes, for example, sense if the source of the link is an index unit and the target is a data unit. The designer may use a simple link to connect these components, thus, if the user selects an entity from the index, it is displayed by the data unit. However, if the user just got to the corresponding page, then he did not have the opportunity to interact with the index and the data unit cannot display anything. For such cases, it is recommendable to use an automatic link instead. The automatic link makes sure that, if the user visits the page, a certain entity from the index is automatically selected and transported to the data unit without any user interaction, thus the data unit may display this initial value. The default is to display the first entity of the index. The graphical notation of an automatic link is like of a simple link except for a small box containing the letter "A" at the middle of the link arrow.

The last link type is the *Transport Link*. Its aim is to transfer values between two content units that reside on the same page without any user interaction. This link type is used if a content unit requires some information independently of the user's navigation behavior. For example, if a data unit on a page displays information about a certain entity, a multi-data unit may display information about a set of related entities, thus a transport link may connect these two units to transport the identification of the data unit's entity to the multi-data unit. The graphical notation of a transport link is a dashed arrow.

The WebML specifies the user interface of the Web application more precisely than previously introduced Web engineering methods. The hypertext model of the entire Book Portal application is too large to be depicted in one figure. Therefore, specific characteristics of the WebML graphical notation are explained with a set of partial examples.

Figure 3.18 depicts the definition of the Book Portal's main page. Note that among all Web engineering methods introduced in this chapter, only the WebML is capable of modeling a version of this page that comes close to the actual requirements. The reason for that is the inablity of all other methods to define user interface nodes, that do not correspond strictly to any entity of the content model.

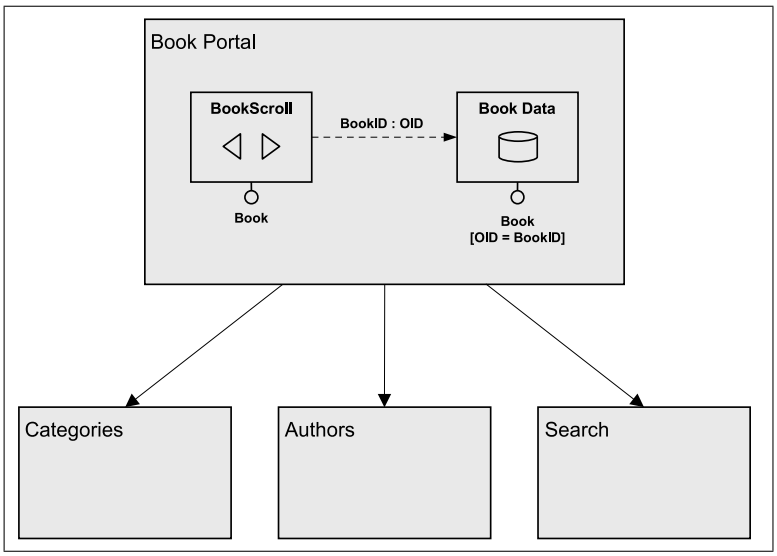

Figure 3.18: Hypertext Model of the `Portal` Page for WebML

The portal page ought to show a guided tour of popular books regarding the average review scores provided by portal users. Unfortunately, the expressive power of WebML selectors are not sufficient enough to express this query. Instead, the WebML specification of the portal page in Figure 3.18 shows a guided tour of all books. To this end, the `Book Portal` page includes two content units. The `BookScroll` scroller unit on the left-hand side specifies the navigation interface for browsing through all books of the portal. The source specification below this unit states that entities of the `Book` type are required. The unit has no selector, thus all books are considered. The scroller unit is connected to a data unit with a transport link. The link transports the *OID* of the currently selected book in the *BookID* parameter to the data unit. The source specification of the data unit defines that the unit displays information about `Book` entities. Additionally, the selector of the unit states that the *OID* of the displayed book must be equal to the value that is transported by the link in the *BookID* parameter. Consequently, the data unit shows information about the book that is currently selected by the scroller unit.

The `Book Portal` page is also connected to the `Categories`, `Authors`, and `Search` pages with simple navigation links. Unfortunately, the WebML does not define model elements that correspond to the notion of a menu or an anchor. Thus, the main menu of the Book Portal has to be modeled with simple links to corresponding pages.

A somewhat more complex example is depicted in Figure 3.19, which shows the hypertext model of the `Book` page.

The `Book` page contains the `Book Data` data unit that displays the book's title, abstract,

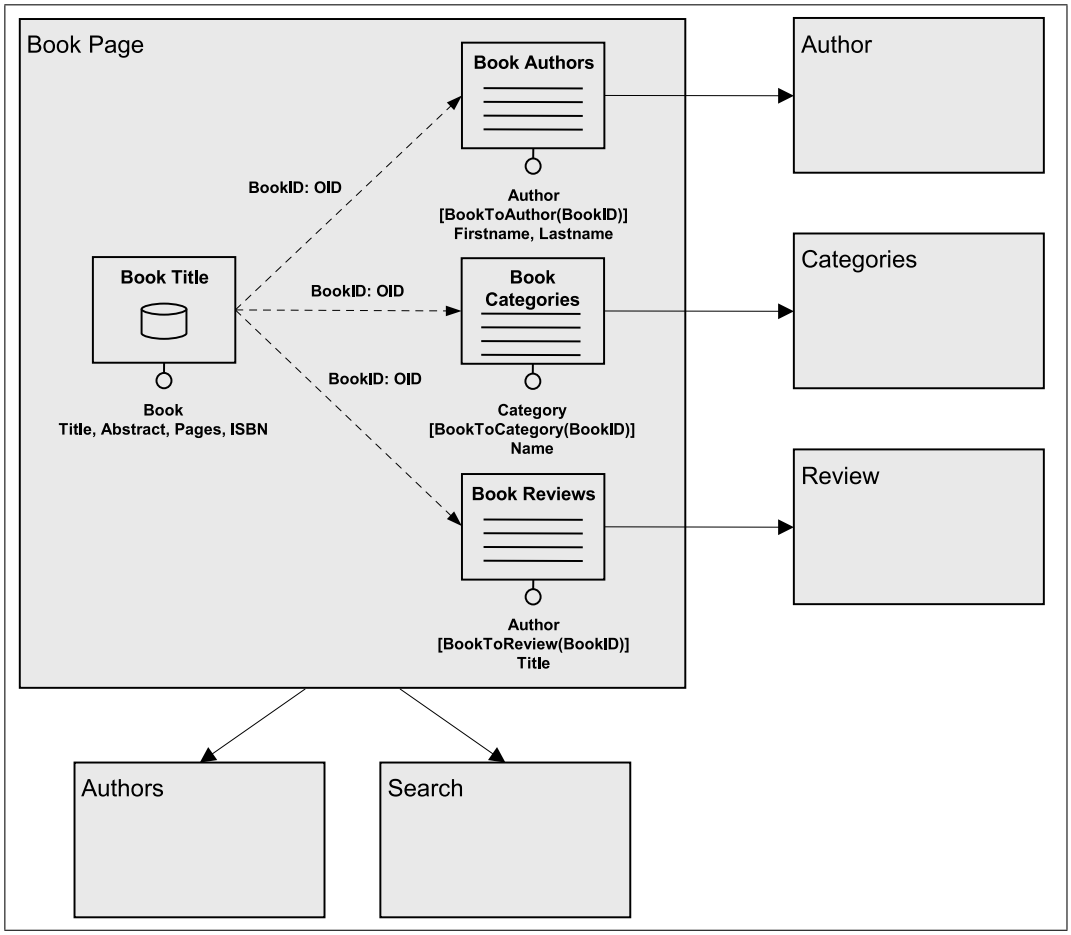

Figure 3.19: Hypertext Model of the `Book` Page for WebML

number of pages, and the ISBN. Of course, the page can only display data about a book if it is clear which book entity is to be displayed. To this end, the selector of the data unit specifies that the *OID* of the book entity has to be equal to the value of the *BookID* parameter that is transported by the incoming link of the page. Note that the identification of the book is forwarded to the three indexes of the `Book` page by corresponding transport links. The indexes represent the book's authors, categories, and reviews. Each index defines a source and a selector that corresponds to the information to be shown. Additionally, the indexes specify which entity attributes are to be presented as keys of the index. For example, the source specification of the `Book Authors` index states that the index presents `Author` entities. The selector of the index references the *BookToAuthor* relationship and selects those `Author` entities that have a corresponding relationship to the `Book` entity with the id *BookId*. Finally, the attribute specification states that only the *Firstname* and *Lastname* attributes of each author are to be shown as keys of the index. Note that all indexes are connected with navigation links to corresponding pages of the Book Portal, however, link parameters and the detailed specification of all pages is omitted for the sake of simplicity.

Content Publishing Patterns The WebML defines a small set of basic content units, which may be used to represent content on the Web application's user interface. In contrast to most other Web engineering methods, the WebML does not employ concrete model elements that

correspond to well-known presentation patterns, e.g., a guided tour. Instead, it suggests to compose these patterns using its elementary content units and calls them *content publishing patterns*. Subsequently, some of the most common patterns are introduced.

The *Cascaded Index* pattern allows the Web application's user to browse through a set of indexes with different content focus to finally reach the presentation of a desired entity. This pattern is utilized several times by the Book Portal application. For example, the user is allowed to select a book from the book index on the `Categories` page and after that an author on the `Book` page, which leads him to the `Author` page, where information about the author entity is presented. The WebML implements this pattern with two consecutive index units and a data unit. The book index allows the selection of a book and transports the identification of the book to the author index that employs an appropriate selector to present only authors of the selected book. Finally, the author index transports the identification of the author to the data unit that shows information about the author.

The *Filtered Index* pattern allows the user to conduct a search and select an entity from search results index in order to view information about the entity. This pattern also is employed by the Book Portal application. The `Search` page provides the user a search form that leads to the `Search Results` page, which presents an index of books that correspond to the user's query. After selecting an author entity from the index, the user is presented information about it. This pattern may be defined with WebML in a straightforward manner with an entry unit, an index unit and a data unit. The entry unit receives user input and forwards it to the index unit, which employs an appropriate selector to present authors that match the user's query. Finally, the index transports the author's identification to the data unit to present author information.

The *Guided Tour* is a commonly used pattern in Web application development and allows the user to browse through a set of entities one by one. The Book Portal application utilizes this pattern to present books on the `Book Portal` page. This pattern may be composed with a WebML scroller unit and a data unit. To this end, the scroller unit may be configured to allow browsing all books of the portal, i.e., specify "Book" as source and no selector. Additionally, the scroller unit is connected to the data unit with an automatic link (see Section 3.4.4.2) so that the identification of the currently selected `Book` entity is transported to the data unit, which presents the corresponding book.

Note that the WebML defines a few further content publishing patterns that represent mostly variations of previously introduced patterns [CFB+03]. For example, the *Filtered Scrolled Index* is a filtered index, which additionally splits up search results into partitions of the same size and allows to browse between partitions. Similarly, *Indexed Guided Tour* is an extension of a simple guided tour and additionally provides an index in front of the standard guided tour functionality thus the user may first select an entity from an index before browsing through entities one by one.

Operation Units Among all Web engineering methods introduced in this chapter, the WebML is the only one that supports the design of Web applications with content management functionality. A typical Web application of this class does not only provide information based on entities and relationships but also allows the user to create, to modify, and to delete entities and relationships between them. To this end, the WebML provides a set of *Operation Units* that may be used to specify content management functionality [BCFM00][CFB00][CFB+03]. The WebML inserts operation units into the hypertext model, thus, corresponding functionality is well integrated into the Web application's navigation structure.

All operation units of the WebML have some common characteristics. First, they require certain input data that is to be processed by the corresponding operation. For example, an operation that creates a new entity certainly needs valid values for all attributes of the entity. To this end, operation units always have one or more inbound links that transport required parameter values. Second, there are two ways the execution of an operation may come to an end. It may succeed or fail. For both possibilities, there is a corresponding link type, i.e., an *OK* link and a *KO* link, which may be used to direct the Web application's user to an appropriate page after the execution of the operation. Subsequently, most important WebML operation units are introduced.

The *Create Unit* allows to create a new entity of a certain type. To this end, the *source* specification of the unit specifies the name of an entity type of which a new entity is to be created. The create unit needs at least one inbound link that delivers all required attribute values. Usually, these values originate from an entry unit. The create unit also provides a set of *assignments* that map names of link parameters to attribute names. In case of the create unit, the OK link usually points to a page that presents the newly created entity and the KO link guides the user back to the entry form.

The *Modify Unit* allows to modify attribute values of one or more entities of a certain type. The *source* specification identifies the entity type of which entities are to be modified. The modify unit requires two kinds of input data. First, data that may be used for the identification of those entities that should be modified. For example, an inbound link may transport the ID of a single entity or just a simple value that is matched against all values of a certain attribute of all entities to select an appropriate entity set for modification. Second, one or more values that serve as new attribute values of selected entities. Parameter names of inbound links are mapped to attribute names of the corresponding entity type with a set of *assignments*. The OK link of the modify unit usually leads to a page that presents the modified entities and the KO link usually points back to the entry form, from which new attribute values for the modification originate.

The *Delete Unit* may be used to delete one or more entities of a given type. The entity type is determined as usual by the unit's *source* specification. The delete unit requires data that is delivered by an inbound link and can be used to determine those entities, which are to be deleted. To this end, inbound link parameters may deliver the IDs of entities for direct identification or some values that can be used in a *selector* expression to determine the appropriate set of entities. The OK link of the delete unit usually leads to a page that lists other entities of the given entity type thus the user may observe the absence of the deleted entity. In most cases, the KO link leads back to the page, on which the user initiated the delete operation.

Besides operations that allow to create, to modify, or to delete entities of the Web application's content, the WebML also defines operations for the management of relationships between entities. The *Connect Unit* creates a new relationship instance between two entities. To this end, the unit requires the identification of both entities between which the new relationship is to be created. Usually, this identification is provided by inbound links that originate from a page displaying information about both entities. Note that the WebML does not support attributes that characterize relationships, thus the connect unit does not require any further input values. The *source* specification refers to one of the two role names of the corresponding relationship. For example, if the designer intends to define an operation that creates a relationship between the `Author` and the `Book` entities then the corresponding connect unit may use the *AuthorToBook* role name as source specification. The OK link of the unit may point to a page

that presents information about the connected entities and the KO link may lead back to the page the user came from.

The *Disconnect Unit* deletes an existing relationship instance between two entities. Similarly to the connect unit, it requires as input the identification of those entities between which the relationship is to be removed. As in the case of other operation units, input parameters are transported to the unit by inbound links, which, for example, may originate from a page that displays information about both entities in question. The *source* specification identifies the relationship of which an instance is to be deleted unambiguously. Analog to the previous example, the *AuthorToBook* role name may be used as source specification to define a disconnect operation for the relationship between the `Author` and the `Book` entities. The OK and KO links of the unit may point to the page the user came from.

Note that among all presented Web engineering methods of this chapter, the WebML is the only one that is able to define the `Manage Review` page of the Book Portal application, because this page relies on content management functionality. Figure 3.20 depicts the WebML specification of the `Review` and *Manage Review* pages, which allow to view reviews and to associate reviews to books.

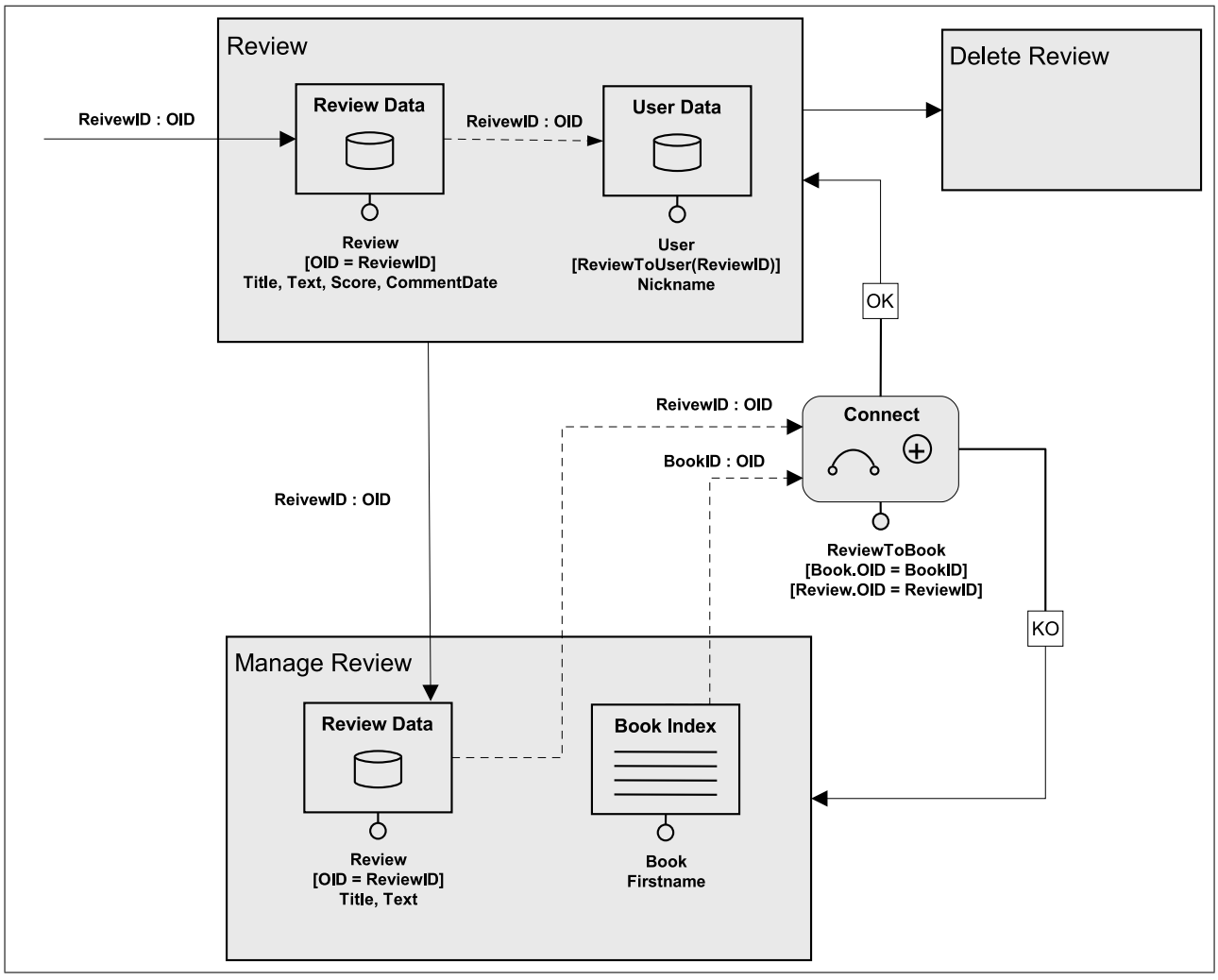

Figure 3.20: Review Management Specification with WebML

The `Review` page contains two data units that present information about a review and also

includes links to other pages, which are concerned with review management, i.e., the `Manage Review` and the `Delete Review` pages. The *Review Data* unit receives the id of a `Review` entity in the *ReviewID* parameter of an inbound link and displays some information about the review, e.g., the title, the review text, the score, and the date of the review. A transport link automatically forwards the id of the review to the *User Data* unit that displays the nickname of the user who created the review.

The *Manage Review* page, which is accessible through a link from the *Review* page, provides functionality for associating a review to a book. To this end, the page contains a *Review Data* unit that displays the review title and the review text and a *Book Index* unit that presents an index of all books of the portal. Both units are connected to the *Connect Review* operation unit that creates a new relationship instance between a `Review` entity and a `Book` entity. To this end, the id of the review and the book are forwarded to the connect operation unit by corresponding links originating from the *Review Data* unit and the *Book Index* unit. Note that the *ReviewID* and *BookID* parameters are used by the connect unit to identify the entities to be connected. Finally, the OK link of the connect unit points to the `Review` page and the KO link leads back to the `Manage Review` page.

Note that the WebML also allows to combine operation units in order to define more complex content management patterns. For example, a very common pattern is the creation of an entity and its subsequent assignment to an already existing entity with a newly created instance of a certain relationship. Note, however, that the graphical notation of the WebML may become confusing if the content management patterns get more complex.

3.4.4.3 Implementation

The WebML does not provide a graphical presentation model that determines the look and feel of the Web application's user interface. Presentation artifacts are produced during the implementation phase of the WebML development process. The method employs a template-based, generative approach to create the Web application's implementation which consists of four phases. First, the hypertext model is used to automatically generate an XHTML template of each page. This template is a page skeleton that includes an initial representation of all page components. Second, the presentation for each content unit is created manually by a graphic designer using CSS. Third, XSLT technology is used to automatically insert these presentation definitions into the generated templates. Finally, an application programmer extends the generated Web application with custom functionality.

3.4.4.4 Conclusions

The WebML has been inspired by previous Web engineering methods like the RMM and the OOHDM. However, it introduces a new way to define the hypertext structure of the Web application. Whereas other methods focus on capturing the Web application's navigation structure at a rather conceptual level, the WebML provides a hypertext model that employs model elements with a lower abstraction level, which facilitate the exact composition of the Web application's user interface and the generation of an implementation. Also, the WebML manages to integrate operations into its hypertext model which enables the approach to define Web applications with content management functionality. Subsequently, advantages and disadvantages of the WebML method are presented and compared to other Web engineering methods.

Similarly to the RMM, the WebML employs E-R modeling to define the Web application's *Data Model*. Unfortunatelly, the E-R notation used by the WebML has the disadvantage that it cannot express relationship attributes. Other than that, the expressive power of the WebML's content model is equivalent to that of other methods.

The WebML's *Hypertext Model* pursues a unique way of defining the Web application's user interface and navigation structure. All other Web engineering methods introduced in this chapter focus on the definition of the navigation structure on a conceptual level and not on the construction of the user interface. To this end, they utilize the notion of a navigation space containing navigation nodes and they define different access structures, e.g., indexes or guided tours which connect navigation nodes and strongly rely on relationships of the content model. Consequently, an index may only list all objects of a certain type or all objects that have a certain relationship to an object that is in focus of a specific navigation node. For example, if the Web application's user views information about a book provided by a corresponding *book* navigation node, then this node may provide an index to all authors of the book relying on the *author writes book* relationship between these objects. However, more complex filtering of objects to be shown in an index is not supported. Furthermore, as there may be only one *book* navigation node in the navigation space, only a single index that shows book authors may exist in the entire Web application, because the index is bound to the book context.

In contrast to that, the WebML's approach to define the Web application' user interface and navigation structure is much more flexible. It does not rely on the concept of navigation nodes that have an associated context. It utilizes the concept of user interface pages, which are initially context independent. Thus, a single page may represent information about several entity types of the content model. Especially, the WebML allows to define several pages that present information about the same entity. It also allows to define custom links between pages, thus, the Web application's navigation structure is independent from its content structure. Furthermore, the WebML utilizes a simple query language that determines entities, which are to be presented by content units. The query language has access to attributes and relationships of entities and may compare attribute values and constants using arithmetic expressions. Last but not least, the WebML hypertext model provides operation units that may be used to define Web applications with content management functionality. Other Web engineering methods introduced in this chapter define operations at a very abstract level and do not integrate them properly into the navigation structure of the Web application.

Unfortunately, the WebML way to model the Web application's user interface and especially its approach to integrate operations into the hypertext model has also some disadvantages. First, the WebML concentrates on the definition of the user interface and defines several link types to connect elements of the hypertext model. Some link types denote a navigation step by the Web application's user, some other specify merely the transport of values between model elements. Consequently, the navigation structure of the Web application may become unclear. To make things worse, the WebML defines content management operations directly in the hypertext model. As operation units require a number of inbound and outbound links, the navigation structure becomes even more concealed.

All together, the WebML provides the most powerful graphical notation among all presented methods in this chapter. Especially, the low abstraction level of the hypertext model for defining the Web application's user interface enables the generation of large parts of the implementation, which has to be modified only marginally to get a fully functional application prototype.

3.4.5 Summary

The four methods that have been introduced in previous sections build a representative selection of model-based Web engineering methods that have been proposed by the Web engineering community. The following table shows an overview of the models that are proposed by these methods for designing Web applications.

	Content	Navigation	Operations	Presentation
RMM	"Entity-Relationship Model"	"Relationship Management Diagram"	-	-
OOHDM	"Conceptual Model"	"Navigation Class Diagram" "Navigation Context Diagram"	rudimentarily in "Conceptual Model"	"Abstract Data Objects" "Abstract Data Views"
UWE	"Content Model"	"Navigation Space Model" "Navigation Structure Model"	rudimentarily in "Content Model"	"Presentation Model"
WebML	"Data Model"	"Hypertext Model"	"Operation Units" in "Hypertext Model"	-

Table 3.1: Model Overview of Introduced Web Engineering Methods

The introduced methods have their individual advantages and weaknesses. A common feature among all methods is the utilization of a comprehensive content model, although all methods use a different name for this model type. The second model category deals with the definition of navigation structures. This is a well supported feature by most methods, but there are subtle differencences. The OOHDM and UWE methods identify nodes of their content models that are relevant for navigation and connect corresponding navigation nodes with standard access structures that are derived from relationships between content model nodes in a straightforward manner. The RMM also uses a set of standard access structures, e.g., indexes and guided tours, but it also provides the concept of m-slices that allow a fine-grained definition of what information appears on user interface pages. The WebML uses the Hypertext Model for navigation modeling, which is the most sophisticated navigation model among the introduced methods. It allows the definition of user interface pages including arbitrary content and arbitrary navigation structures. Unfortunatelly, all methods neglect to provide an operation model that allows to define comprehensive application logic. The RMM does not support operations at all. The OHHDM and the UWE methods allow to specify operations only at an abstract level in their content models, however, it is unclear how these operations correspond to user interface components. The WebML is the only method that has a reasonable concept for

defining operations by providing so-called *operation units*. Unfortunately, it integrates operation units into its hypertext model instead of defining a separate model for them. The OOHDM and the UWE methods also provide models for defining the presentation of the Web application. Their presentation models provide elements that define the appearance of user interface pages. The RMM and the WebML do not have explicit presentation models, however, they specify user interface pages in theit navigation models more precisely than the OOHDM and the UWE methods.

Ultimately, these methods provide more or less suitable models for designing Web applications, however, the overall design is only cohesive enough in the case of WebML to be used as a straightforward implementation guideline or as input for a CAWE tool that is able to generate large parts of the Web application's implementation. Further methods and tools that have been investigated but not selected to be discussed in detail are ADM [AMM98], ADM-d [MAM03] and ADM-2 [AP03], HDM [GPS93], HDM2 [GMP93], HDM-lite [FP00] and HDM2000 [BGP00], Hera [HFV03] and Hera-S [vdSHBC06], Homer [MAM+00], the OO-HMethod [GCP00] and OOWS [PAF01a][PAF01b] (both based on the OO-Method [PGIP01][PIP+97]), Strudel [FFLS97] [FFK+98][FFLS00][FFLS98], Fun-Strudel [FST99] and Strudel-R [FLSY99], Tiramisu [ALW99], W2000 [BGP01], and WSDM [TC03].

CHAPTER 4

Modeling Web Applications with flashWeb

The main contribution of this work is the introduction of the flashWeb model-based approach for Web application development, which is described in this chapter. The flashWeb method uses UML and UML-near notations, therefore, the semantics of the proposed models is in most cases intuitively comprehendible. UML is a formal language and it is common knowledge, thus, any UML-near notation is immediately familiar to software engineers. Despite of this fact, a set of additional formalisms are used throughout the following chapters in order to define the precise meaning of every model construct. To this end, the models as well as the model weaving constructs are described by a UML meta model at the beginning of each chapter. Additionally, composite operations in Section 4.4.2.2 are explained using an auxiliary graphical notation.

This chapter is structured as follows. Section 4.1 introduces the basic characteristics of the flashWeb method. Section 4.2 explains which models the method provides for which purposes and introduces the general idea of graphical connections between different models. Finally, each model of the flashWeb method is described in separate sections.

4.1 flashWeb Characteristics

As already indicated in Section 3.4 (see concluding subsections, respectively), most existing Web engineering methods have some deficiencies that are worth addressing. Accordingly, the flashWeb method tries to avoid the shortcomings of other methods and is based on a set of key principles that are introduced subsequently.

The first main characteristic of flashWeb is the extensive usage of **graphical models**. To this end, it utilizes different models that define different aspects of the Web application. For each model, the level of abstraction and the number of model elements are defined in a way that allows to specify a preferably maximal portion of the Web application leaving only a minimal part that has to be implemented in a customized manner. Note that, even for such custom parts, flashWeb models provide custom elements that integrate custom code into the models (see subsequent paragraph on extensibility). It is vital for the success of a Web engineering method to offer a set of model elements with sufficient expressive power and flexibility. Ap-

propriate graphical models can be easily transformed into a functional implementation by a code generator in a matter of seconds. In contrast to that, custom parts that have to be implemented and integrated into the system manually require orders of magnitude more time and resources. Therefore, every piece of the Web application that can be captured with a formal model and transformed into code automatically saves a lot of time and effort for the development team and makes the success of the development project more likely.

A major innovation of the flashWeb approach is the introduction of the novel *Operation Model*, which allows to capture all business logic operations of a Web application in a single model. The model provides for each entity of the target domain a set of standard content management operations, which allow simple access to the Web application's content. Furthermore, composite operations combine standard operations in order to compose composite pieces of application logic. Finally, custom operations enable the Web application developer to define arbitrary application logic. The Operation Model may be combined with other models of the flashWeb method in a very flexible manner. The model is introduced in great detail in Section 4.4.

An important characteristic of flashWeb is the high level of **model independence**. Of course, the models can be flexibly combined (see next paragraph) to create a cohesive specification of the Web application, however, the number of constraints that a model may lay upon another model is minimal. For example, flashWeb's Operation Model provides a standard set of content management operations for each class of the Content Model, but it is not restricted to this initial set. Additionally, it may include an arbitrary number of operation classes that provide custom or composite operations that are independent of the Content Model. The level of independence between the Operation Model and the Composition/Navigation Model, which defines the Web application's user interface is even greater. Model elements for user interface definition are generic in nature and may be arbitrarily combined with content management operations. Additionally, arbitrary user interface structures may be specified that do not rely on operations at all. This level of model independence is often not the case with other Web engineering methods. However, the need for flexibility is a key characteristic in every Web application development project. This is due to the fact that requirements for Web applications are diverse and they often change, even during a single development project. Therefore, the models in use have to provide the greatest level of flexibility. Any unnecessary constraints that originate from the development method or are imposed by the nature of the models on the development team result in a smaller coverage of the Web application's functionality and, as explained in the previous paragraph, in the consumption of unnecessary resources.

One of flashWeb's novelties is its high level of **model weaving** capability. Although the method's models are independent in a sense that they do not lay unnecessary constraints on each other, the models can be graphically combined to create a cohesive specification of the Web application. For example, a model element that defines an object listing on the Web application's user interface can be graphically connected (by an edge) to a content management operation that delivers the corresponding data. This kind of graphical connection between different models is a unique characteristic of flashWeb in the research field of Web engineering. Without having precise connections between models, a Web engineering method may apply one or both of the following techniques. First, it may simply repeat the required graphical element instead of referring to it. Second, it may refer to the graphical element by name. Both approaches have disadvantages. Repeating model elements causes unnecessary redundancies, which are especially undesirable for graphical modeling because model size is always a

great concern. In contrast to that, referring to model elements by name is not as depictive as a graphical reference. Therefore, the flashWeb approach relies on graphical references (edges) between different models thereby providing a superior overview of the designed system and a faster and more intuitive way for defining references. Again, the overall benefit for the Web application development process is saving time.

Besides a set of appropriate model elements, flashWeb models offer **extension facilites**, which ensure that the complete functionality of the Web application is captured and that a fully functional implementation can be generated. Web applications can have arbitrary complexity. Of course, graphical modeling can cover only a certain part of this complexity, because it usually does not have the expressive power of a programming language. Consider a simple algorithm of the Book Portal example that computes the ten most popular books of the portal based on user ratings. Such an algorithm cannot be expressed with flashWeb models or with a model of any other Web engineering method because that would require the expressive power of a programming language. Web engineering methods that are capable of code generation usually ignore this problem completely and declare such components as necessary customization work after the generation process, i.e., the required algorithm is implemented in a suitable programming language of the target runtime environment and merged into the generated code. However, this approach has a huge disadvantage. The created models cannot be used for further code generation without overwriting the altered code. There are partial solutions to this problem, which are detailed in Section 3.2.6.

The flashWeb method pursues an approach that eliminates this problem and provides an optimal integration of custom application logic into its models. To this end, it uses custom model elements that define the corresponding component as precisely as possible and integrates it into the model. In case of the simple algorithm that was mentioned before, flashWeb's Operation Model offers a so-called *custom operation* that can be used to specify the name (e.g. *getPopularBooks*), the result type (e.g. a list of *Book* objects), and the implementing code of the operation. This specialized model element integrates the custom functionality into flashWeb's Operation Model smoothly and ensures that the corresponding custom code is automatically merged into the implementation during the code generation process. The overall advantage of such an extension mechanism for the Web application development process is substantial. The developer may model standard functionality with standard model elements, he may integrate the signatures of custom parts into the models and add an implementation stub or even implement the custom functionality right ahead. In any case, the models can be used repeatedly to generate a functional implementation that does not have to be manually extended at some point after the generation process.

As already indicated, the flashWeb Web engineering method supports **code generation**. However, achieving the generation of a fully functional implementation is not a trivial task. As a matter of fact, several aspects that have been illustrated previously in this chapter are important prerequisites for effective code generation. First, the models have to possess sufficient expressive power to capture a potentially maximal part of the requirements. Second, different models have to be connected in a formal way in order to enable the generation of a cohesive implementation. Finally, models have to provide extension facilities that allow for custom business logic components.

For example, it is not sufficient to handle business logic operations at an abstract level. An operation *purchaseBook()* may be sufficient to capture the fact that the Web application allows its users to buy books, however it is not even nearly sufficient for the generation of a functional

implementation. Ordering products is a far more complex activity and it usually involves selecting desired products, providing payment information, etc. It cannot be implemented with a single operation. As already mentioned, the second common problem is that connections between different models are not precise enough for code generation purposes. For example, if the *addToShoppingCart()* operation, which may be part of the book purchase process, should be made available to the Web application's user, then it is not sufficient just to specify an *"Add to Shopping Cart"* button in the specification of the user interface. The operation has to be associated to the corresponding button precisely. The flashWeb method provides arbitrary flexibility for the developer to define business logic operations and to graphically connect them to user interface elements, thereby ensuring a cohesive specification. As mentioned before, it also provides extension facilities for custom business logic components, which is also an aspect that is often ignored by other Web engineering methods. Of course, such details are only relevant if a method supports code generation in the first place.

A final characteristic feature of the flashWeb method is relying on **object-oriented principles** throughout the entire development process. The flashWeb method uses object-oriented concepts in all of its models. To this end, the Content Model is specified using the UML and all other models employ a UML-near notation. The developer may define and use well known object-oriented concepts like types, classes, and objects in all flashWeb models.

4.2 Overview

The flashWeb Web engineering method provides graphical models that are primarily suited to be used in the design phase of a Web application development process. The focus of the method is on design models because they offer the appropriate level of abstraction that facilitates code generation and therefore can be used for rapid Web application development. Some of the Web engineering methods introduced in Section 3.4 also provide a model for requirements specification. Although these models can be very useful to capture requirements, they are too high-level to be mapped precisely to design models and therefore they cannot be used directly for code generation. Of course, many techniques for requirements analysis and many different models for requirements specification can be used in conjunction with the flashWeb method. However, corresponding results have to be manually transformed into flashWeb design models. An appropriate graphical model for requirements modeling that can be easily mapped to flashWeb's models is the User Interaction Diagram [VSDS00] introduced in Section 3.4.2.1.

The flashWeb method provides four models to describe a Web application. The **Content Model** specifies all real-world objects and their relationships that are relevant for the designed Web application. To this end, this model employs classes, attributes, and associations that are well known from object-oriented modeling. The **Operation Model** specifies content management operations that provide full read and write access to objects and object relationships, i.e., to the content of the Web application. Additionally, it provides composite and custom operations that allow to define arbitrary business logic. Elements of the **Composition/Navigation Model** define the composition of the Web application's user interface and specify theWeb application's navigation structure. Components of the user interface access operations in order to present or manipulate content and to execute business logic. Finally, the **Presentation Model** defines the visual appearance of the user interface. Unlike the first three flashWeb models, it

does not utilize a graphical notation. It consists of a set of style definitions that can be assigned to elements of the Composition/Navigation Model in a flexible way.

As mentioned in Section 4.1, a unique feature of flashWeb are graphical connections between different models. A graphical edge may associate a certain element of a model to an element of another model in order to express a certain semantic connection. Figure 4.1 depicts a simple example of flashWeb's graphical models and model connections. Note that the example uses the Book Portal scenario, which was introduced in Section 2.2.3.

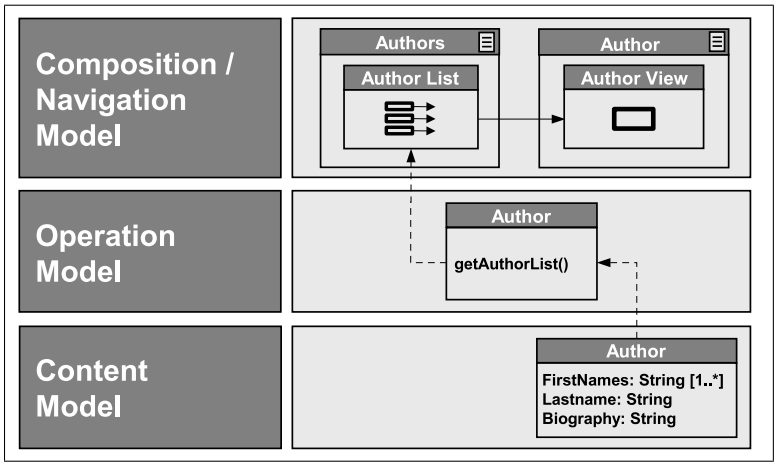

Figure 4.1: Overview of flashWeb Models

In this example, an author is modeled with an `Author` class of the Content Model. The Operation Model defines the *getAuthorList* operation for the `Author` class that returns a list of all author objects. Finally, the Composition/Navigation Model defines the `Authors` and `Author` pages, which provide information about authors and a single author respectively.

Connections between the Content Model and the Operation Model (dashed arrow) indicate which operations access which content objects. For example, in Figure 4.1, the directed arrow between the `Author` class of the Content Model and the *getAuthorList* operation of the Operation Model visualizes that the operation accesses objects of the `Author` type. Connections between the Operation Model and the Composition/Navigation Model show, which Web page components access which operations.

The `Authors` page of the Composition/Navigation Model contains an `ObjectIndex` element with the name "Author List" that lists all authors and allows the selection of a single author from this list. This model element is connected to the *getAuthorList* operation indicating that it uses data originating from this operation. The `Author` page contains an `ObjectView` element with the name "Author View" that shows details about the selected author. If the user of the Web application selects an author on the `Authors` page, then he is directed to the `Author` page.

Note, the example in Figure 4.1 is kept simple and omits some modeling details that are yet to be introduced. The intention of this example is to give an idea of the nature of flashWeb models and basic model interactions. All flashWeb models are explained thoroughly in the next section. Corresponding subsections also provide detailed examples using the Book Portal scenario.

4.3 Content Model

The **Content Model** of flashWeb defines the information domain of the developed Web application. It specifies all real-world entities and relationships between them that are relevant for the Web application. This model builds the basis for content-intensive Web applications by defining the application's content structure, which may be referenced by other models. The Content Model utilizes the syntax of a UML class diagram. Figure 4.2 shows the meta model of flashWeb's Content Model.

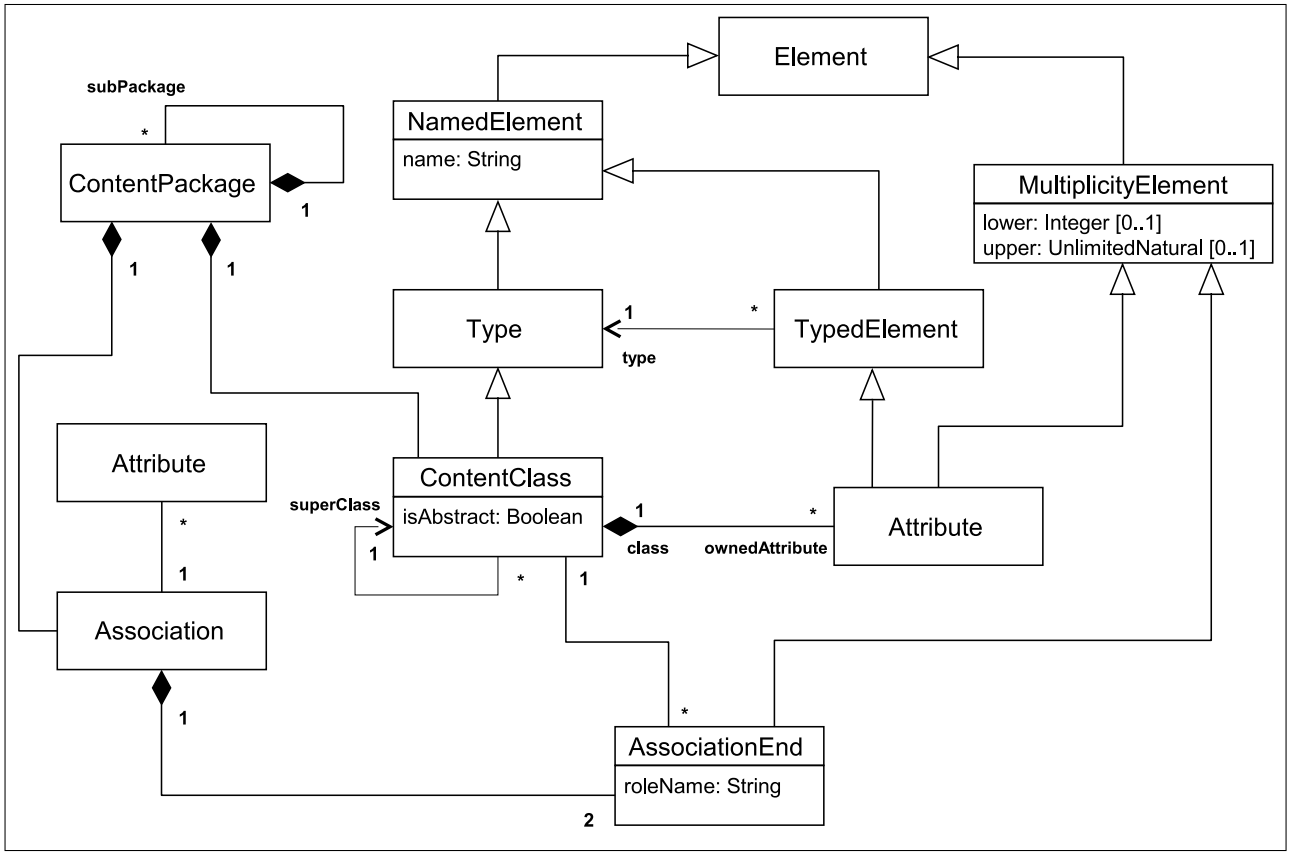

Figure 4.2: Meta Model of the Content Model

Main elements of the Content Model are the `ContentPackage`, the `ContentClass`, and `Association` components, which will be introduced thoroughly using the Book Portal example application from Section 2.2.3.

The UML class diagram is well suited to model the Web application's content. Not surprisingly, it is used by several other Web engineering methods for this task, e.g., the OOHDM or the UWE method. Subsequently, a short description of the main characteristics of the Content Model are introduced and slight differences to other methods are described. Figure 4.3 depicts the Content Model of the Book Portal example application from Section 2.2.3.

4.3.1 Content Packages

The `ContentPackage` element of the Content Model provides a means to separate self-contained subsections of the Web application. An arbitrary number of classes can be put into a package

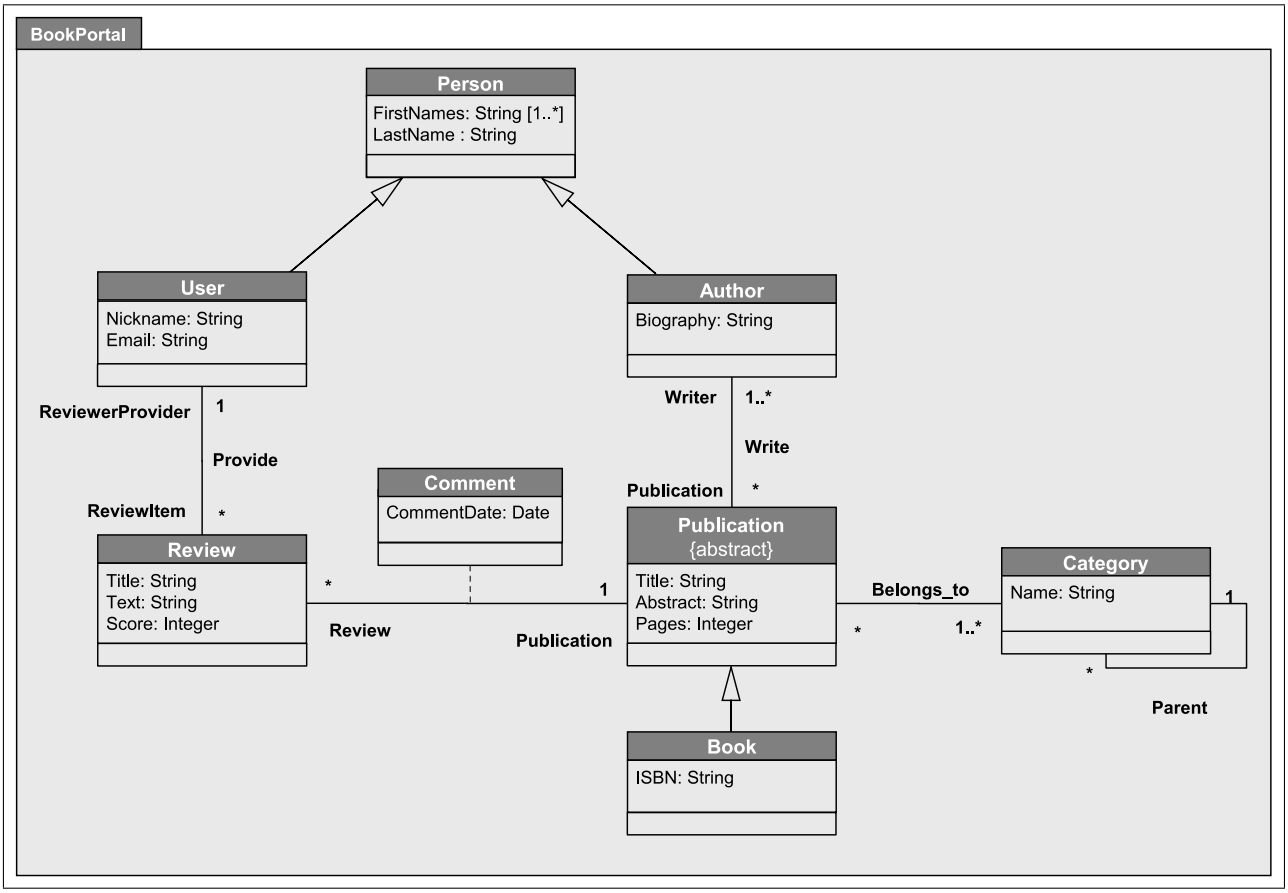

Figure 4.3: Content Model of the Book Portal with flashWeb

and the system may consist of an arbitrary number of packages. The example in Figure 4.3 includes one package with the name "BookPortal". Given the rather small size of the Book Portal example, a single package is sufficient to hold all classes of the model. Note that most Web engineering methods do not employ the notion of a package in their content models to identify sub-systems of the modeled Web application. However, this model element is very useful if the model is getting large.

4.3.2 Content Classes

A single entity of the real world is modeled with the `ContentClass` element and a set of attributes. For example, authors of the Book Portal application are captured with the `Author` class shown in Figure 4.3.

Note that the graphical notation of a content class is composed of three parts. The header section displays the name of the class. The attributes section contains all attributes that characterize the given class. Finally, the operations section usually contains a set of operations that specify the behavior of the modeled object. The flashWeb method employs a separate model for defining operations which is introduced in Section 4.4. Consequently, the operations section of content model classes are left empty.

A content class may hold an arbitrary number of attributes. Each attribute is characterized by its type and its multiplicity. The type of an attribute may be chosen from a set of primitive data types like *Integer*, *String*, or *Boolean*. Additionally, the type may be an enumeration of

these basic data types. Finally, each class of the model defines a new type that may be used as attribute type in any other class of the model. The multiplicity of an attribute is defined through a min-max notation. Both values can be arbitrary positive integers or * to denote infinity. For example, the Biography attribute of the `Author` class is of type *String* and has the multiplicity [1..1]. This multiplicity means that this attribute may have exactly one value. Note that this is the standard multiplicity for attributes, thus it is usually omitted in the model.

The Content Model supports specialization/generalization as known from object-oriented modeling. As depicted in Figure 4.3, the `Author` class is a specialization of the `Person` class. Consequently, an author possesses all attributes of a person, e.g., the *FirstNames* attribute, which is of the *String* type and has the multiplicity [1..*]. Accordingly, any person may have one or more first names.

Note that flashWeb models are strictly typed. To this end, data types and multiplicities are used in all models to characterize attributes, variables, operation parameters, etc.

Finally, a class may be assigned the *abstract* property to indicate that there will be no instances of that particular class in the implemented system. Abstract classes are used in inheritance hierarchies to capture common attributes of their sub-classes. In Figure 4.3, the `Publication` class is marked abstract, because the Book Portal application does not have general publication items. Instead, it manages books, which are specialized publications. Note that the *abstract* property does not have any further relevance for the Content Model. However, it will be important for the Operation Model (see Section 4.4), because abstract classes possess a different set of operations than regular classes.

4.3.3 Associations

The Content Model allows the specification of relationships between modeled objects. To this end, any two classes may be connected by an `Association` as known from UML. An association is characterized by a name, the multiplicities of participating classes, and role names that describe the role of an object in the corresponding relationship. Consider the `Write` association between the `Author` and `Publication` classes in Figure 4.3. The role of an author in this relationship is to be the "Writer" and the role of a publication is to be simply the "Publication". Of course, an author may have an arbitrary number of publications, thus the multiplicity at the publication-end is *, which is the short form for [0..*]. In contrast to that, a publication must have at least one author thus the multiplicity at the author-end is [1..*].

Relationships between objects may be characterized by attributes. This is modeled by an association class, which is connected to the corresponding association with a dashed line. Analog to a plain content model class, an association class may define an arbitrary number of attributes in the attribute section of the class. The name of the association is displayed in the header section. In Figure 4.3, the `Comment` association between the `Review` and `Publication` classes is specified by an association class providing the *CommentDate* attribute.

Besides simple associations, the Content Model allows two further association types that specify asymmetric relationships, i.e., one partner of the association plays a more dominant role than the other. Figure 4.4 depicts a possible extension of the Book Portal's content model providing corresponding examples.

The first of these association types is the *aggregation*, which specifies a part-of relationship. The current example shows the `PromotionFlyer` class, which is connected to the `Book` class. Each book may contain one to three promotion flyers (consider the multiplicity [1..3]), which

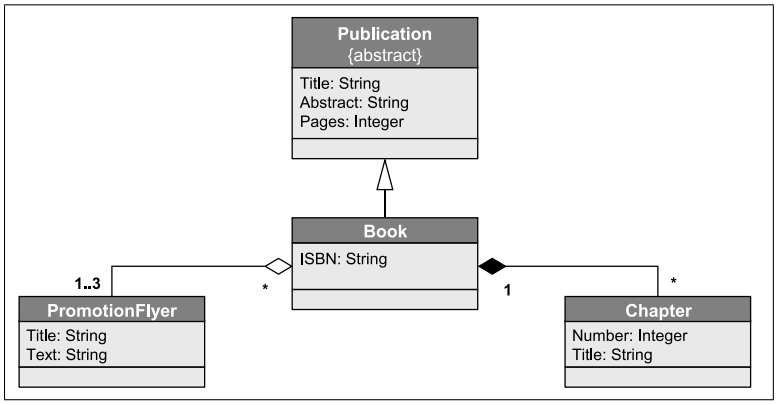

Figure 4.4: Aggregation and Composition Examples

include some advertisement, e.g., about reading glasses. Note that the same flyers can be also included in other books, thus a flyer is not an exclusive part of a book. This fact is modeled by the multiplicity * at the book-end of this association. In contrast to an aggregation, the *composition* specifies an exclusive part-of relationship. Figure 4.4 shows the *Chapter* class, which models chapters of a book. A book may have an arbitrary number of chapters indicated by the * at the chapter-end of the association. However, a specific chapter can never be part of another book, thus the multiplicity at the book-end is 1. Note that aggregations and compositions may be also characterized through role names. However, in this example, role names are omitted as they would match the corresponding class names.

Note that similarly to attributes, associations are also inherited from a super-class to all of its sub-classes. For example, the `Comment` association between the `Review` and `Publication` classes is inherited to the `Book` class, thus it also exists between the `Review` class and the `Book` class. This will be relevant for the Operation Model (see Section 4.4), because it provides for classes that participate in associations a set of content management operations that manage relationships between objects.

4.3.4 Notation Variations

Any graphical modeling solution for Web application development has the disadvantage that with the growing size of the models it gets increasingly difficult for the developer to keep a proper overview of the developed application. There are two major goals that have to be achieved by a modeling solution to overcome this dilemma. First, the model has to be able to abstract from details, thus the developer may keep track of the overall application structure and of dependencies between different application parts. Second, it must be possible to focus on parts of the model that are currently under development.

To this end, each flashWeb model provides alternative graphical notations that hide details, which may be irrelevant at a certain point in time. The ultimate goal of these simplified notations is to allow developers to focus on those parts of the model that are of interest to them. Of course, this flexibility can only be achieved with an appropriate CAWE tool (see Section 5.2) that supports these alternative notations.

The Content Model provides two simplifications, which are demonstrated in Figure 4.5.

The first solution of the Content Model to abstract from details is to represent *packages* and *classes* with a minimized graphical notation, which hides all sub-elements of the given model

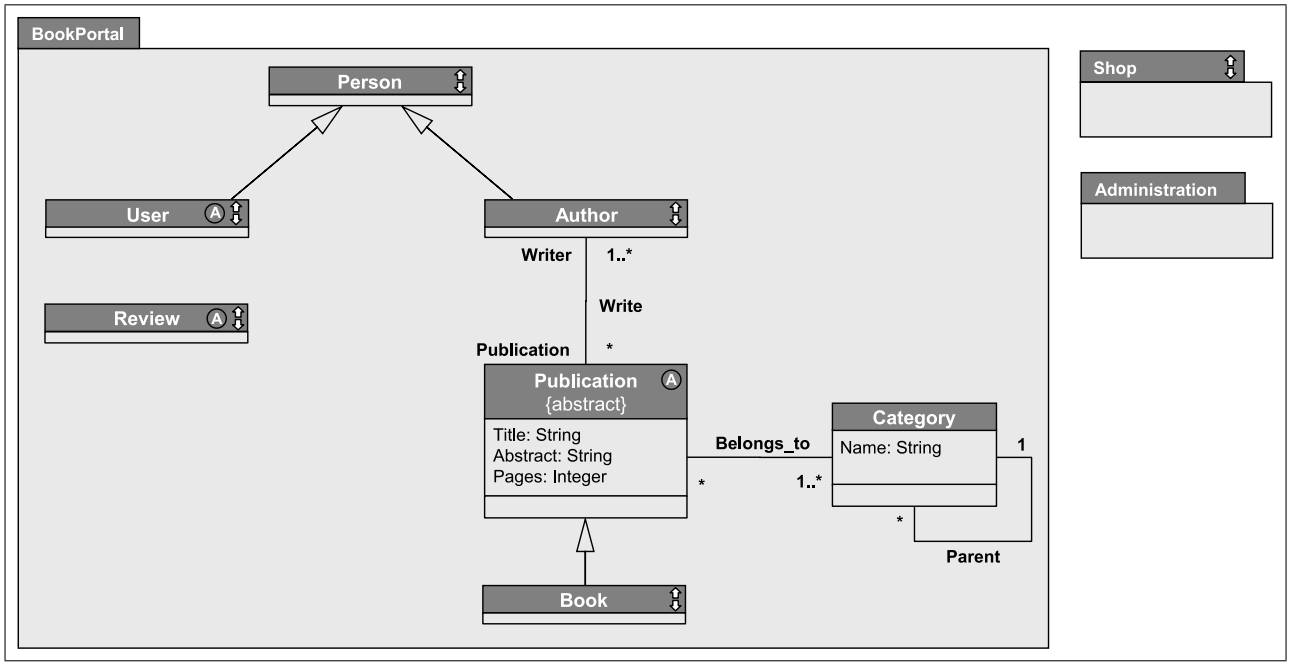

Figure 4.5: Notation Variations of the Content Model

element. In case of a package element, this means that all classes that are inside of the package are hidden. Observe the minimized `Shop` package on the right-hand side of Figure 4.5. This package may provide content definitions for an online shop component of the Book Portal. A minimized package element contains a symbol (two arrows directed into opposite directions) in the element's header section which indicates that the corresponding package can be maximized. Note that the `Administration` package, which does not contain any sub-elements yet, is maximized, thus it does not show the corresponding symbol in the header section.

Similarly to packages, a class element can be represented with a simplified notation as well. A minimized class element does not show the attributes of the class or the operations section. Instead, a small, empty portion of the attributes section is displayed. Analog to the package element, the header section of a minimized class element contains the appropriate *arrows* symbol. Examples of the minimized representation of a class element can be observed for the `Person` and `Author` classes in Figure 4.5.

The second approach of the Content Model for a simplified presentation is to hide associations. In Figure 4.5, most associations are hidden. An example is the `Provide` association between the `User` and `Review` classes. Note that if there exists at least one hidden association in which a class is participating, a corresponding symbol (a circle with the letter A) is shown in the header section of the class. This can be observed at the `Review` class, for which no associations are visible, and also at the `Publication` class because of the absence of the `Comment` association. In contrast to that, the `Category` class has no *A*-symbol because all relevant associations are visible.

Note that these simple mechanisms allow a fine-grained control of which model elements are displayed and how detailed the provided information is. A typical usage scenario of these simplified representations in a development setting may hide all packages of a Web application except for the one that is under development. The package may show minimized versions of all classes and hide all associations except for a small subset which is in the developer's focus.

4.3.5 Summary

As Web applications are usually content-intensive systems, many Web engineering solutions originate from the data/content management research area. Accordingly, content modeling as an essential part of any Web engineering method is usually a well-supported activity. Besides minor deficiencies of some other methods, e.g., WebML's inability to model relationship attributes (see Section 3.4.4.1), most methods employ content models with similar expressive power. However, they do not provide alternative simplified notations to ensure the usability of the model in a development environment.

4.4 Operation Model

The **Operation Model** of flashWeb defines a set of standard content management operations that provide full read and write access to the Web application's content. Additionally, this model supports the definition of arbitrary business logic. To this end, the model provides two modeling constructs. First, composite operations (see Section 4.4.2.2), which may compose standard content management operations but also other composite operations into more complex units of business logic. Second, custom operations (see Section 4.4.2.7), which integrate custom application code into the model.

The Operation Model serves as an intermediary between the Composition/Navigation Model (see Section 4.5) and the Content Model. Elements of the Composition/Navigation Model define the Web application's user interface and may be associated to arbitrary operations of the Operation Model. Ultimately, the Operation Model provides access to the Web application's content as well as to further business logic and allows to integrate them into the Web application's user interface in a flexible manner. Note that component reuse is an important feature of the Operation Model. To this end, each operation of the model is defined only once and can be used by an arbitrary number of Composition/Navigation Model elements. Figure 4.6 depicts the meta model of flashWeb's Operation Model.

The syntax of the Operation Model is simple. As a matter of fact, it corresponds closely to the syntax of the Content Model. Main constructs of the model are the `OperationPackage` and the `OperationClass` elements, which will be introduced subsequently. Figure 4.7 depicts a representative part of the Book Portal's Operation Model.

4.4.1 Operation Packages

Similarly to the Content Model, the Operation Model provides the `OperationPackage` element. Actually, for each package of a content model, there usually exists a corresponding package of the operation model bearing the same name. In most cases, this makes sense and helps the developer to keep the overview. Note however, that an operation model may specify additional packages that do not correspond to any content model constructs. An operation package may contain an arbitrary number of operation classes, which provide different content management operations (see subsequent sections). Ultimately, an operation package bundles a part of the Web application's business logic. Figure 4.7 depicts the "BookPortal" operation package and a set of exemplary operation classes.

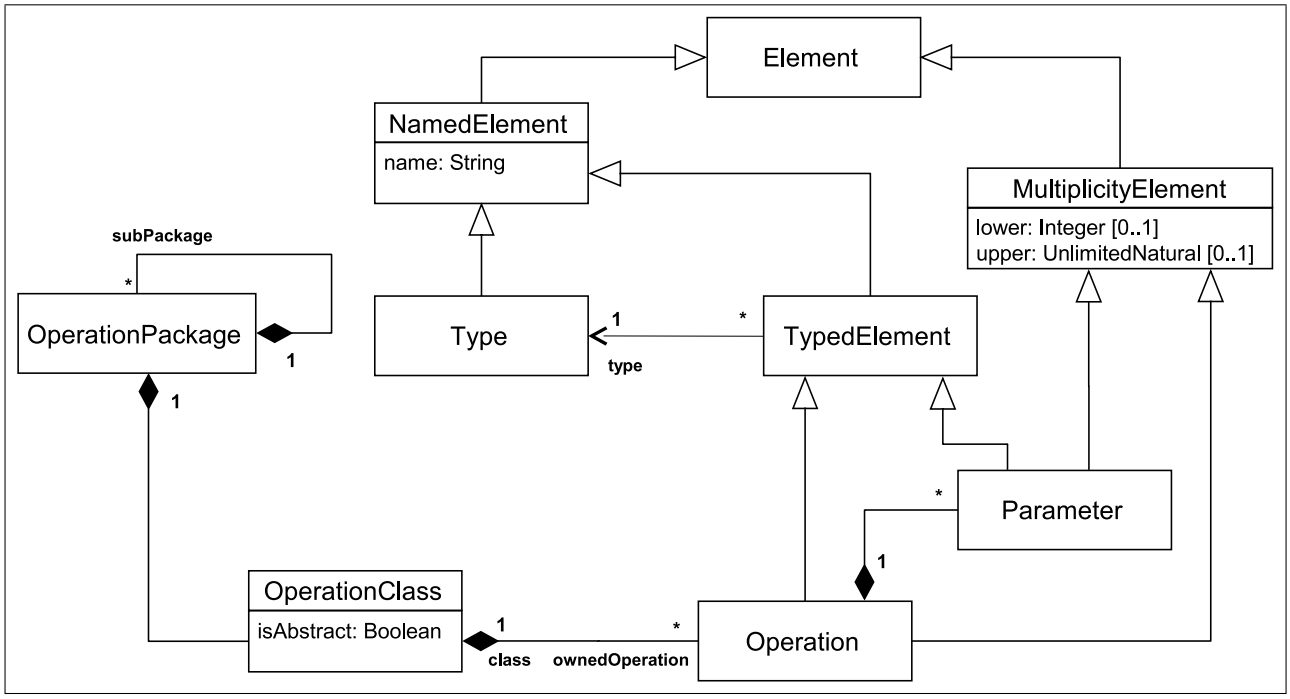

Figure 4.6: Meta Model of the Operation Model

4.4.2 Operation Classes

The `OperationClass` is the central model element of the Operation Model. The major goal of an operation class is to contain a set of operations that manage objects of a certain type. These *standard* operations provide full control over a specific object, e.g., they allow to create, to modifiy or to delete a book in the Book Portal application. Furthermore, standard operations manage relationships between objects, e.g., they allow to associate a book to an author. Additionally, an operation class may specify *composite* operations, which may combine other operations into a larger unit of application logic. Finally, *custom* operations allow the developer to integrate custom functionality, e.g., a specific algorithm into the Web application's application logic. Note that all operation types are explained thoroughly in subsequent sections.

The graphical notation of an operation class is composed of three parts. The header section displays the class name. The attributes section is of course empty, as attributes of the class are defined in the content model. The goal of an operation class is to specify operations, which are placed into the operations section of the class.

An operation model provides for each content class a corresponding operation class. Observe, for example, the `Author` operation class in Figure 4.7, which corresponds to the `Author` content class in Figure 4.3. Accordingly, this operation class provides standard content management operations for an author object. The signature of an operation is simple. It is composed of the operation name, the parameter list, and the data type of the operation's return value. Observe the *createAuthor* operation, which requires three parameters and returns an object of the `Author` type.

Note that abstract content model classes have operation model counterparts that provide only a few operations. This is due to the fact that abstract classes have no instances, thus most content management operations do not make sense. An example in Figure 4.7 is the `Publication` operation class. Finally, also association classes of a content model have oper-

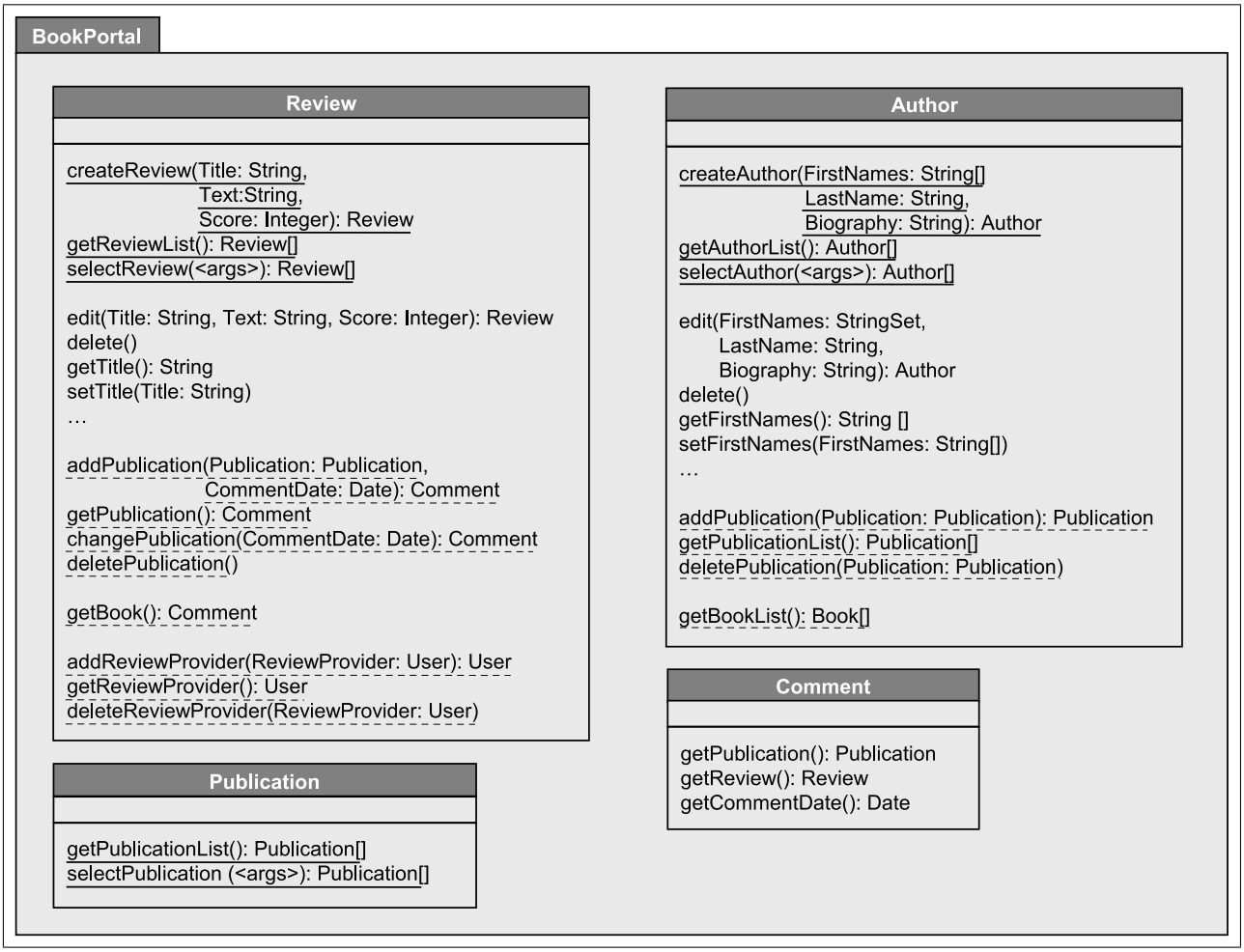

Figure 4.7: Partial Operation Model of the Book Portal

ation model counterparts. Their aim is to provide access to relationship partners and relationship attributes. Note that associations between classes are visualized in the Content Model and are not repeated in the Operation Model.

4.4.2.1 Standard Operations

Basic operations provide simple access to the Web application's content. They allow to retrieve, to create, to modify, or to delete content objects. Additionally, they also provide similar access to associations. For example, the right-hand side of Figure 4.7 depicts the `Author` operation class, which lists the most important standard operations for the `Author` type.

The Operation Model distinguishes three categories of standard operations. The first category includes **class-level operations**, which cannot be called directly on a single object because they concern several objects or the corresponding object does not exists yet. Class-level operation signatures are underlined, as you can observe in Figure 4.7. Listing 4.1 shows the generalized signatures of class-level operations.

The first class-level operation is the *constructor*, which creates a new object of a certain type. It requires parameter values for all attributes of the object and returns the newly created object. An example for a constructor is the *createAuthor* operation of the `Author` class in Figure 4.7.

```
create<type>(<params>): <type>
get<type>List(): <type>[]
select<type>(<params>): <type>[]
```

Listing 4.1: Class-level Operations

A frequently required functionality is to acquire all objects of a certain type. This is achieved by the *get<type>List()* class-level operation. For obvious reasons, this operation does not require any parameters. It returns a set of objects, which is indicated by the type name and the trailing square brackets at the end of the operation signature. An example for the *object-list* operation is the *getAuthorList* operation of the `Author` class in Figure 4.7. Its return type is *Author[]*, i.e., a set of `Author` objects.

Another common requirement is to deliver a set of objects that have certain characteristics, e.g., a list of authors with a specific last name. To this end, the *select<type>(<params>)* class-level operation allows to filter objects according to one or more attribute values. Accordingly, the operation requires one or more parameters that provide the necessary attribute values. For example, the *selectAuthor* operation of the `Author` class may be called with the parameter LastName='Smith' to acquire all authors that have the last name "Smith". This operation returns a set of `Author` objects, so the return type is specified with *Author[]*.

Note that the set of class-level operations is kept minimal. There are further possible methods of this category that can be added if necessary. For example, some developers like to work with Ids. Although, the flashWeb method abstracts from the notion of unique identifiers, because in object-oriented modeling the notion of an Id is not really necessary, the set of class-level operations could be extended to include operations that allow the usage of Ids. For example, the *get<type>(Id)* and *delete<type>(Id)* operations could be used to acquire or to delete an object of a specific type with the provided identifier. For the current example, the corresponding operations are *getAuthor(Id)* and *deleteAuthor(Id)*.

The second category of standard operations is called **instance-level operations**, because they provide functionality to manage a single object of a certain type. The object, on which an instance-level operation is called, will also be referred to as the operation's context. Signatures of instance-level operations are not highlighted in any way, as it can be observed in Figure 4.7. Listing 4.2 shows the generalized signatures of all instance-level operations.

```
get<attribute>(): <type>
set<attribute>(<param>)
edit(<params>): <type>
delete()
```

Listing 4.2: Instance-level Operations

There are two instance-level operations that access attribute values of an object. The *getter* operation, i.e., *get<attribute>* retrieves the value of a certain attribute. For example, the *getFirstNames* operation returns the first names of an author. As usual, the type of the operation's return value is indicated at the end of the operation signature, in this case *String[]*, i.e., a set of string values.

Accordingly, the *setter* operation, i.e., *set<attribute>(<param>)* sets the value for a certain attribute. Using the current example, the *setFirstNames* operation allows to alter an author's first names. Of course, this operation requires a set of *String* values to be passed on to the *FirstNames* parameter. Note that a *setter* operation has no return value.

An operation class provides *getter* and *setter* operations for all attributes of the corresponding content class. However, for the sake of simplicity, these methods are omitted for all further attributes of the `Author` class in Figure 4.7.

Of course, in some cases, it is desirable to modify all attributes of an object at once. The appropriate instance-level operation for this task is the *edit* operation. It requires values for all attributes of an object as parameters and returns the modified object as the result. An example of this operation for the `Author` class can be observed in Figure 4.7. It requires values for all attributes of an author and returns the modified `Author` object.

Finally, the last instance-level operation allows to delete an object. The *delete* operation requires no parameters and has no return value. Accordingly, the operation signature is rather simple, as it can be observed at the `Author` class in Figure 4.7.

The third and last group of standard operations belong to the category of **association-level operations**. These operations allow to manage associations between objects, i.e., to associate two objects and to change or to remove the association between them. As it can be observed in Figure 4.7, operations of this category are dashed underlined. Note that, for each association in the content model, a set of corresponding association-level operations are created for each participating class. However, the number and signatures of operations depend on several factors that will be explained subsequently. Operation names are constructed using the role names of associations and return types of the operations are the types of association partners or of corresponding association classes. Listing 4.3 shows the generalized signatures of all association-level operations.

```
add<partner_role>(<partner_obj>, <params>): <type>
get<partner_role>List(): <type>[]
change<partner_role>(<partner_obj>, <params>): <type>
delete<partner_role>(<partner_obj>)
```

Listing 4.3: Association-level Operations

The first association-level operation creates an association between two objects. The *add <partner_role>* operation requires at least one parameter, which is the partner object that should be associated to the current object. If the association between the two objects is described by attributes, i.e., the corresponding association is modeled with an association class in the content model, this operation requires further parameters, i.e., values for each describing attribute. Note that the return type of this operation may also vary. In case of a simple association, the currently associated partner object is returned. If the association has describing attributes then an instance of the association class is returned holding the associated partner and all describing attributes.

Consider the simple `Write` association between the `Author` and `Publication` classes in Figure 4.3. This association is represented by the *addPublication* method in the `Author` class, which assigns a publication to an author. Accordingly, the operation requires an object of the `Publication` type as parameter and it returns the same object as the operation result. Usu-

ally, the `Publication` class would include the corresponding *addAuthor* operation as well, however, the `Publication` class is abstract, thus it contains only class-level operations. Of course, the `Book` class contains the *addAuthor* operation because it is a sub-class of the abstract `Publication` class.

As further example, consider the `Comment` association between the `Publication` and the `Review` classes in Figure 4.3. This association is characterized by the *CommentDate* attribute. Accordingly, the `Review` class in Figure 4.7 contains the *addPublication* operation, which requires an object of the `Publication` type as association partner and a value for the *CommentDate* attribute as second parameter. Also, the return value of this method is not of the `Publication` type but of the `Comment` type. This type represents the association between the `Review` and `Publication` classes. An instance of the `Comment` class holds both objects that participate in the association and all describing attributes. Observe the `Comment` class in Figure 4.7. It provides the *getPublication*, *getReview*, and *getCommentDate* operations, which return the corresponding partners and the single describing attribute of this association.

The second association-level operation returns one or more association partners of an object. The *get<partner_role>List* operation requires no parameters and its return value is determined by two factors. First, analog to the *add* operation, the return type is different if the corresponding association has describing attributes. Second, the multiplicity of the return value corresponds to the multiplicity that was defined at the partner-end of the association in the content model.

As an example, consider the *getPublicationList* operation of the `Author` class in Figure 4.7. It returns all publications of an author. Of course, an author may have several publications, thus this operation returns a set of objects, which is indicated by the “List" keyword in the operation name and the *Publication[]* return type. Another *get* operation is the *getPublication* operation of the `Review` class, which returns the publication for that the review was written. A review can be associated to a single publication, thus the name of this operation does not include the “List” keyword and it returns a single object, which is of the `Comment` type.

Note that specialization/generalization hierarchies and abstract classes make things a little bit more complex regarding association-level operations. Abstract classes (like `Publication`) have no instances in an actual implementation. Accordingly, at any association-level operation of the Book Portal application, any parameter or return value, that is of the `Publication` type, will be actually an instance of a sub-type, e.g., a `Book` object. Also, *get* operations are handled in a special way in the case of specialization/generalization. If a class is associated to another class that has sub-classes, then additional *get* operations are added for each sub-type. An example is the *getBookList* operation of the `Author` class which complements the *getPublicationList* operation and allows to retrieve those publications of an author that are also books. Note that the Book Portal example is kept simple and that the `Publication` class has only a single sub-class. If it had more, lets say an additional `Article` sub-class, then the `Author` class would include an additional *getArticleList* operation.

The third association-level operation allows to change attribute values of an association between two objects. The *change<partner_role>* operation is only available in an operation class if the association is characterized by attributes, i.e., the corresponding association in the content model has an association class. The operation requires the partner object as first parameter if the multiplicity of the association at the partner-end allows to have multiple partners. Otherwise, the association partner is unambiguous. Values for all describing attributes as further parameters are always required. The return value of the operation is of the type that is defined

by the corresponding association class.

An example in Figure 4.7 is the *changePublication* operation of the `Review` class. It allows to change the *CommentDate* attribute of an association between the `Review` and `Publication` objects. This operation does not require the association partner (e.g., `Publication`) as first argument, because a review may have only a single associated publication. However, the operation still requires a parameter value for the *CommentDate* attribute. The return value of the operation is of the `Comment` type.

The final association-level operation allows to remove an association between two objects. Similarly to the *change* operation the *delete<partner_role>* operation requires the partner object as parameter if the object is allowed to have several partners. If only a single association partner is allowed, the operation does not require parameters. This operation does not have a return value.

Consider the *deletePublication* operation in the `Review` class in Figure 4.7. This operation removes the `Comment` association between a `Review` and a `Publication` object. As a review may be associated only to a single publication, the operation does not require any parameters.

Note that the introduced standard operations already facilitate the development of Web applications with powerful content management functionality. A Web application that provides access to the complete set of standard operations allows the Web application's user to create, manage, and delete objects and relationships between them according to the content management schema defined by the Web application architect. However, there are still some tasks that cannot be achieved with standard operations. A common requirement is to execute several standard operations at once as part of a single transaction. This problem is solved by composite operations, which are introduced in the following section.

4.4.2.2 Overview of Composite Operations

Composite operations combine two or more (standard) operations as one transaction. They play an important role in the design of the Web application's business logic because it is a common requirement to execute several operations in response to a single user action. The notion of composite operations is of course elementary for any programming language. However, most modeling languages, e.g., UML, do not provide a graphical notation for the composition of operations.

Consider the example of a Web application user, who wants to issue a book review at the Book Portal application. If there were only standard operations available, then the user would be forced to create the review first and then to associate the review to a book in a second step. The corresponding standard operations for this task are the *createReview* and the *addPublication* operations of the `Review` class in Figure 4.7. In contrast to that, a composite operation may combine these two standard operations to form a single unit. Such composite operations can be integrated into the Web application's user interface much easier.

There are four types of composite operations, which are depicted in Figure 4.8. The **sequential execution** groups two independent operations into a single unit and defines the execution order. The **simple chain** not only defines the execution order but also allows the second operation to receive the results of the first operation. The **join** defines the execution order of three operations and allows the third operation to receive the results of the first two operations. Finally, the **split** defines the execution order of three operations allowing the first operation to pass on results to the second and third operations.

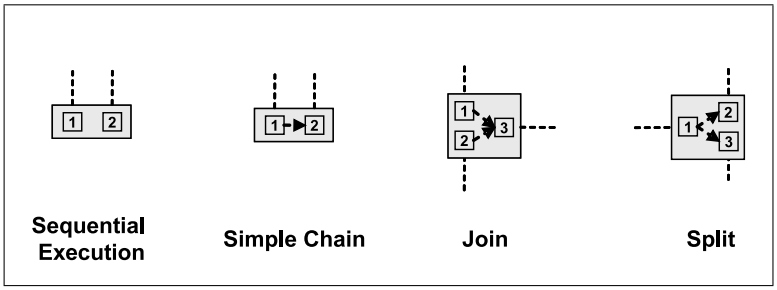

Figure 4.8: Composite Operations

All composite operations share some common characteristics, which are explained in this section. Figure 4.9 introduces a simple auxiliary notation and a formalism, which will be used subsequently to explain input and output mappings that have to be taken into account for composite operations.

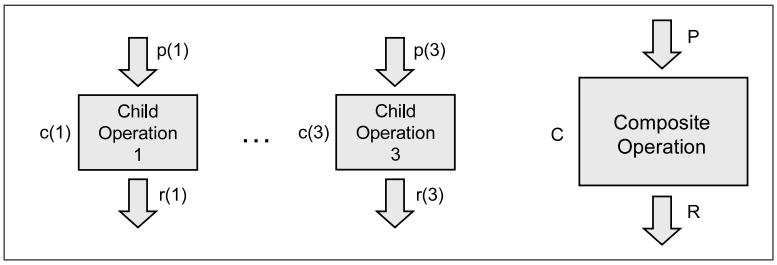

Figure 4.9: General Auxiliary Notation for Composite Operations

As shown in Figure 4.8 a *composite* operation merges the execution of two or three *child* operations and acts as a single unit of the Web application's business logic. A child operation may be any operation of the operation model, i.e., a standard operation, another composite operation or a custom operation (see Section 4.4.2.7).

Regarding the composition of child operations into a composite operation, there are three aspects that have to be considered. The set of *input parameters*, the operation's *return value*, and in case of instance-level or association-level operations the operation *context*. Let P be the set of input parameters of a composite operation and $p(n) \mid 1 \leq n \leq 3$ the set of input parameters of child operations. Also let R be the return value of the composite operation and $r(n) \mid 1 \leq n \leq 3$ the return values of child operations. The context of an operation is the object on which the operation is called. For example, the context of a *delete* operation may be a `Book` object. If this operation is executed, the book object is deleted from storage. The context is of course only relevant for instance-level and association-level operations (see Section 4.4.2.1). Let C be the context of the composite operation and $c(n) \mid 1 \leq n \leq 3$ the contexts of child operations. Note that the order of a composite operation's parameter list is defined by the order of child operations, i.e., first it includes parameters that are required by the first operation, then parameters that are required by the second operation, etc. If a child operation requires a context and the context is provided as parameter in the parameter list of the composite operation, then it is always placed in the list before any other parameter that is required by the same child operation. Note that this auxiliary notation and the introduced formalism are not part of the flashWeb Operation Model. They are merely used to provide a detailed explanation of input-output mappings between composite operations.

4.4.2.3 Sequential Execution

The **sequential execution** combines two operations into a composite operation unit that are sequentially executed but are not depending on each other, i.e., the result of the first operation is not used by the second operation. Figure 4.10 depicts both the auxiliary notation (left) and the flashWeb notation (right), which can be used in the Operation Model.

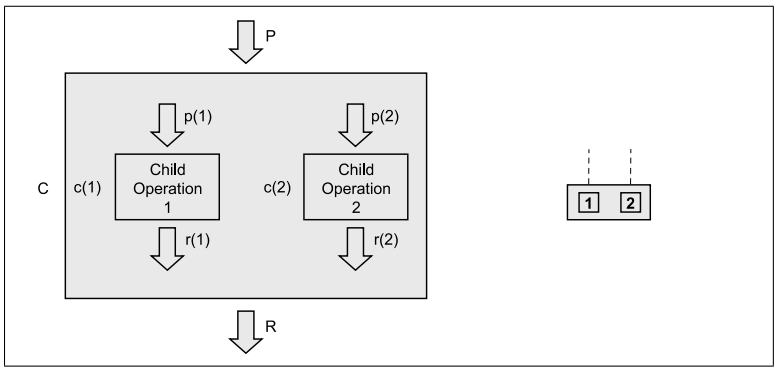

Figure 4.10: Notation of the Sequential Execution

Although the sequential execution of two child operations does not seem very complicated, there is a number of configurations that can occur regarding the mapping of input and output values. The auxiliary notation in Figure 4.10 shows all relevant input and output values for all participating operations. First, parameters of both child operations usually originate from the parameter list of the composite operation ($p(1) \subset P, p(2) \subset P$). Of course, one or both child operations may not require parameters at all (e.g., $p(1) = \emptyset$). Second, one or both child operations may require a context (e.g., instance-level operation). The context object for one or both child operations may originate from the parameter list of the composite operation (e.g., $c(1) \in P$) or may be identical to the context of the composite operation if it is an instance-level operation. Finally, the return value of the composite operation can be either the return value of one of the child operations ($R \in \{r(1),\ r(2)\}$) or it may have no return value at all ($R = \emptyset$).

Note that the flashWeb Operation Model employs a simpler graphical notation to specify a sequential execution. The so-called *operation connector* of this operation is depicted on the right-hand side of Figure 4.10. Its aim is to indicate which child operations are part of the composite operation, in which order they are executed and the return value of which child operation serves as return value of the composite operation. Note that in the operation model the operation connector is placed next to the signature of a composite operation and the numbered *ports* of the connector are connected to the signatures of child operations by directed dashed edges. The direction of these edges indicate whether an operation requires parameters or whether it has a return value. Figure 4.11 depicts a concrete operation model example of the sequential execution composite operation.

In this example, the *replaceReview* composite operation is specified, which deletes an existing review and creates a new one instead. Also this operation returns the newly created *Review* object. Note that for the sake of simplicity it is assumed that the review is not associated to any publications yet, thus the association does not have to be re-created as well. Also note that in Figure 4.11 all operations of the `Review` class that are irrelevant for the current example are omitted.

The first child element of the *replaceReview* composite operation is the *delete* operation of

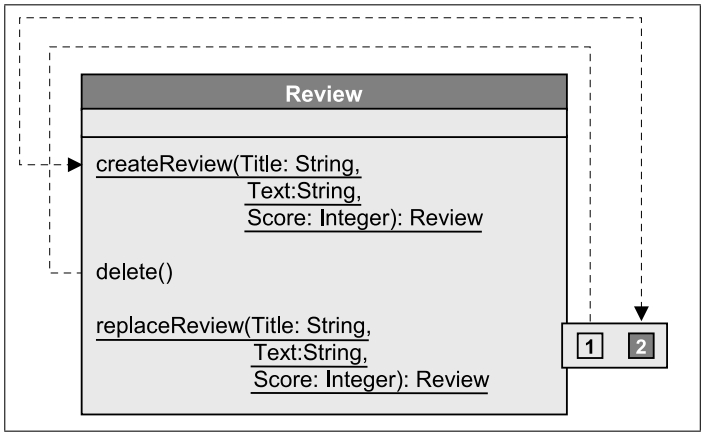

Figure 4.11: Sequential Execution Example

the `Review` class. This operation is at the instance-level, i.e., it requires a context, which is obviously a `Review` object. The composite operation just shares the context with the *delete* operation ($C = c(1)$). The *delete* operation does not require any parameters ($p(1) = \emptyset$) and does not return a value ($r(1) = \emptyset$).

Note that all of these facts are indicated by the graphical notation in Figure 4.11. The signature of the *delete* operation is connected by a dashed edge to the first port of the *replaceReview* operation's connector, indicating that it is the first child operation. Both operations are instance-level operations (not underlined) of the `Review` class, showing that they share the same context. The signature of the *delete* operation indicates that it does not require parameters or return a value. This is additionally indicated by the missing arrow heads of its connection to the operation connector.

The second child element of the *replaceReview* composite operation is the *createReview* operation of the `Review` class. It is a class-level operation, i.e., it does not require a context ($c(2) = \emptyset$). This child operation has input parameters, which are provided by the parameter list of the composite operation ($p(2) \in P$). As a matter of fact, these are the only values that are required to execute both child operations, so the operation signatures are identical ($P = p(2)$). The *replaceReview* operation returns the return value of the *createReview* operation ($R = r(2)$).

These facts are also clearly represented by the graphical notation in Figure 4.11. The *createReview* class-level operation (underlined) of the `Review` class is connected to the second port of the *replaceReview* operation's connector. The arrowheads at both ends of the connection indicate that this child operation requires parameters and returns a value. Of course, this can be observed at the operation signature as well. Finally, the return value of this child operation is used as the return value of the composite operation, which is indicated by the highlighted second port of the operation connector.

4.4.2.4 Simple Chain

The **simple chain** is similar to the sequential execution as it combines two sequentially executed child operations into a composite operation. However, the return value of the first child operation is required as input for the second child operation. Figure 4.12 depicts the auxiliary notation (left) and the operation connector (right).

The auxiliary notation in Figure 4.12 shows input and output values for the combined child

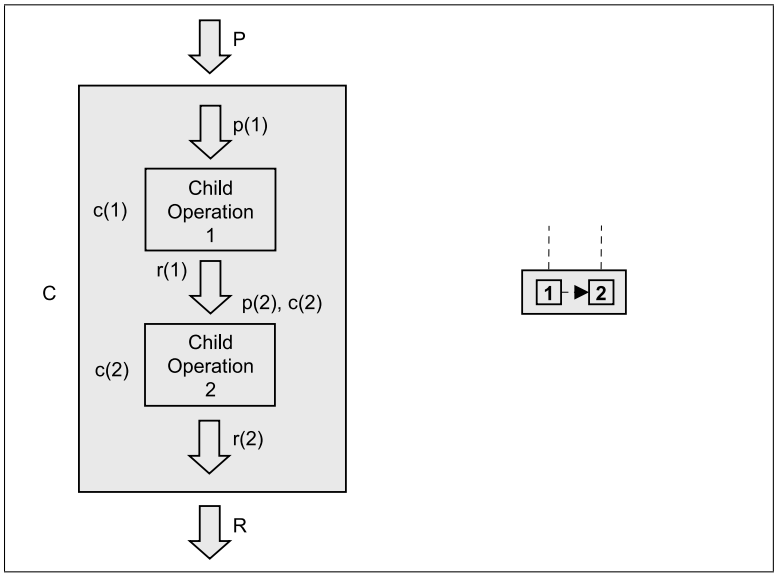

Figure 4.12: Notation of the Simple Chain

operations. If the first child operation requires parameters, then these have to come from the parameter list of the composite operation ($p(1) \subset P$). In contrast to that, the parameters for the second child operation may come from the composite operation's parameter list ($p(2) \cap P \neq \emptyset$) or from the return value of the first child operation ($r(1) \in p(2)$). Both child operations may require a context. Similarly to parameters, the context may come from the parameter list of the composite operation (e.g. $c(1) \in P$) or for the second operation from the return value of the first operation (e.g. $c(2) = r(1)$). The context may also be identical to the context of the composite operation (e.g. $c(1) = C$). The return value of the composite operation may be the return value of one of the child operations ($R \in \{r(1),\ r(2)\}$). Note that the operation connector of the simple chain composite operation on the right-hand side of Figure 4.12 is almost identical to the previously introduced connector of the sequential execution operation. The only difference is that the dependency between the two child operations is indicated by an arrow that points from the first to the second port of the operation connector. Figure 4.13 depicts an operation model example for the *simple chain* composite operation.

The composite operation of this example is named *createReviewForProvider*, and besides creating a new `Review` object, it also associates it to a `User` object. The newly created *Review* object is the operation's return value. Note that the creation of an object in the context of another object is a common content management pattern. Also note that Figure 4.13 shows only those operations of the `Review` class that are relevant for this example.

The first child element of the *createReviewForProvider* composite operation is the *createReview* operation, which is indicated by its connection to the first port of the operation connector. It is a class-level operation thus it does not require a context ($c(1) = \emptyset$). The *createReview* operation obviously requires values for all attributes of a `Review` object, which are provided by the parameter list of the composite operation ($p(1) \subset P$). The created `Review` object is the return value of the *createReview* operation and is used by the second operation as context. This fact is indicated by the solid line of the arrow between the two ports of the application connector. If the return value would be required for the second operation as a simple parameter, the arrow between the ports had a dashed line.

The second child element of this composite operation example is the *addReviewProvider* association-

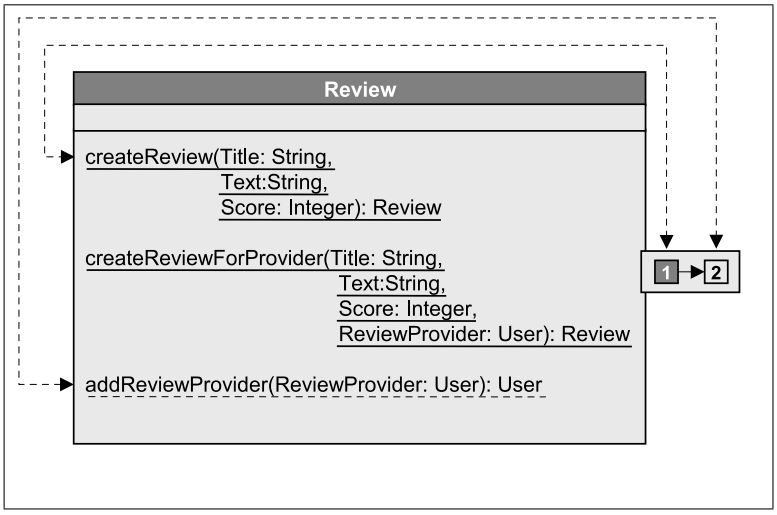

Figure 4.13: Simple Chain Example

level operation of the `Review` class indicated by its connection to the second port of the operation connector. This operation associates a user to a review and requires a `Review` object as context, which is the return value of the *createReview* operation ($c(2) = r(1)$). The *addReviewProvider* operation requires a single parameter, which is the `User` object that is to be associated to the newly created `Review` object. This parameter originates from the parameter list of the composite operation ($p(2) \subset P$). The return value of this operation, which is the associated `User` object is not required any further. In contrast to that, the return value of the first child operation serves as the return value of the composite operation ($R = r(1)$). This fact is indicated by the highlighted first port of the operation connector.

4.4.2.5 Join

The **join** is the first composite operation to combine three child operations. The idea is similar to the *simple chain* operation with a minor difference. Instead of one, there are two child operations, which deliver return values for a third child operation. Figure 4.14 depicts the auxiliary notation and the operation connector for this operation.

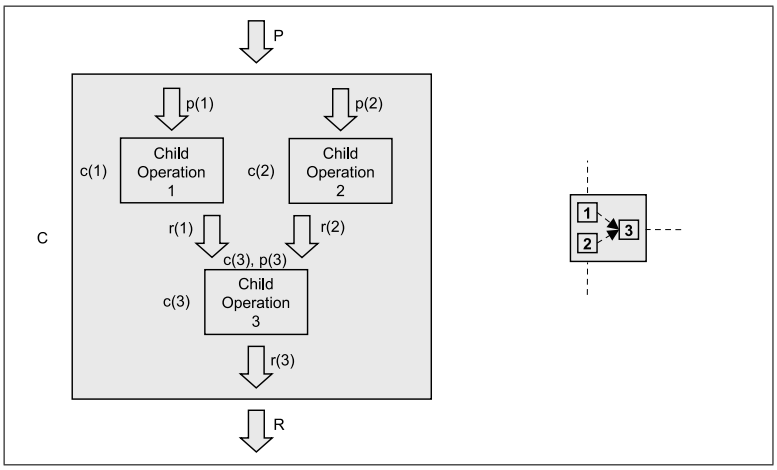

Figure 4.14: Notation of the Join

The auxiliary notation in Figure 4.14 depicts input and output value dependencies between three child operations. The composite operation may deliver parameters for the first and second child operations ($p(1) \subset P, p(2) \subset P$). Parameters for the third operation may come from the parameter list of the composite operation ($p(3) \cap P \neq \emptyset$) or from the return values of the first two operations ($r(1) \in p(3),\ r(2) \in p(3)$). Of course, one or more child operations may require a context, i.e., they may be instance-level or association-level operations. In that case, the context for the first two operations may have the same context as the composite operation ($c(1) = C, c(2) = C$) or originate from its parameter list ($c(1) \in P,\ c(2) \in P$). In case of the third operation, the context may originate additionally from the return value of one of the first two operations ($c(3) \in \{r(1),\ r(2)\}$). Finally, the return value of the composite operation may be the return value of any of the child operations ($R \in \{r(1),\ r(2),\ r(3)\}$).

The operation connector on the right-hand side of Figure 4.14 illustrates the dependencies between all child operations. Arrows point from the first two ports to the third port of the operation connector indicating that the return values of the first two operations are required as input for the third operation. Figure 4.15 depicts an operation model example for the join composite operation.

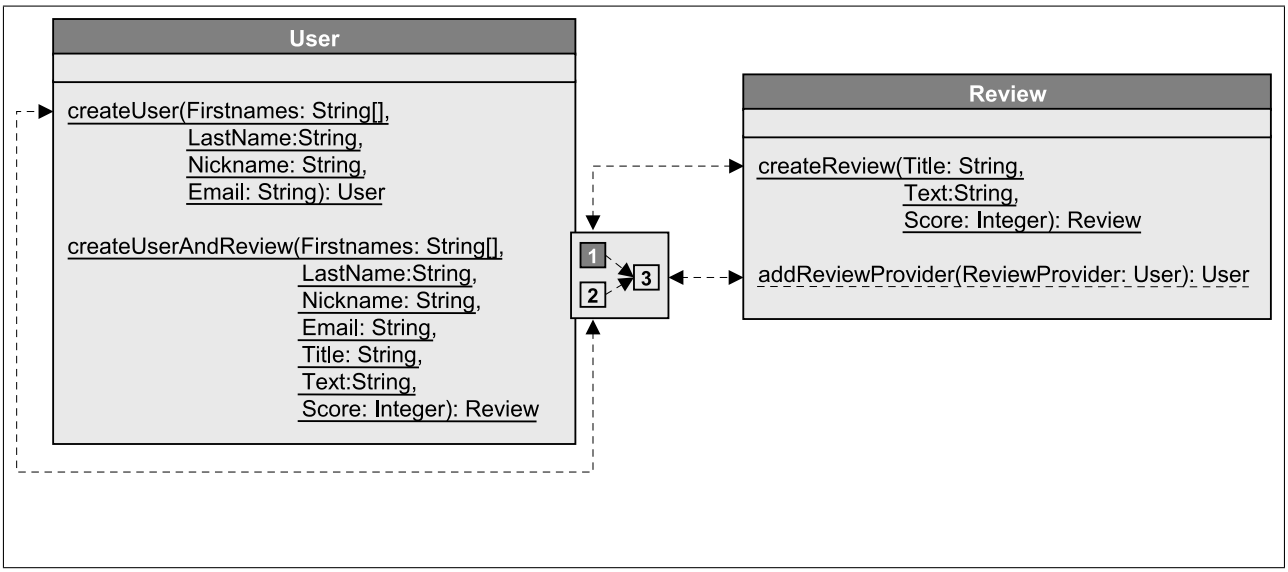

Figure 4.15: Join Example

The *createUserAndReview* composite operation of this example combines three child operations. First, the *createReview* class-level operation of the `Review` class creates a `Review` object. Second, the *createUser* class-level operation of the `User` class creates a `User` object. Finally, the `addReviewProvider` association-level operation of the `Review` class associates the newly created review to the also newly created user. Parameters that are required by the first two operations originate from the parameter list of the composite operation ($p(1) \cup p(2) = P$). The execution order of child operations is indicated by the connections between the composite operation's connector and the signatures of the corresponding child operations. Also, the arrows between the connector's ports indicate that the *addReviewProvider* operation receives input from the other two child operations. The solid arrow between ports 1 and 3 shows that the return value of the *createReview* operation is used as context ($r(1) = c(3)$), while the dashed arrow between ports 2 and 3 shows that the return value of the *createUser* operation serves as

parameter ($r(2) = p(3)$) for the *addReviewProvider* operation. Finally, the return value of the first child operation is the return value of the composite operation ($R = r(1)$).

4.4.2.6 Split

The last composite operation is the **split**. Similarly to the join operation, it combines three child operations. However, instead of *joining* the return values of two child operations, it *splits* the return value of a single child operation and provides it for two other operations. Figure 4.16 depicts the auxiliary notation and the operation connector for this operation.

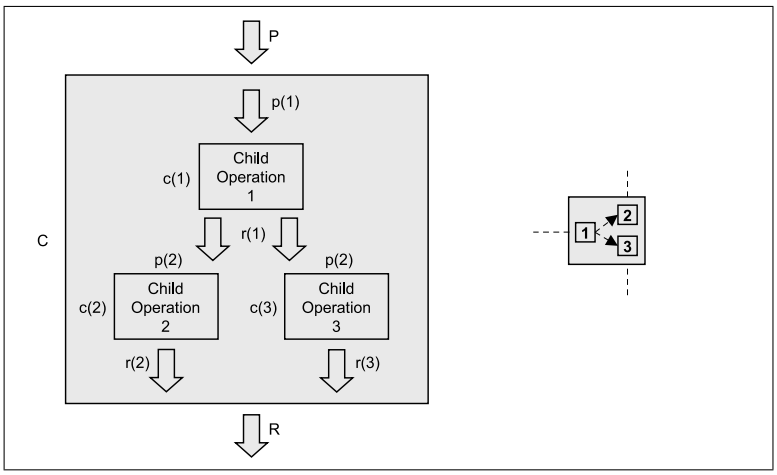

Figure 4.16: Notation of the Split

If the first child operation requires parameters, then these have to come from the composite operation's parameter list ($p(1) \subset P$). Parameters for the second and third operations may come from the parameter list of the composite operation ($p(2) \cap P \neq \emptyset$, $p(3) \cap P \neq \emptyset$) or from the return values of the first operations ($r(1) \in p(2)$, $r(1) \in p(3)$). If one or more child operations require a context, then it may be identical to the composite operation's context (e.g. $c(1) = C$) or the context may be a parameter of the composite operation (e.g. $c(1) \in P$). Additionally, the context of the second and third child operations may be the return value of the first child operation ($c(2) = r(1)$, $c(3) = r(1)$). As usual, the return value of the composite operation may be the return value of any of the child operations ($R \in \{r(1), r(2), r(3)\}$).

The operation connector on the right-hand side of Figure 4.16 illustrates the nature of the split operation. Arrows point from the first port to the second and third ports of the operation connector. This indicates that the return value of the first child operation is provided as input for the second and third child operations. Figure 4.17 depicts an example of the split composite operation.

The *addReviewForPublication* composite operation creates a review of a certain user for a certain book. To this end, it combines the *createReview* class-level operation, the *addReviewProvider* association-level operation and the *addPublication* association-level operation, which associates the review to a publication, e.g., a `Book` object. Parameters for the *createReview* operation are provided by the parameter list of the composite operation ($p(1) \subset P$). The arrows between the operation connector's ports indicate that the second and third operations receive input from the first operation. In this case, the input is the context for both operations ($c(2) = c(3) = r(1)$), which is the newly created `Review` object returned by the *createReview* operation. This fact is

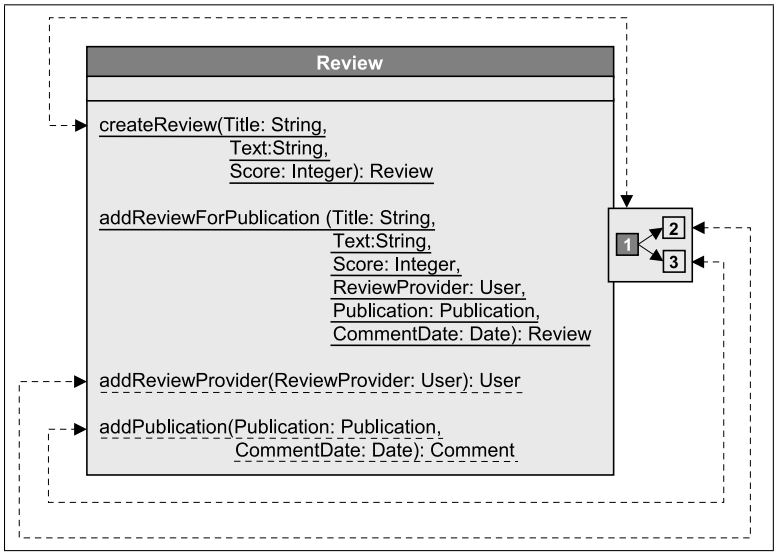

Figure 4.17: Split Example

indicated by the solid arrows between the connector's ports. Finally, the return value of the *createReview* operation is the return value of the composite operation ($R = r(1)$). This is indicated by the operation connector's first port, which is highlighted.

Of course, a composite operation may not only include standard operations but also other composite operations. Accordingly, a composite operation may recursively combine an arbitrary number of other operations and thus define a complex piece of application logic. Note that the functionality of the *join* and *split* composite operations may be simulated by two *simple chain* operations, respectively. However, as the corresponding examples demonstrate the *join* and *split* operations are suitable to implement frequently required application patterns, thus they are included into the flashWeb Operation Model for convenience reasons.

Note that the aim of composite operations is not to provide a graphical programming language. Although they are very useful for defining common content management patterns, the expressive power of composite operations, which only include standard operations and other composite operations, is limited. Important programming concepts like conditions and iteration are not supported. Thus, if the power of a programming language is required the Web application developer should utilize custom operations, which are introduced in the next section.

4.4.2.7 Custom Operation

The aim of the Operation Model is to define basic building blocks of the Web application's business logic in a graphical manner. Previous sections introduced a set of standard operations, which provide simple content management functionality and composite operations, which allow to combine them into larger blocks of application logic. However, even the simplest Web application usually includes functionality that cannot be expressed with standard operations. The Book Portal application (see Section 2.2.3) provides an appropriate example. The `Portal` page shows a guided tour of the ten most popular books of the portal, ranked by review scores of portal users. In order to get the list of most popular books, a simple algorithm has to be executed, which includes calculating the average review score for each book of the portal and

sorting the books by this average score.

To consider application logic that is beyond the scope of standard operations and their combination into composite operations, the Operation Model provides the **custom operation**. It is a container for arbitrary application code utilizing an appropriate programming language of the implementation environment. As mentioned before, the aim of the operation model is to provide *building blocks* of business logic and the model element with the finest granularity for this task is an *operation*. Beyond that, the Operation Model does not provide modeling help. However, as any other operation, a custom operation is described by its signature, which integrates the provided piece of application logic very well into a Web application's operation model.

Figure 4.18 shows the partial `Book` operation class. Additionally to some remaining standard operations like the *createBook* class-level operation, this class includes the *getTopTenRatedBooks* custom operation, which implements the functionality that was described previously.

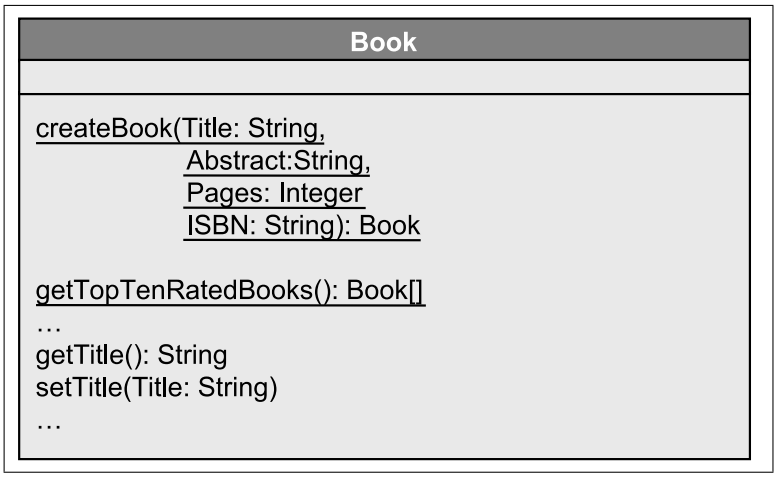

Figure 4.18: Custom Operation Example

The signature of a custom operation is the same as of any other operation of the Operation Model. It is composed of the operation name, the parameter list and the type of the return value. Note that the current custom operation example in Figure 4.18 does not require any parameters and it returns a set of `Book` objects.

Custom operations may be defined at all levels that have been described in Section 4.4.2.1. A custom operation may be a class-level operation defining functionality that is not restricted to dealing with a single instance, it may be an instance-level operation that manages a single instance or an association-level operation that handles relationships between instances. Note that the developer may specify the appropriate category arbitrarily and that the chosen category is indicated by the same graphical schema (underlining) as for standard operations. Additionally, the signature of a custom operation is displayed in italic to distinguish it from standard and composite operations.

Similarly to composite operations, a custom operation cannot be created automatically but has to be defined by the Web application developer. The definition includes two steps. First, the developer specifies the operation signature, i.e., the operation name, the parameter list, and the return type. The signature integrates the custom operation smoothly into the operation model of a Web application and ensures that it may serve as a child operation in composite operations or that it may be used by the Composition/Navigation Model (see Section 4.5) to

integrate custom application code into the user interface. Second, the developer provides the implementation using an appropriate programming language, i.e., the programming language of the target runtime environment for which the Web application is developed. Listing 4.4 shows a possible implementation of the *getTopTenRatedBooks* operation in Python.

```
1  books = getClass().getBookList()
2  book_ratings = {}
3
4  #calculate and store average ratings for each book
5  for book in books:
6      ratings = [review.getScore() for review
7                                    in book.getReviewList()]
8      avg_rating = sum(ratings) / len(ratings)
9      book_ratings[book] = avg_rating
10
11 #sort books by average rating in descending order
12 books.sort(lambda x, y: cmp(book_ratings[y],
13                              book_ratings[x]))
14
15 #return the top ten (or less)
16 return books[:10]
```

Listing 4.4: Example Custom Operation Code

Note that this implementation is pretty simple and self-explanatory. However, there are some details that have to be explained subsequently. Of course, it is desirable for the Web application developer to be able to access any operation of the operation model from any custom operation. To this end, the developer may use the special *getClass* operation that returns a desirable *class* object, which provides all class-level operations of a certain operation class. If the *getClass* operation is called without parameters, then it returns an object representing the class, in which the current operation is defined. As an example, observe the first line in Listing 4.4. The *getClass* operation provides access to the `Book` class and the *getBookList* class-level operation may be called, which returns a list of all books. Furthermore, the *getClass* operation may provide access to an arbitrary class of an operation model if it receives the qualified name of the class (including the names of all packages separated by dots) as parameter. Accordingly, the *getClass* operation may also provide access to the *Book* class if it is called with the "BookPortal.Book" parameter value.

The Web application developer may use instance-level and association-level operations as well. These operations require a context, i.e., an object on which they may be called. Listing 4.4 provides examples for both cases. In line 7, the *getReviewList* association-level operation is called on a `Book` object in order to acquire all reviews of a book. Finally, during the iteration over all reviews (lines 6 and 7), the *getScore* instance-level operation is called on all `Review` objects.

Of course, not only standard operations are allowed to be accessed from a custom operation, but also composite operations and other custom operations. Ultimately, the Web application developer has access to the complete application logic and may extend it arbitrarily. Thus the custom operation is an extension facility of the Operation Model, which allows to specify

application logic that cannot be expressed with standard or composite operations. The importance of such extension facilities is described in Section 4.1. Furthermore, the implications of extension facilities for the code generation process are discussed in Section 3.2.6.

4.4.3 Connections Between Operations and Content

Graphical connections between models are a key characteristic of the flashWeb development process. They express an important part of the Web application's logic and provide an enhanced overview thereof. Figure 4.19 depicts an UML class diagram that defines data access connections between the Operation Model and the Content Model.

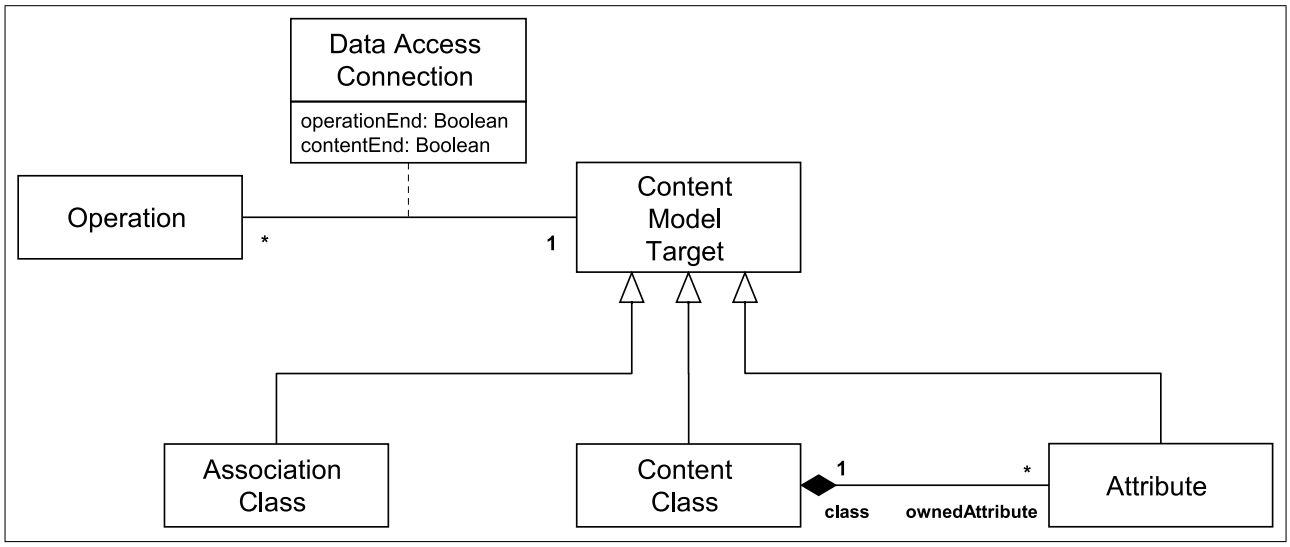

Figure 4.19: Meta Model of Connections between the Operation Model and the Content Model

A data access connection visualizes for each operation the manner of data access. It may directly connect an operation to a target in the Content Model, which may be a class, an attribute, or an association class. A data access connection may be unidirectional or bidirectional indicating whether an operation receives data from the content storage, writes data to the content storage, or both. Corresponding boolean attributes of the `Data Access Connection` association specify whether the connection is directed towards the operation end, the content end, or both. Figure 4.20 depicts an example including the graphical representation of data access connections of the `Review` operation class to corresponding content model elements.

For example, the *createReview* class-level operation creates a new `Review` object and is connected to the corresponding content model class, i.e., the `Review` class. This operation is concerned with an object as a whole, thus its scope is the class and accordingly the target of the data access arrow is the header of the `Review` content model class. Note that the arrow is bidirectional indicating that the operation modifies the Web application's content and also receives data from the content storage, i.e., the newly created `Review` object. Note that there are also further operations that have the scope of a whole object, e.g., the *edit* and *delete* operations. These are all connected to the header of the `Review` class.

The scope of an operation may also be a single attribute. This is the case for the so-called *getter* and *setter* operations. An example is the *getTitle* operation of the `Review` operation class. It is connected directly to the *Title* attribute of the `Review` content model class. Of course,

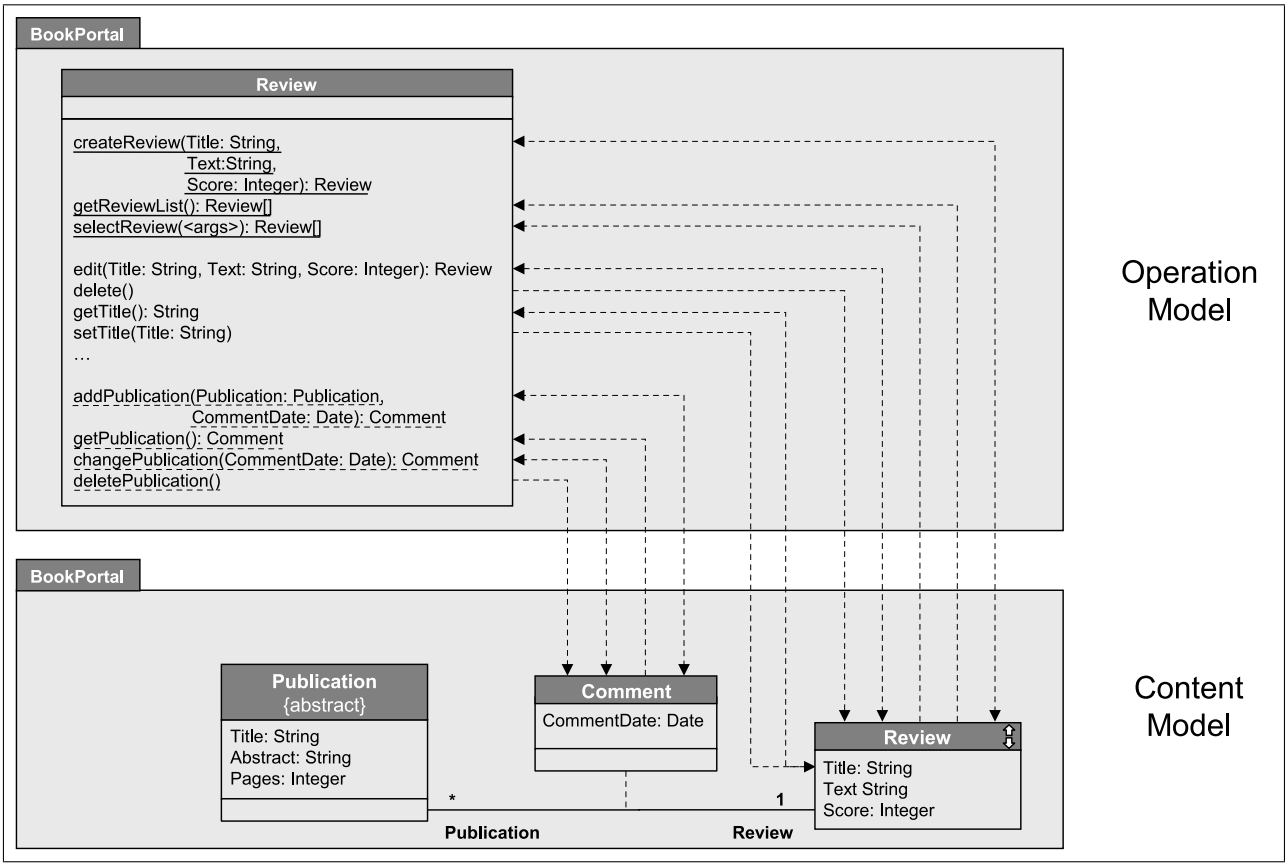

Figure 4.20: Operation Model - Content Model Connections Example

this operation only reads data thus the data access arrow is directed from the attribute to the operation signature. In contrast to that, the *setTitle* operation, which alters the value of the *Title* attribute of a `Review` object, has a data access arrow that is directed from the operation signature to the attribute.

Finally, operations that manage relationships between objects are connected to the corresponding association or association class. For example, in Figure 4.20 the *addPublication* operation is connected with a bidirectional data access arrow to the `Comment` association class indicating that the operation manages the *Comment* relationship between `Publication` and `Review` objects.

Note that the target (scope) of an operation in the Content Model does not exactly correspond to the level (see Section 4.4.2.1) of the operation, but they come close. Class-level operations are always connected to the header of the corresponding content class. Similarly, association-level operations are always connected to the association or to the association class, for which they provide data management functionality. However, instance-level operations are more diverse. If the operation concerns the whole object (e.g. *edit*), it is connected to the class. Otherwise, i.e., if the operation manages a single attribute, then it is connected directly to the attribute.

The semantics of model connections between the Content Model and the Operation Model are straightforward. They visualize for each operation the manner of data access, i.e., what is accessed by the operation and how it is accessed (read/write). Also, the semantics of data access connections of standard operations is the same for all classes, e.g., the *getter* operation provides read access to a single attribute, no matter to which attribute in which class it belongs.

Therefore, an appropriate CAWE tool may generate these connections automatically. Note that composite and custom operations do not have data access connections to the Content Model. Both operations combine standard operations, which have their own connections.

4.4.4 Notation Variations

Although data access connections can be very helpful to provide a quick overview over the way operations access content, their number can grow fast. Note that Figure 4.20 shows only a single operation class and it even omits some operations. However, similarly to the Content Model the flashWeb method defines alternative notations for the Operation Model as well. These notation variations ensure that data access connections, which are not in the focus of concern, may be temporarily hidden.

The operation model in Figure 4.21 depicts the `Review` class, which is displayed with this minimzied notation. Observe the *arrows* symbol on the right-hand side of the class header indicating that the class is minimized and that operations of this class are hidden. Given an appropriate CAWE tool that is able to manage flashWeb models, the developer may minimize all operation model classes that are not relevant for him and focus on a few classes that are in the focus of his interest. Analogously to the content model, operation packages may be represented with the minimized notation as well.

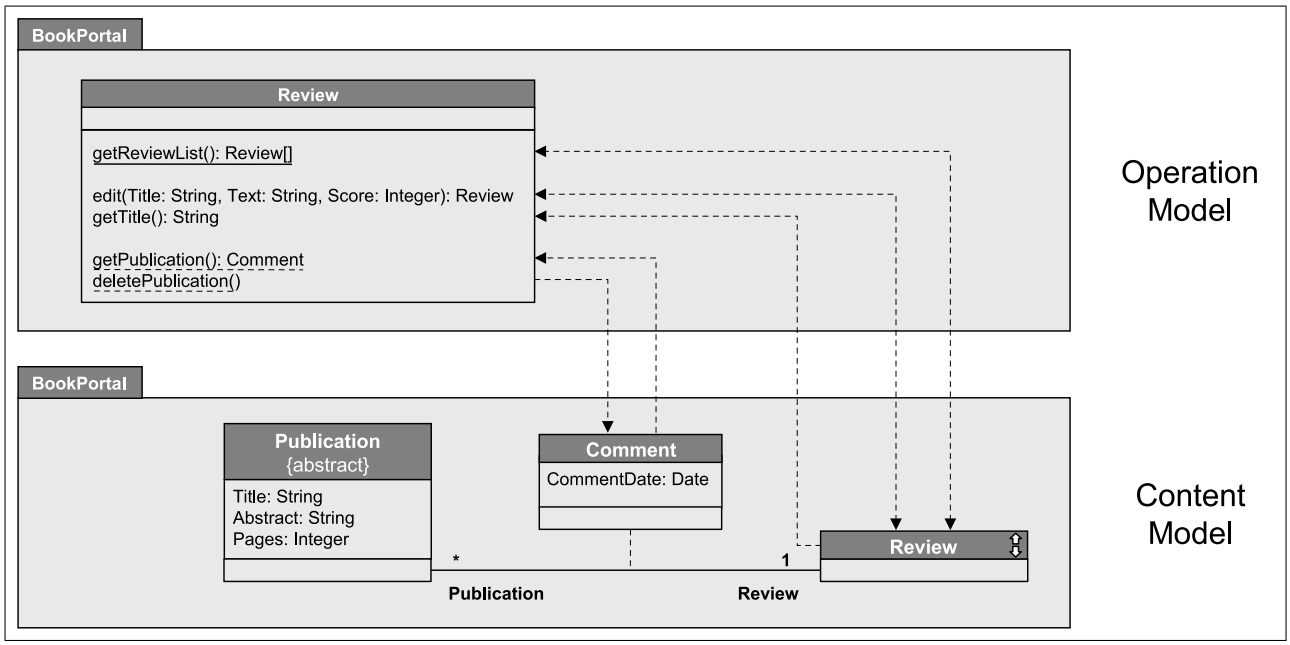

Figure 4.21: Minimized `ContentClass` Example

Note that minimized notations in the content model and the operation model also effect the visualization of data access connections between the models. For example, if a data access connection points from a *getter* operation to an attribute of a content class and the visualization of the class is changed to the minimized notation, then the corresponding attribute is hidden and the data access connection cannot be attached to it. Thus, if model elements that are connected through data access connections are minimized, then the connections have to be adapted as well. The minimized notations may be applied to different models (content and operation)

and at different levels (classes, packages), thus, if the developer employs the minimized notation, then a large number of different configurations may occur. Subsequently, the most important variations are introduced using the example from Figure 4.20. However, for the sake of simplicity, subsequent examples omit most operations of the `Review` operation class. The remaining operations are sufficient to demonstrate the nature of graphical minimizations that affect data access connections. Note that the `Publication` operation class and its connections are omitted as well. Figure 4.21 depicts the case where a content model class is minimized.

This minimization step affects only data access connections that were attached to attributes of the `Review` content class, i.e., *getter* and *setter* operations of the `Review` operation class. The only difference is that data access connections cannot be attached to the corresponding attributes anymore, thus they are attached to the left-hand side of the class header. This way it is signalized that these connections have different semantics than connections, which are attached to the top of the class header.

A further minimization step may be applied to an operation class. This case is depicted in Figure 4.22.

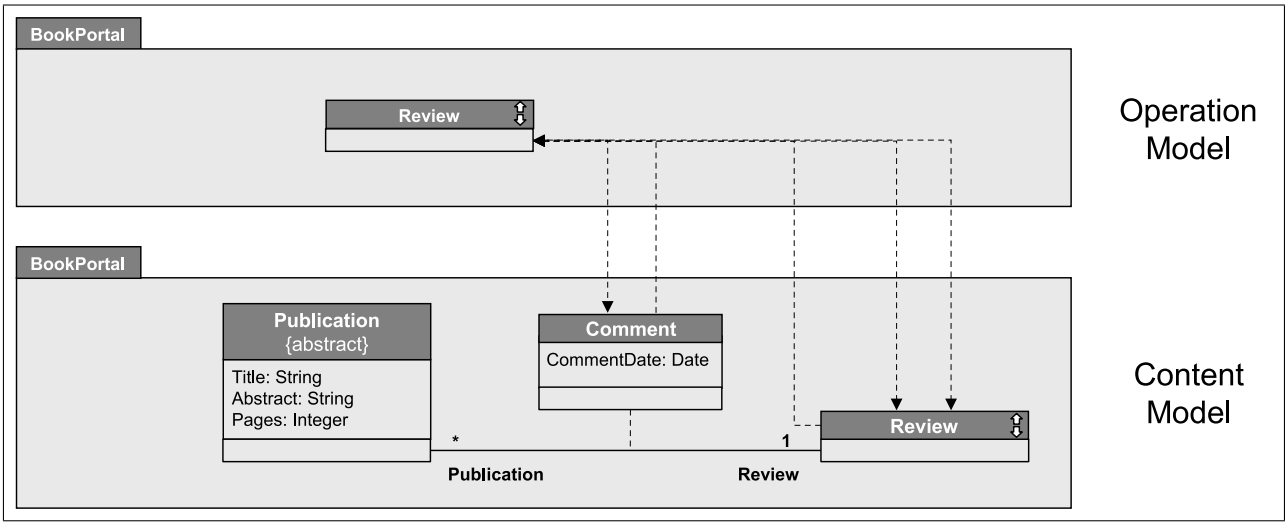

Figure 4.22: Minimized `OperationClass` Example

In this example, the `Review` operation class is minimized, thus all operation signatures of the class are hidden. Of course, data access connections that were attached to operation signatures must be adapted accordingly. To this end, the three remaining data access connections that originate from the `Review` class and are attached to the `Review` content class and the `Comment` association class indicate that the `Review` operation class includes one or more corresponding operations. Note that these connections are not directed as they cannot be associated to single operations.

A further option to achieve a minimized notation of the content model is to hide associations of a class as described in Section 4.3.4. Figure 4.23 demonstrates this option by hiding the `Comment` association between the `Review` and `Publication` classes.

Note that both content model classes have the *A*-symbol in their class headers indicating that they participate in an association, which is currently not visible. Of course, if an association or an association class is hidden, then data access connections that were attached to them cannot be visualized. This can be observed in Figure 4.23, where the `Review` operation class has only connections to the `Review` content class.

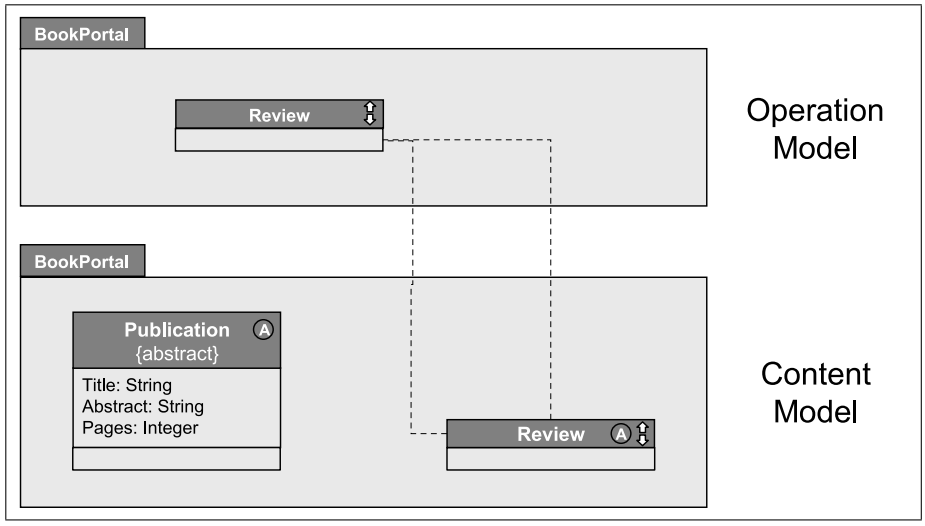

Figure 4.23: Hidden `Association` Example

Finally, both models provide the option to minimize one or more packages. Figure 4.24 depicts an example of a minimized operation package.

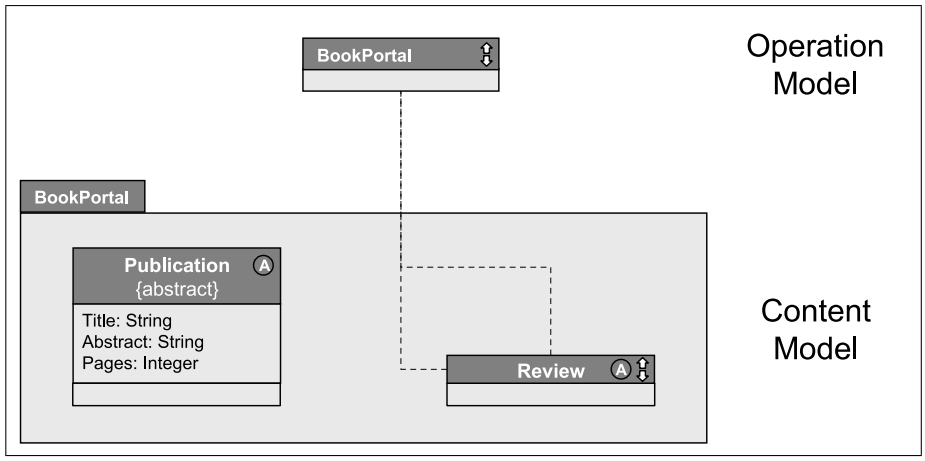

Figure 4.24: Minimized `OperationPackage` Example

Of course, if a package is minimized, there are not many data access connections left to be visualized. The current example depicts two connections from the minimized `BookPortal` operation package to the minimized `Review` content class indicating that the package has an operation class that accesses content defined by the content class. Note that in this setting the `Publication` class normally has the same connections as the `Review` class, because the operation model does define a `Publication` operation class that includes corresponding operations. However, the `Publication` operation class has been omitted in all previous examples for the sake of simplicity, thus its connection to the content model is omitted here as well.

4.4.5 Summary

The Operation Model of the flashWeb approach is a unique characteristic among Web engineering methods. Most methods usually concentrate on content or on hypertext modeling and

neglect handling data management operations properly. Those methods that do consider operations usually do so at an abstract level and integrate operation specifications into existing models. Note that in Section 3.4 a set of typical Web engineering methods has been introduced. Corresponding sub-sections explain how these methods handle operations. For example, the OOHDM (Section 3.4.2) and the UWE (Section 3.4.3) methods do not provide modeling constructs that specify standard operations. With these methods, it is only possible to specify custom operations without predefined semantics which are integrated into their content models. In contrast to that, the WebML method (Section 3.4.4) considers standard content management functionality. However, it integrates operations into its Hypertext Model, which creates an unnecessary complex graphical notation.

The separate Operation Model of the flashWeb method benefits the Web application development process and aids the developer in different ways. It provides a clear separation of concerns, which is helpful for creating and maintaining the models. For example, during the design phase of a development project a domain modeling expert may create the Web application's content model and another expert may create and extend the operation model with a set of business logic operations containing complex computations. Thus, one advantage of separate models is their support for dividing the overall problem into easier manageable pieces and allowing developers to work on partial solutions in parallel. Furthermore, this clear separation of the Web application's design makes it much more easy to overview its functionality, which may be very beneficial, if it has to be extended or errors have to be found. Also the explicit and precise specification of operations makes it possible to integrate them seamlessly into the user interface in a flexible way. Modeling the user interface and integrating content management functionality will be the topic of subsequent sections.

4.5 Composition/Navigation Model

The **Composition/Navigation Model** of flashWeb defines the Web application's user interface. As the name of the model suggests, its purpose is twofold. First, it provides a set of graphical model elements, which allow piece-by-piece composition of user interface pages. Second, it allows to connect different model elements by navigation edges thereby defining the Web application's navigation structure. Furthermore, most model elements may be graphically connected to the Operation Model in order to integrate content management operations into the user interface. Section 4.5.1 provides a detailed overview of general concepts of the model and Section 4.5.2 presents the list of the most important model elements.

4.5.1 General Concepts

Most elements of the Composition/Navigation Model rely on the same set of basic modeling concepts, which are introduced in this section. First, many model elements may contain sub-elements in order to recursively construct parts of the user interface. This aspect is described in Section 4.5.1.1. Second, each model element defines a namespace of variables that may be used to store input and output data for a user interface component. The concepts of variables and namespaces are explained in Section 4.5.1.2. Third, some model elements may be connected to operations of the Web application's operation model. The nature of these connections and their relationship to namespaces is described in Section 4.5.1.3. Fourth, navigation edges, which

define the Web application's navigation structure are explained in Section 4.5.1.4. Finally, Section 4.5.1.5 introduces the notion of conditions, which determine whether a component of the user interface is displayed. Figure 4.25 depicts a generalized example that illustrates the most important concepts of the Composition/Navigation Model.

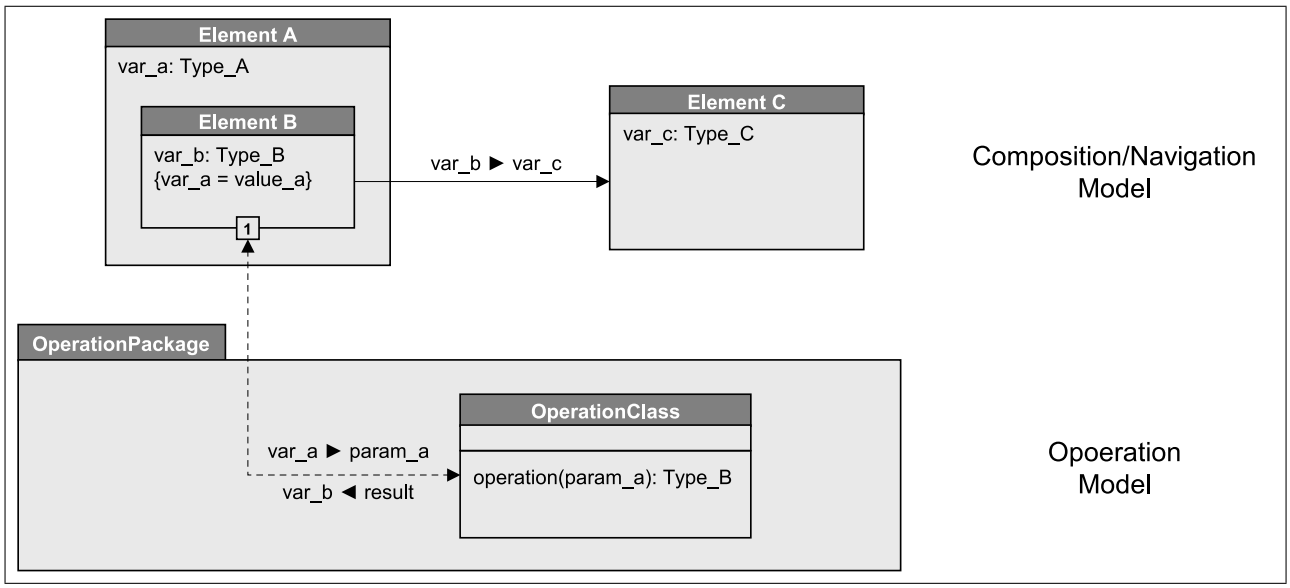

Figure 4.25: Composition/Navigation Model Concepts

4.5.1.1 Recursive Composition

Similarly to the other flashWeb models, the Composition/Navigation Model provides a set of graphical model elements, which may be combined in different ways to build the model, i.e., the Web application's user interface. Each element specifies certain user interface functionality, e.g., data display or data manipulation. In many cases, a model element is allowed to include child elements, which define a certain part of the functionality specified by the element. Using such model elements, the Web application's user interface may be constructed in a recursive manner. The graphical notation for sub-element inclusion is simple and intuitive. All child elements are placed inside the graphical representation of the parent element. Observe the elements *A* and *B* on the left-hand side of the composition/navigation model in Figure 4.25. Element *B* is a child component of element *A*, which is made obvious through the graphical representation. Note that this example contains a fair amount of further information, which is to be ignored at this point. Subsequent sections will provide appropriate explanations.

4.5.1.2 Variables

Each element of the Composition/Navigation Model may define an arbitrary number of variables. A **variable** is specified by a *name* and a *type*, which determines what kind of data the variable may hold. For example, element *A* in Figure 4.25 defines a variable with the name *var_a* and the type *Type_A*. The purpose of a variable is to store data that is required by a certain element, e.g., to display the data. Variable names that are defined for a single element build the element's **namespace**.

Of course, at modeling time there is no data flow and variables do not actually *store* data. This is only the case if the models are used to generate an implementation, which is executed in an appropriate runtime environment. Thus at modeling time, a variable is just a name that may be used in mappings between namespaces in order to indicate which model elements (e.g. operations) provide data and which elements receive it.

Note that in case of nested element structures a child element includes all variables of the parent element's namespace. This fact is not expressed in a graphical manner and has to be kept in mind by the developer. For the current example this means that element *B* also includes the variable *var_a* from the namespace of element *A* into its own namespace. Thus, if element *B* had a child element that would require content of type *Type_A*, it would be no problem to assign the variable *var_a* to it. Note that the expression in curly braces is a condition, which will be explained in Section 4.5.1.5. However, you may observe that the condition also references the variable *var_a* although defined in element *A*.

Also note that the Composition/Navigation Model employs the notion of an element's **context**. The context is the set of those variables from the element's namespace that are actually used by the element. A certain element may define an arbitrary number of variables but usually it only requires a few of them to fullfill its purpose, e.g., to display the data provided by certain variables. The definition of the context also considers nested element structures. If a child element requires a variable, i.e., it is in its context, then it also belongs to the context of the parent element. For example, the context of element *B* in Figure 4.25 includes the variables *var_a* and *var_b*. It uses *var_a* in a condition and it forwards *var_b* to another element (see Section 4.5.1.4). The context of element *A* is the same (i.e. *var_a* and *var_b*), because it is the parent of element *B* and it does not require any further variables. In contrast to that, the context of element *C* is empty because it does not use the variable *var_c* in any way.

4.5.1.3 Operation Calls

A unique feature of the flashWeb method is its ability to connect different models graphically in order to show model dependencies and to define application logic. In Section 4.4.3 connections between the Operation Model and the Content Model were introduced. However, those connections have merely informational value. They express certain semantics that cannot be changed. In contrast to that, connections between the Composition/Navigation Model and the Operation Model are much more dynamic and also much more relevant for the application logic. A single connection of this type is called an **operation call** and specifies which operation of an operation model is used by a certain element of a composition/navigation model.

Elements of the Composition/Navigation Model are generic in nature, thus they do not specify what actual data they display or manage. This information is provided by operation calls. For example, if a certain element can display a list of objects, it does not specify what type of objects it requires or how many of them. If the element is connected to an operation that delivers `Book` objects (see Section 2.2.3), then it displays books, if it is connected to another operation that returns `Author` objects, then it displays authors. Correspondingly, the Web application developer may integrate content and functionality into the user interface in a very flexible and powerful way.

The graphical representation of an operation call is a dashed arrow between an operation signature and the **port** of a Composition/Navigation Model element. The port is represented by a small numbered square at the bottom of the element. Note that depending on the element

type it may possess several ports, which are sequentially numbered. An operation call also specifies a mapping of input and output values between the element and the operation, and the graphical connection representing the operation call may be unidirectional or bidirectional. If an operation requires parameters, then corresponding mappings are provided on the right-hand side of the connection and it is directed towards the operation signature. Note that, if the operation requires a context (see Section 4.4.2.1), then instead of a parameter name the special keyword "context" is used. If an operation returns a value, then a mapping on the left-hand side of the connection maps the return value to a variable of the element's namespace and the connection is directed towards the element's port.

These characteristics may be observed in Figure 4.25. Element *B* has a single operation port, which is connected to the example operation's signature. The operation requires the parameter *param_a*, which is provided by the mapping on the right-hand side of the connection. This mapping specifies that the variable *var_a* is mapped to the parameter *param_a*. The second mapping on the left-hand side of the connection maps the result of the operation to the variable *var_b* in the namespace of element *B*. Note that the connection is bidirectional as the operation requires parameters and returns a value.

4.5.1.4 Navigation Structure

The navigation structure of the Web application is defined through directed solid edges between elements of the Composition/Navigation Model. Similarly to operation calls, navigation edges may also provide mappings that map variables of two different namespaces. The purpose of these mappings is to specify value *transfers* between components of the user interface. Of course, the actual transfer may only take place during the execution of a Web application that has been generated from the model.

Figure 4.25 provides a simple example of a navigation step between element *B* and element *C*. Variable *var_b* from the namespace of element *B* is mapped to variable *var_c* of the namespace of element *C*. Thus, user interface components that may be defined under these elements may access the same value under different names.

4.5.1.5 Conditions

The last concept of the Composition/Navigation Model is the possibility to specify conditions that determine whether a certain user interface component defined by a model element is included into the Web application's user interface. A condition may be included into the specification of any model element and it will be evaluated during the execution of an implementation that was generated from the model. If the condition is satisfied, the corresponding component is included into the user interface.

A condition is specified in curly braces under the variable definitions in the top left corner of a model element. It may use binary arithmetic operators ($=, \neq, <, >, \leq, \geq$) to compare variables and values of basic data types. Additionally, comparisons may be combined with the Boolean operators AND, OR and NOT. Accordingly, a condition is always a boolean expression that may be evaluated to *true* or *false*.

Figure 4.25 includes an example condition for element *B* testing whether the variable *var_a* has the value *value_a*. Note that, instead of a comparison, an expression or a part of it may consist of a single variable. In that case, the expression evaluates to true if the variable exists and has a value that itself evaluates to true. In this context, any value that is not *false* is treated as

true. For example, the condition {var_b} may be used to test whether this variable has actually a value.

4.5.2 Model Elements

There are four basic categories of Composition/Navigation Model elements. **Structure elements** mainly serve as containers for other elements. In most cases, they also serve as child elements allowing the definition of nested containment hierarchies. **Content elements** integrate pieces of the Web application's content into the user interface, i.e., they display one or more objects or calculated values. **Data-entry elements** provide for the user different ways to modify the Web application's content. Data that is selected or entered is forwarded to an operation or to another Composition/Navigation Model model element. **Navigation elements** define navigation edges that connect elements of the Composition/Navigation Model thereby defining the Web application's navigation structure. Note that these categories only define the main purpose of a certain model element. There are several elements that combine more than one of these aspects in some way and therefore belong to several categories. Subsequent sections introduce all elements of the model one by one.

4.5.2.1 User Profile

A `UserProfile` is a structure element that holds the complete user interface specification for a certain user group. Usually, a Web application provides a different interface for different user groups. For example, administrators may access certain areas and functionality of the application, which is prohibited for standard users. Note that user profiles cannot overlap thus a certain profile has to include all elements that define the user interface for a certain user group. Also note that the `UserProfile` element may contain only `Page` elements as direct child elements.

The graphical representation of a user profile employs the same notation as the `Package` element of the Operation Model. Observe the *Standard* user profile on the right-hand side of Figure 4.26 containing a single `Page` element.

4.5.2.2 Page

The `Page` element represents the most basic concept for structuring the Web application's user interface. Of course, for each Web application the number of actual pages may vary considerably. Some applications consist of a few pages that offer information in a highly dynamic manner, e.g., using the Ajax technology set. Other, more conventional applications redirect the user more often to a new page in order to update the user interface. Anyway, the concept of a user interface page (Web page) is indispensable. Accordingly, the flashWeb method employs the `Page` element to model a page of the user interface. Figure 4.26 illustrates the syntax of this element and also provides an example using the Book Portal (see Section 2.2.3 application.

The left-hand side of the figure depicts the generalized notation of the `Page` element. The graphical notation consists of two parts. The header section shows the name of the page and displays a small page icon on the right-hand side, whereas the body section includes variable definitions and an area where child elements may be placed. A `Page` element may contain an arbitrary number of other model elements. Of course, it may not contain other pages because

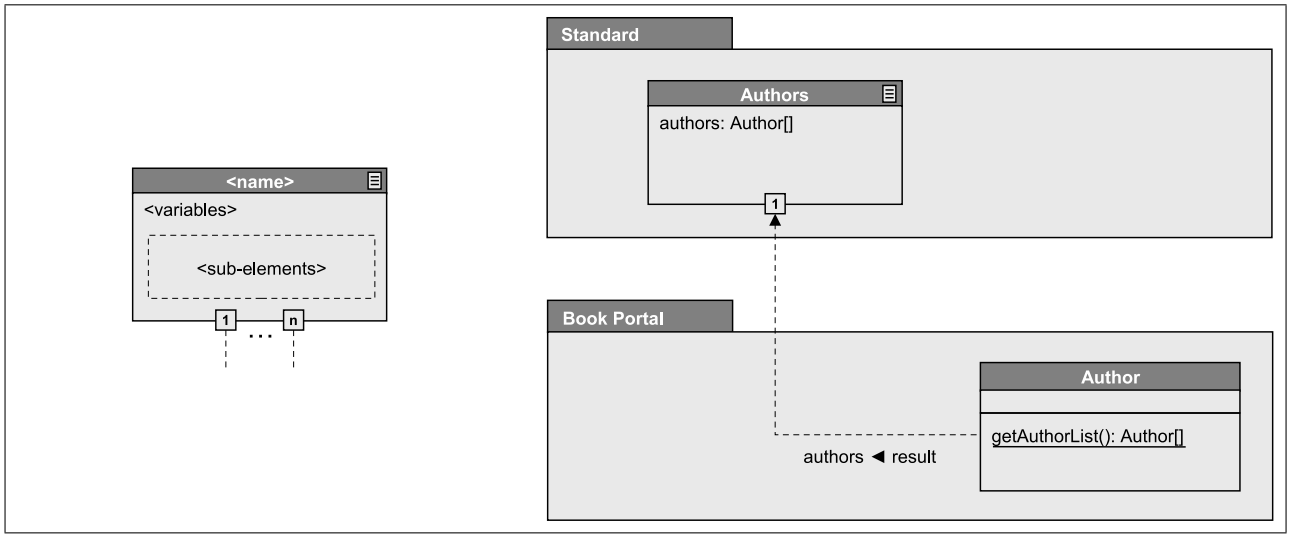

Figure 4.26: The `Page` Element

a page is only allowed as direct child element of a user profile. Also, the `Page` element may have an arbitrary number of ports, which may be connected to arbitrary operations.

The right-hand side of Figure 4.26 depicts a simple example including the *Authors* page, which has no sub-elements but defines the *authors* variable that may store a list of `Author` objects. The only port of the `Page` element is connected to the *getAuthorList* operation of the `Author` operation class which delivers all authors of the Book Portal. In this example the *authors* variable is only used to store authors and is not accessed by any further model element. Of course, the page element could contain sub-elements that make use of the authors list.

4.5.2.3 Area

The `Area` element may be used to partition a single user interface page into several areas. An area element may include an arbitrary number of child elements including other `Area` elements, thus it facilitates the definition of an arbitrarily nested page structure. Figure 4.27 depicts the element's syntax and a simple example.

The graphical notation of the `Area` element, which is depicted on the left-hand side of the figure, is almost identical to the notation of the `Page` element. The element's header displays a name and an icon and the body section contains variable definitions and a sub-element area for child elements. Also it may have an arbitrary number of ports. The only difference is that this element may additionally specify a condition (see Section 4.5.1.5), which appears under the variable specifications.

The right-hand side of Figure 4.27 shows a simple example depicting the *Authors* page, which now includes an `Area` child element with the name "Authors Area". In contrast to the previous example, here the *getAuthorList* operation is connected to a port of the `Area` element and not to a port of the `Page` element. Correspondingly, the area specification also contains the *authors* variable, which is mapped to the result of the operation. Additionally, the area specification includes the simple condition {authors}, which checks whether the variable has a valid value. In this trivial example, this is always the case because the corresponding operation always delivers a true value.

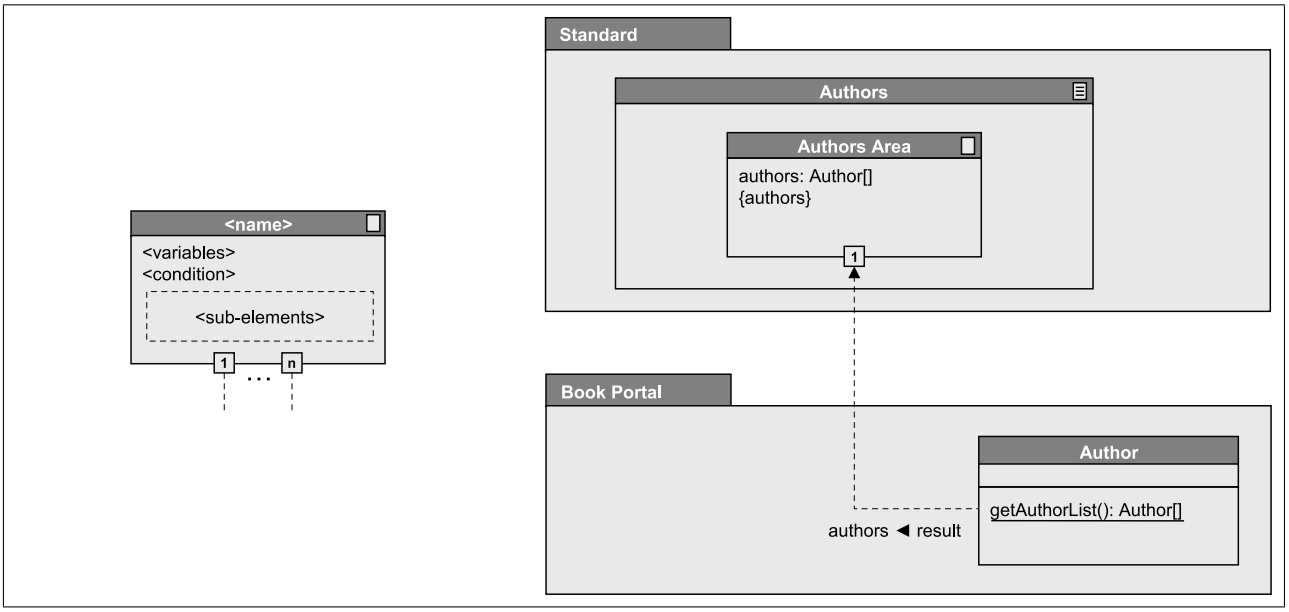

Figure 4.27: The `Area` Element

4.5.2.4 Label

A very simple but important element of the Composition/Navigation Model is the `Label` element. It allows the specification of a simple label that may show arbitrary text provided by the application developer at modeling time. A label may be used as a companion for many other elements that need to be described with a piece of text. For example, a model element that presents a list of content objects usually does not have an explicit header, which states the purpose of the list. Therefore, a `Label` element providing the text "Author List" may inform the Web application's user that the corresponding user interface element represents a list of authors. In contrast to that, the text "Author Index" may indicate that the user may not only view information about authors but also select a single author from the index. Figure 4.28 shows the general notation of the `Label` element and also a usage example.

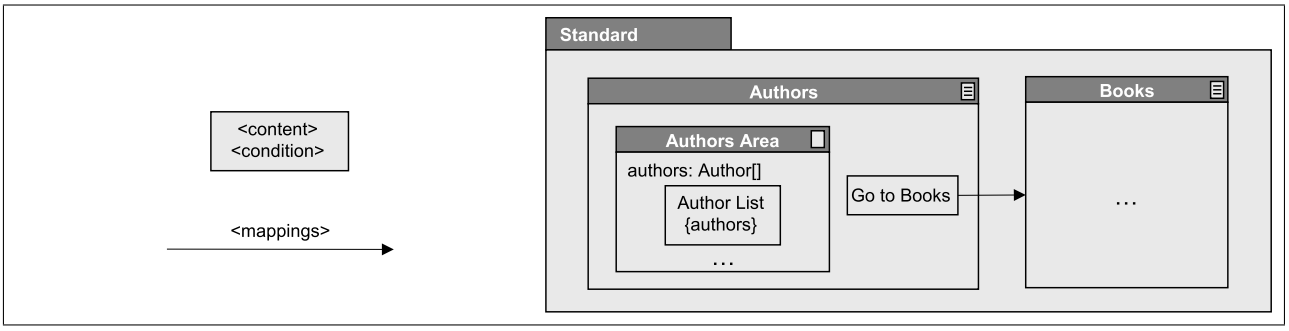

Figure 4.28: The `Label` and `Link` Elements

The general graphical notation of the `Label` element, which may be observed on the left-hand side of the figure is simple. In contrast to most other model elements, it does not have a header but only a body section. It contains the textual content that is to be presented on the user interface and an optional condition.

The right-hand side of Figure 4.28 shows an example for the usage of the `Label` element.

This example includes the *Authors* page, which contains an `Area` element with the name "Authors Area". This area is supposed to show information about authors, e.g., an author listing. To this end, it contains the *authors* variable, which provides the corresponding list of `Author` objects. Note, however, that the model element specifying the author list is omitted in this example for the sake of simplicity. In order to describe the author listing, the example contains a `Label` element with the text "Author List". The element also contains a simple condition, which makes sure that the label is only shown if there are actually any objects to show.

4.5.2.5 Link

The concept of navigation steps that define the navigation structure of the Web application has been already introduced in Section 4.5.1.4. The corresponding member of the Composition/-Navigation Model that may be used to specify a navigation step is the `Link` element. Any model element that provides some kind of navigation functionality relies on this element in order to specify the next node in the navigation path. To this end, the `Link` element is used to graphically connect the anchor element and the target element in the model.

The general graphical notation of the `Link` element is a solid directed arrow, which is depicted on the left-hand side of Figure 4.28. Optionally, the arrow may be labeled with namespace mappings that associate a variable of the anchor element's namespace to the target element's namespace (see Section 4.5.1.2 and Section 4.5.1.4).

A simple example for the usage of the `Link` element is depicted on the right-hand side of Figure 4.28. The *Authors* page contains a `Label` element with the text "Go to Books". This label serves as the anchor element of the link, which points to the *Books* page. The `Label` element and the `Link` element together specify a hyperlink of the user interface. Note that the contents of the *Book* page are omitted for the sake of simplicity.

4.5.2.6 Action Link

The `ActionLink` element specifies a user interface component that allows the Web application's user to take a navigation step and to simultaneously execute a content management action, i.e., execute an operation from the Web application's operation model. To this end, the element combines features of the `Label` and the `Link` elements that were introduced previously and it uses an operation call (see Section 4.5.1.3) allowing to be connected to an operation. Figure 4.29 shows the general notation of this element and provides a usage example.

The graphical notation of the `ActionLink` element that is depicted on the left-hand side of the figure is self-explaining if one is familiar with model elements that have been introduced in previous sections. Basically, it is composed of a `Label` element that has additionally a port, which may be connected to an operation. Furthermore, this element integrates a `Link` element that may be connected to another component of the model, e.g., a `Page` element.

The right-hand side of Figure 4.29 depicts an example illustrating a possible usage scenario for an `ActionLink` element. The *Author* page includes an `Area` element with the name "Author Area", which presents information about an author. Details of this presentation are irrelevant for this example thus they are omitted. Next to this area is an `ActionLink` element baring the text "Delete Author" that specifies a user interface component allowing to delete the `Author` object that is currently viewed from the Web application's storage. This may be an important functionality for a site administrator. Correspondingly, the operation call of the `ActionLink` element is connected to the *delete* operation of the `Author` operation class. The

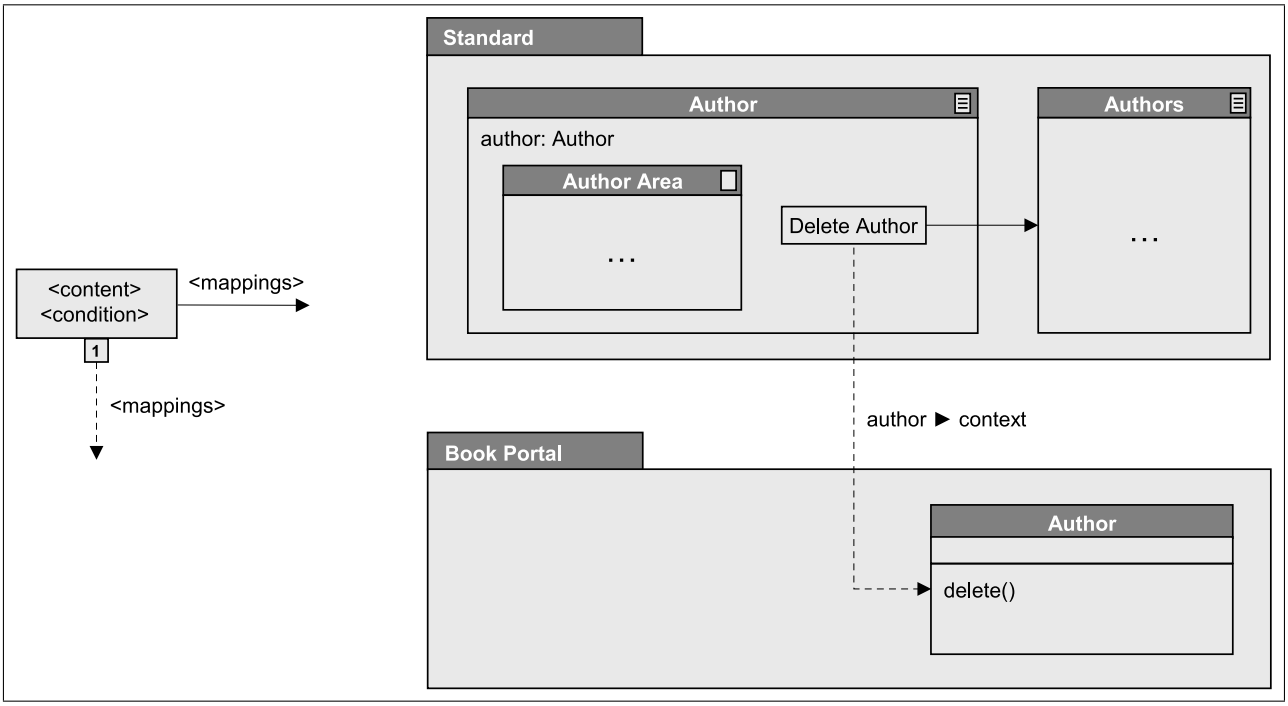

Figure 4.29: The `ActionLink` Element

`Author` object from the *author* variable of the *Author* page is provided to the operation as context. Finally, the link of the `ActionLink` element is connected to the *Authors* page, thus after deleting an `Author` object the Web application's user is directed to a page that for example presents the list of all authors.

4.5.2.7 Content Item

A `ContentItem` element integrates a single piece of information into the Web application's user interface. This information may be a simple value of a basic data type, e.g., the last name of an author or the textual representation of an object, e.g., a comma-separated list of all object attributes. Figure 4.30 shows a general graphical representation of this element and a usage example.

The graphical notation of the `ContentItem` element may be observed on the left-hand side of the figure. The body section of this element displays a solid line, which is the characteristic icon identifying a content item. Besides the icon the body section may contain a condition. The header section shows the element's *context*, i.e., a single variable that provides content for this element. Note that the context for this element may originate directly from an operation call or from a variable of the parent element's namespace. If the context originates from an operation call, then the element's port is connected to an operation in the usual way and the name to which the operation's result was mapped is displayed in the header. However, if the context is a variable from the parent element's namespace, then there is no operation call, the element's port is not displayed and the header just shows the name of the corresponding variable.

These two cases are demonstrated by the example on the right-hand side of Figure 4.30. The *Author* page includes two `ContentItem` elements and defines the *author* variable in its namespace. Note that the origin of a value for this variable is not relevant for this example. Thus, just assume that it stores an `Author` object. The first content item on the left-hand side

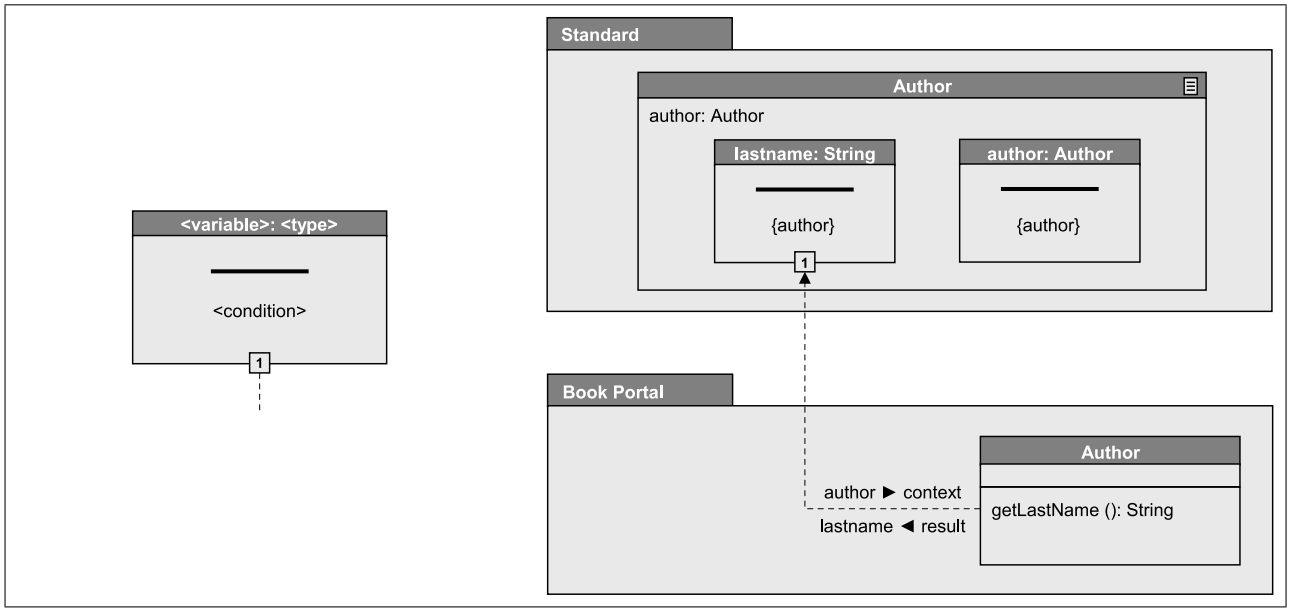

Figure 4.30: The `ContentItem` Element

of the *Author* page is connected to the *getLastName* operation. The *author* variable is mapped to the context of the operation and its result to the name "lastname", which is displayed in the content item's header. Obviously, this content item receives its context, which is the author's last name from the connected operation. This content item also specifies the {author} condition, which makes sure that the item is only shown if the *author* variable exists and has a valid value.

The second content item on the right-hand side of the *Author* page receives its context from the *author* variable of the page's namespace. Accordingly, this variable and its type are shown in the element's header. Therefore, this content item represents an `Author` object in a simple textual form. This content item uses the same condition as the first example.

4.5.2.8 Object View

A very common requirement for a Web application is to display data about a single object in a structured way. This requirement is fulfilled by the `Object View` element. In contrast to the `ContentItem` element, which may also display single values, the `ObjectView` element can also represent an entire object at once. To this end, it provides the list of the object's attributes together with corresponding attribute values. A suitable rendering of this element may be a simple two-column table. The first column may contain attribute names and the second the attribute values. Of course, the model element itself does not suggest a specific rendering, thus alternative representations are possible. Figure 4.31 depicts the general notation of this element and also provides an example demonstrating how it may be used.

The general notation of the `ObjectView` is depicted on the left-hand side of the figure. The element's body section contains the icon, which is a simple rectangle, and a condition, which is optional. The element's header shows the context specification, i.e., a single variable that provides the object to be shown. As usual, the context may originate from an operation call or from a variable of the parent element's namespace. The element possesses a single port that may be connected to an operation or hidden, if the context is a variable from the parent element's namespace. In both cases, the origin of the input is indicated in the header section.

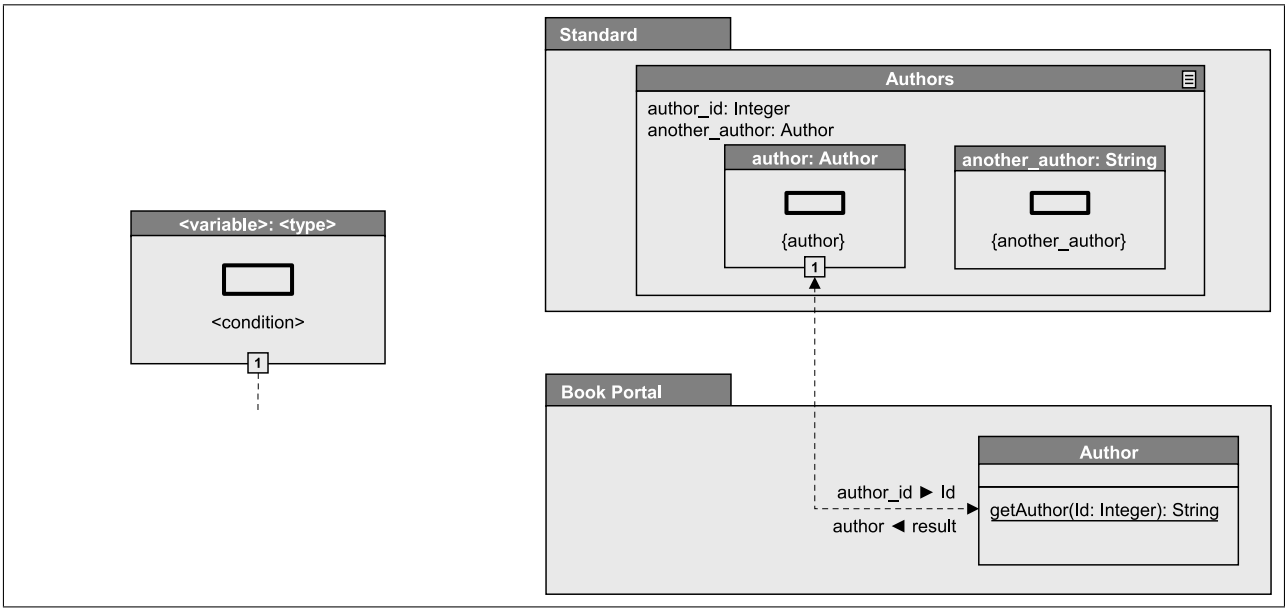

Figure 4.31: The `ObjectView` Element

The right-hand side of Figure 4.31 depicts an example demonstrating both cases. The *Authors* page defines the *author_id* and *another_author* variables. Note that for this example the origin of the values of these variables is not important. Assume that the *author_id* variable stores an integer and that the *another_author* variable stores an `Author` object.

The *Authors* page contains two `ObjectView` elements. The first one on the left-hand side receives its input from the *getAuthor* operation, thus its port is connected to the operation's signature. Note that this operation returns an `Author` object if it is provided an appropriate id. Correspondingly, the *author_id* variable is mapped to the *Id* parameter of the operation. This object view also specifies the {author} condition, thus it is only shown if the operation returns an appropriate result.

The second `ObjectView` element on the right-hand side of the *Authors* page receives its context from the *another_author* variable. This is indicated by the context specification in the element's header. Similarly to the other object view, it specifies a condition, which makes sure that data is only presented if there is a valid input value. Ultimately, both elements represent an `Author` object.

4.5.2.9 Object Editor

The aim of the flashWeb Web engineering method is to allow the specification of Web applications that provide full content management capabilities. To this end, the method utilizes appropriate content management operations (see Section 4.4) that provide read and write access to the Web application's content. However, these operations have to be leveraged by corresponding user interface components in order to allow the Web application's user to access them. The `ObjectView` element that have been introduced previously is concerned with the appropriate presentation of a single content object on the user interface, i.e, it provides read access to the Web application's content. In contrast to that, the aim of the `Object Editor` element is to allow the modification of an existing object, i.e., to provide write access. To this end, the rendering of this element usually includes a Web form, which allows the Web appli-

cation's user to input new values for object attributes. Figure 4.32 depicts the general notation of this element and provides a corresponding usage example.

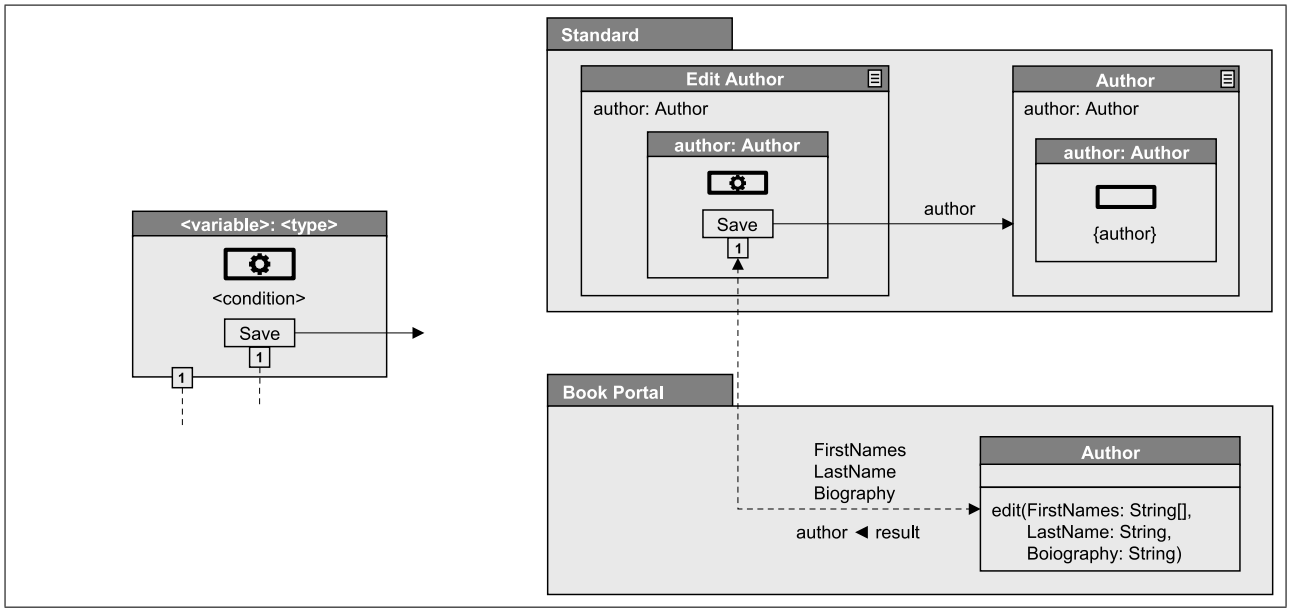

Figure 4.32: The `ObjectEditor` Element

The left-hand side of the figure depicts the general notation of the `ObjectEditor` element. The element's header shows the context, which is a single variable containing an object that is to be modified. The body section contains an icon and an optional condition. Also the element has a single port that may be connected to an operation that delivers the context object. Of course, as usual the context may originate from the parent element's namespace in which case the port is not displayed. Finally, the `ObjectEditor` element has a single `ActionLink` (see Section 4.5.2.6) as sub-element. This sub-element has a single port that may be connected to an operation that actually carries out the modification of the context object. Additionally, the element has a navigation arrow that may be connected to a `Page` or an `Area` element. A possible rendering for the `ActionLink` sub-element is a user interface button that may be activated by the Web application's user in order to call the corresponding operation and possibly to navigate to another user interface page.

The right-hand side of Figure 4.32 depicts an example that illustrates how the `ObjectEditor` element may be used. The *Edit Author* page on the left-hand side contains a single `ObjectEditor` element, which has the *author* variable from the page's namespace as context. The port of the contained `ActionLink` element is connected to the *edit* operation of the `Author` class which is responsible for the modification of an `Author` object. The labels of this connection indicate that the result of the operation is mapped to the *author* variable and that the `ObjectEditor` element provides values for all attributes of an `Author` object as parameters of the *edit* operation. Note that a trivial mapping, which has the same name as source and target, e.g., "FirstNames ► FirstNames" is abbreviated through "FirstNames". Finally, the navigation arrow of the `ActionLink` element is connected to the *Author* page, which presents information about the edited `Author` object utilizing an `ObjectView` element.

4.5.2.10 Object Creator

The Composition/Navigation Model of the flashWeb method supports all standard content management operations that are offered by the method's Operation Model (see Section 4.4). Accordingly, it does not only provide user interface components for presenting or modifying existing content objects but also an appropriate component for creating new instances. This content management operation is facilitated by the `ObjectCreator` element. Figure 4.33 depicts the general notation of this element and provides an example.

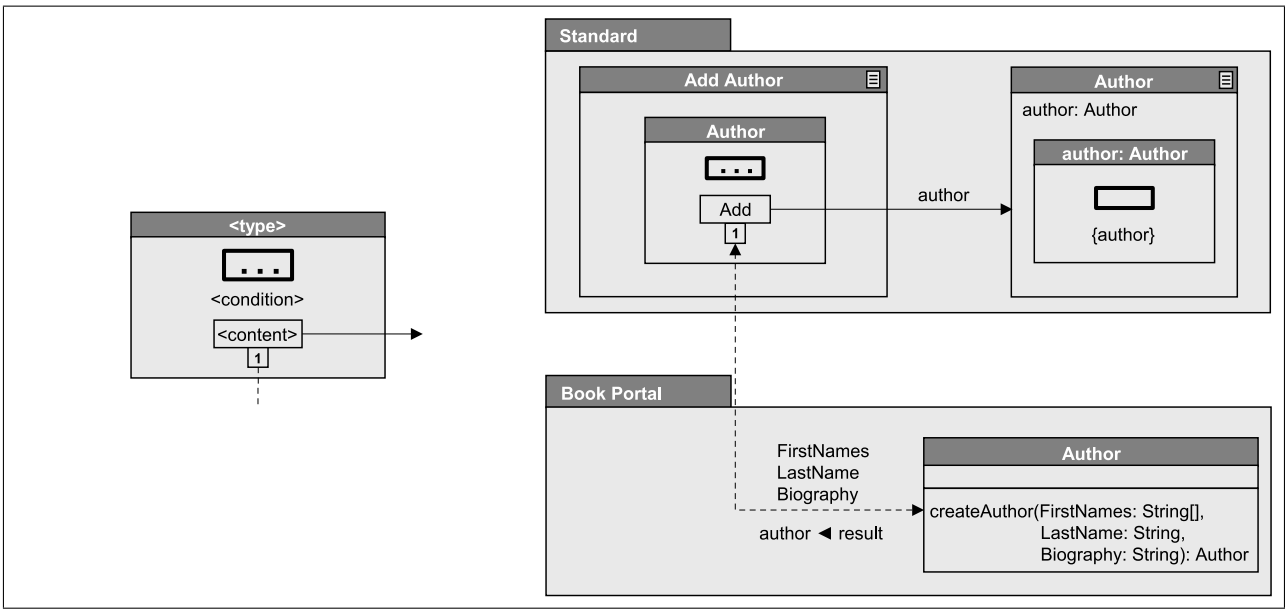

Figure 4.33: The `ObjectCreator` Element

The left-hand side of the figure depicts the general notation of the `ObjectCreator` element. The element's header section specifies the type of the object that is to be created. The body section contains an icon (rectangle with three dots) an optional condition and an `ActionLink` sub-element. The operation port of the `ActionLink` may be connected to a *create* operation that actually creates a new instance of the designated type. Additionally, the navigation link of this element may be connected to a page that is presented to the Web application's user after the new object has been created. A typical rendering of the `ObjectCreator` element includes a form that allows the user to input values for each attribute of the object to be created. The `ActionLink` sub-element is usually rendered as a button, e.g., bearing the label "Add".

The right-hand side of Figure 4.33 depicts a usage example for the `ObjectCreator` element. The example includes two pages. The first one is the *Add Author* page, which contains an `ObjectCreator` element that is set up to create an `Author` object. To this end, the operation port of the `ActionLink` sub-element is connected to the *createAuthor* operation of the `Author` operation class. The `ObjectCreator` element provides all necessary parameters for the operation, i.e., the first name, the last name, and the biography of the author. The return value of this operation is the newly created `Author` object. Observe the navigation link of the `ObjectCreator` element, which forwards the newly created object to the *Author* page. This page contains a simple `ObjectView` element that presents the newly created object.

4.5.2.11 Object List

A common presentation pattern for Web application data is an object listing. Of course, there are different ways to represent a list of objects thus an appropriate model element has to be customizable. This task is tackled by the `ObjectList` element. It supports two different ways to present an object listing. First, if the presentation is not further specified by one or more sub-elements, then this element provides a standard representation. For example, the list of objects may be displayed in a simple table having a column for each object attribute and a row for each object showing appropriate attribute values. Second, the appearance of an entry in the object list may be specified arbitrarily by one or more sub-elements. For example, several `ContentItem` elements may be used as sub-elements of the `Object List` element in order to achieve a customized representation. To this end, the `ObjectList` element provides a special *item* variable that represents a single entry from the list of objects that are presented. Sub-elements of the `ObjectList` may use this *iterator* variable as their context.

Figure 4.34 depicts the generalized notation of this element as well as some examples.

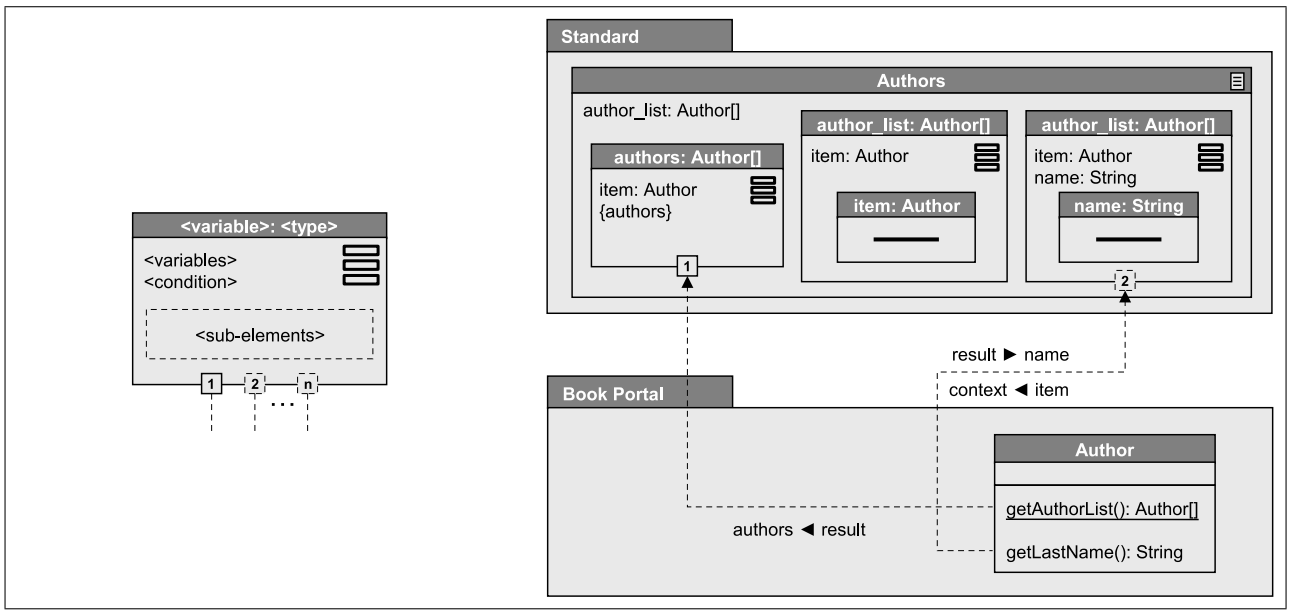

Figure 4.34: The `ObjectList` Element

The general notation of the `ObjectList` model element is depicted on the left-hand side of the figure. The notation employs some usual concepts of the Composition/Navigation Model. The element's header section shows the context specification, the body section displays an icon, includes variable specifications, a condition, and a sub-element area. Furthermore, the element may have an arbitrary number of ports. However, as already pointed out, there are two different ways to set up a `ObjectList` element and its features have to be combined accordingly.

The first setup provides a simple standard representation of the object list, e.g., a tabular view. To this end, the element's port has to be connected to an operation or it may receive its context from the parent element's namespace. Additionally, a condition may be specified. This case is demonstrated by the `ObjectList` element on the left-hand side of the *Authors* page in Figure 4.34. The element's first port is connected to the *getAuthorList* operation, which returns a list of `Author` objects. The operation's result is mapped to the *authors* variable, which is

displayed in the element's header, i.e., it is the element's context. Furthermore, the variable is used in the element's condition.

The second case can be somewhat more complex, because the `Object List` element includes one or more sub-elements that specify the presentation of listed objects. In this case, the element includes the special *item* variable, which may be used by sub-elements as context. This variable represents a single object from the object list. Consider the second example in the middle of the *Authors* page. The context of this particular `ObjectList` element is the *author_list* variable from the *Authors* page. The element contains a *ContentItem* sub-element, which has the *item* variable as context. Accordingly, the presentation of each element in the object list is defined by the `ContentItem` sub-element, which may be rendered as a comma-separated list of attribute values.

Of course, an `ObjectList` element may have an arbitrary number of sub-elements that together specify the representation of an object list entry. The third example on the right-hand side of the *Authors* page demonstrates how so-called *iteration ports* allow the specification of operation calls that provide values, which may be included into the representation of an object list entry. Note that the first port of an `ObjectList` element may be only connected to an operation that delivers the general input for the element, e.g., a list of objects. Any further ports of the element are iteration ports, which are assigned to the presentation of a single object. In an actual implementation, all operations that are connected to iteration ports are executed for each entry of an object listing. Note that an iteration port is represented by a dashed box in contrast to a standard port that is depicted by a solid box. The context of the `ObjectList` example on the right-hand side of the *Authors* page is the *authors_list* variable from the page's namespace. Note that the origin of content for this variable is irrelevant for this example. Of course, the author objects could originate from the *getAuthorList* operation like in the first example. Note, however, that the second port of the `ObjectList` element is connected to the *getLastName* operation of the `Author` operation class. It uses the *item* variable as context and delivers the last name of an author. This value is bound to the *name* variable, which, in turn, serves as the context for the `ContentItem` sub-element. Altogether, this example specifies an author list, which displays for each author his last name.

4.5.2.12 Object Index

It is often the case that a Web application user wants to know more about a specific entry if he is confronted with a list of objects. In this case, it is desirable for the user to be able to select the corresponding entry, for example, by clicking on it in order to be directed to a page presenting the follow-up information. This functionality is provided by the `ObjectIndex` element. Of course, it is very similar to the `ObjectList` element, which represents a list of objects without allowing the selection of an entry. Figure 4.35 depicts the generalized notation of this element and a usage example.

The notation of the `ObjectIndex` element presented on the left-hand side of the figure is almost identical to the notation of the `ObjectList` element. Correspondingly, all explanations and examples provided in Section 4.5.2.11 are also valid for this element. The first of two minor differences is a different icon, which is to be observed in the top-right corner of the element's body section. The second one is the additional navigation link, which allows to connect the element to a follow-up page that usually represents detail information about the selected entry. The navigation link may bear arbitrary variable mappings between the element's namespace

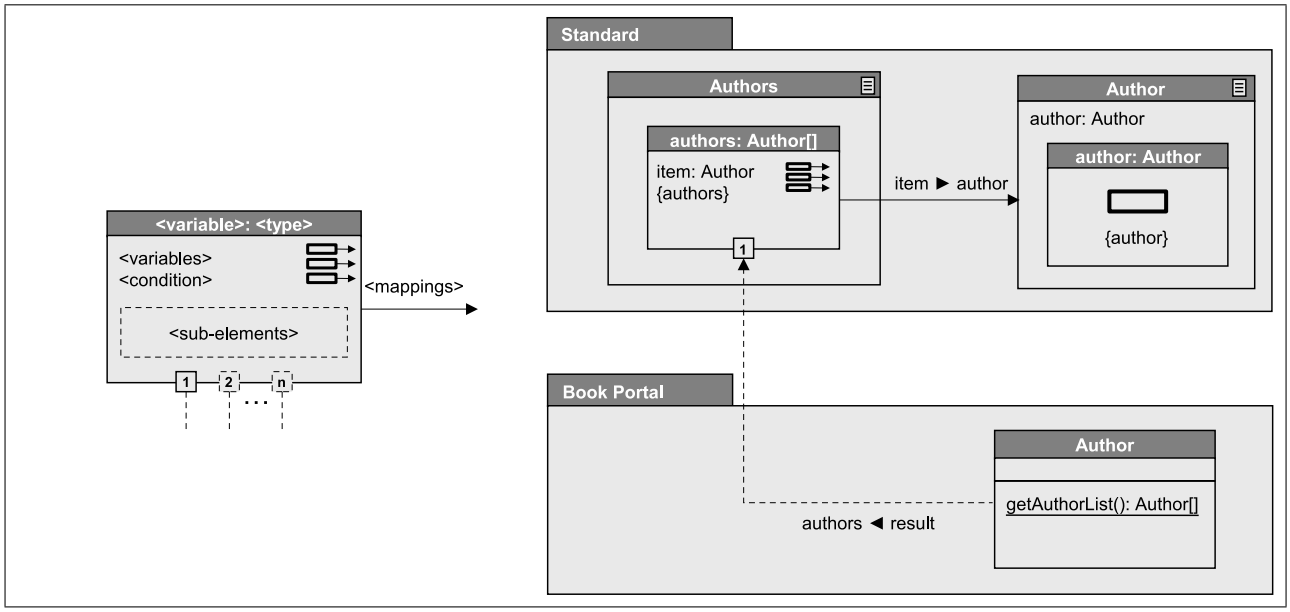

Figure 4.35: The `ObjectIndex` Element

and the namespace of the follow-up page. The special *item* variable in the element's namespace holds the entry that was selected by the Web application's user.

The right-hand side of Figure 4.35 depicts a simple usage scenario. The *Authors* page includes a single `ObjectIndex` element that specifies an index of authors. To this end, the element's first port is connected to the *getAuthorList* operation of the `Author` operation class. Correspondingly, the result of this operation, which is a list of `Author` objects constitutes the element's context. The *item* variable, which is of course of the `Author` type represents the selected `Author` object. This variable is used in the mapping of the element's navigation link, which points to the *Author* page. The *item* variable is mapped to the *author* variable of this page and is used by an `ObjectView` element in order to present detailed author information. Note that both the `ObjectIndex` element in the *Authors* page and the `ObjectView` element in the *Author* page include simple conditions, which ensure that the author listing and the author view are only displayed if the corresponding variables contain appropriate values.

Note that the `ObjectIndex` element presented in this example could also contain sub-elements in order to specify a more customized presentation of a single list entry. This possibility is omitted in the current example for the sake of simplicity. Observe the examples in Figure 4.34, which contains such definitions.

4.5.2.13 Multi-level Object Index

Although a simple index is already a very useful structure for allowing the Web application's user to browse through the Web application's content, sometimes it is desirable for an index to have multiple levels. An extension of the author index example from the previous section may additionally include the list of an author's books on the follow-up page, which is presented to the user after he selects an author entry from the index. However, this information may also be presented by an index with two levels on a single page. The first level may show authors and the second level for each author a list of his books. Listings and indexes with multiple levels are usually rendered with a tree-like representation. This functionality is provided by

the Multi-level Object Index element. Figure 4.36 depicts the generalized notation of this element and an example implementing the mentioned two-level author index.

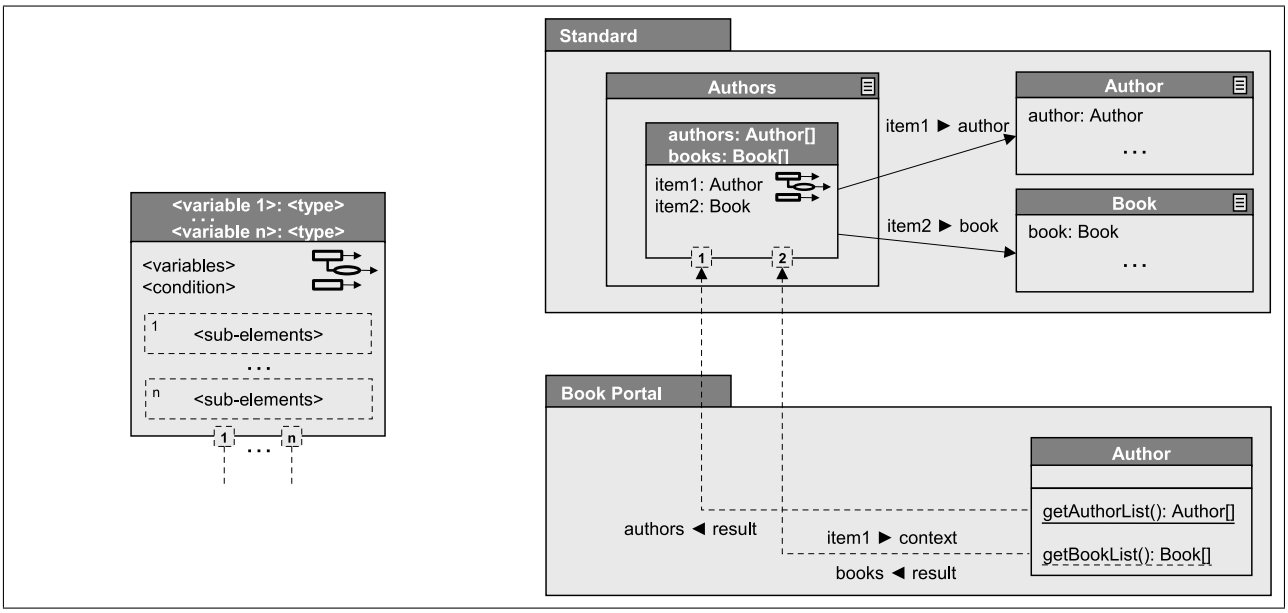

Figure 4.36: The MultilevelObjectIndex Element

Naturally, the general notation of the MultilevelObjectIndex element is similar to the notation of the ObjectIndex element, however there are some differences. First, instead of a single variable the element's header contains a set of variables, one for each level of the index. Second, the element's body section contains for each level a separate sub-element area, which may arbitrarily define the representation of entries at the corresponding level. Third, the element provides for each level an operation port, which may be connected to an appropriate operation that delivers values for the corresponding level. Note that this usage of operation ports differs from that of other elements. For example, a Page element may have an arbitrary number of ports that may be used independently. In contrast to that, an ObjectList or ObjectIndex element has a first port that is responsible for delivering the initial list of objects to be shown and all other ports, so-called iteration ports, are activated once for each element of the initial list. Ports of a MultilevelObjectIndex element perpetuate this concept indefinitely. Every port of this index depends on the port of the previous level, thus it is activated once for every entry of the previous level. Note that all ports of the MultilevelObjectIndex element are depicted with dashed squares in order to indicate that they provide analogous functionality.

The right-hand side of Figure 4.36 depicts a simple example of a MultilevelObjectIndex. The *Authors* page includes the index that lists authors at the first level and an author's books at the second level. To this end, the index has two ports that are connected to appropriate operations of the Author operation class. The first port is connected to the *getAuthorList* operation and delivers the list of all Author objects to the *authors* variable and the second port is connected to the *getBookList* operation that delivers Book objects for an author to the *books* variable. Both variables are displayed in the element's header section. Note that the MultilevelObjectIndex element utilizes a special variable for each level of the index which may hold a single object of the corresponding type. For example, the *item1* variable belonging to the first level is of the Author type and may hold a single object of this type. These desig-

nated variables serve two purposes. First, they may be used as context for subsequent levels. Observe that the *item1* variable holding an appropriate `Author` object is provided as context for the *getBookList* operation. Second, they identify the object at a certain level that is selected by the user. This is the case for both navigation links of the index element. The *item1* variable is mapped to the *author* variable of the *Author* page and the *item2* variable is mapped to the *book* variable of the *Book* page. The rendering of this example on the user interface is straightforward. The *Authors* page shows a two-level index. If the user selects an author at the first level, he is directed to the *Author* page, if he selects a book at the second level, he is directed to the *Book* page. Note that these follow-up pages are not detailed in Figure 4.36. Also note that the representation of objects at the two levels could have been arbitrarily specified using sub-element areas and further model elements.

4.5.2.14 Set Navigator

A presentation pattern that is often provided by Web engineering methods is the -called *guided tour*, which allows the Web application's user to visit several items of a homogeneous list in a sequential manner. Usually, the view of an item provides links to the previous and the next items of the list. This presentational pattern is supported by the `SetNavigator` element. Adhering to the flexible nature of the flashWeb method, this model element is highly customizable. It may represent any homogeneous set of objects that are returned by any operation of the operation model. Furthermore, the actual object view that shows a single element of the object list may be constructed using arbitrary elements of the Composition/Navigation Model, e.g., one or more `ObjectView` or `ContentItem` elements.

Note that the `SetNavigator` element is very similar to the `ObjectList` element that was introduced in Section 4.5.2.11. The minor difference is that in contrast to the `ObjectList` element, which presents a list of objects at once, the `SetNavigator` element presents one object at a time and allows navigating to the next or previous objects. Note that regarding the categories introduced at the beginning of Section 4.5.2, this element represents content and it also provides navigational functionality.

The general syntax of the `ObjectList` and the `SetNavigator` elements is almost identical except for the different icons. In order to avoid repetition, subsequent explanations are kept short. All descriptions of the previous section also apply to the `SetNavigator` element including the examples provided in Figure 4.34. Figure 4.37 depicts the generalized notation of this element and an example of a somewhat more advanced usage of the `SetNavigator` element.

The generalized notation of the `SetNavigator` element on the left-hand side of the figure indicates the main characteristics of the element. The notation is identical to that of the `ObjectList` element except for the different icon in the top-right corner of the element's body section.

The example on the right-hand side of Figure 4.37 demonstrates how elements of the Composition/Navigation Model and operations may be combined in a flexible way to define the view that is presented to the Web application's user for each object of the set. The current example specifies a guided tour of authors and shows for each author some detail information and the list of the author's books. To this end, the *Authors* page contains a `SetNavigator` element, which is connected to the *getAuthorList* operation that delivers a list of `Author` objects. The *item* variable, which in this case represents a single `Author` object, is used by the

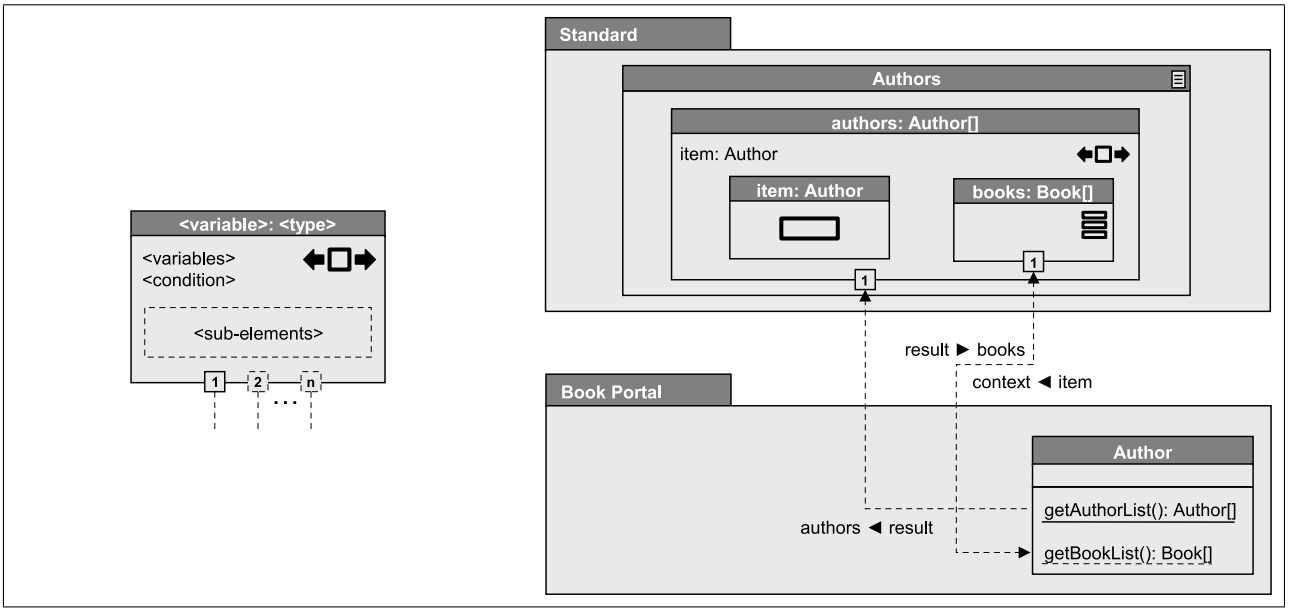

Figure 4.37: The `SetNavigator` Element

two sub-elements of the `SetNavigator` element. The first one is an `ObjectView` element that uses the *item* variable as context in order to define a simple view of an author. The second sub-element is an `ObjectList` element that specifies the list of the author's books. To this end, the *item* variable is passed to the *getBookList* operation as context, which delivers a list of `Book` objects. This operation is connected to the first port of the `ObjectList` element, thus it delivers it's context, i.e., a list of books.

4.5.2.15 Menu

A common component of Web application user interfaces is a menu that directs the Web application user to different parts of the application. The Composition/Navigation Model offers the `Menu` element that provides corresponding functionality. Figure 4.38 depicts the general notation and a simple menu example.

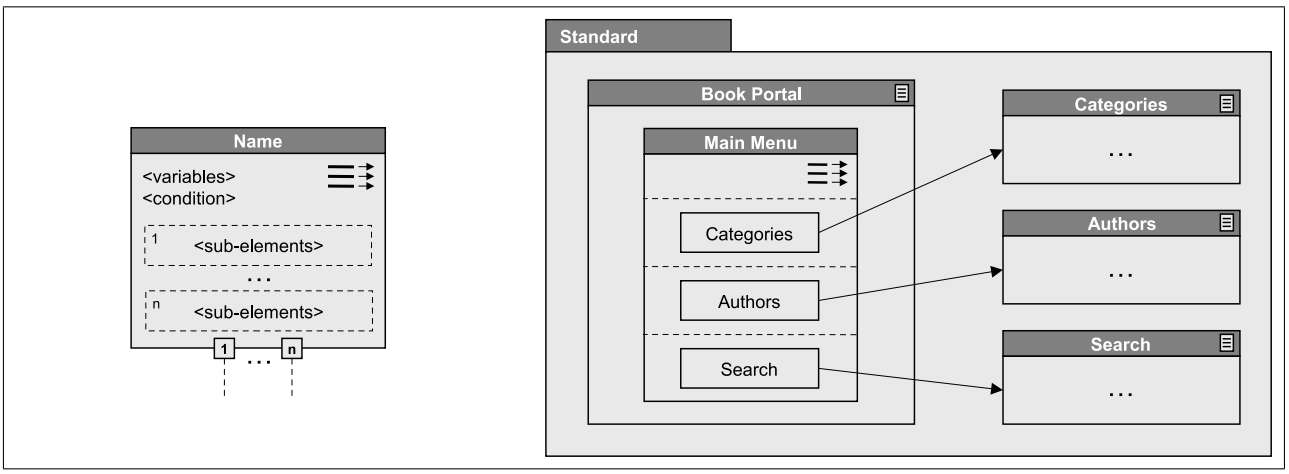

Figure 4.38: The `Menu` Element

The notation of the Menu element, which is depicted on the left-hand side of the figure, relies on the usual concepts of the Composition/Navigation Model. Similarly to an Area element, the header section shows the menu's name, the body section holds an icon, variable definitions, and a condition. A minor difference is that this element may have an arbitrary number of subelement areas, one for each menu entry. Of course, each sub-element area may contain an arbitrary definition of the corresponding menu entry, however, usually a simple definition suffices, e.g., a Label element combined with a Link element. Finally, as many other model elements, the Menu element may have an arbitrary number of operation ports.

The right-hand side of Figure 4.38 depicts an example, which defines the main menu of the Book Portal application (see Section 2.2.3). The Menu element in the *Book Portal* page includes three entries. Each entry is composed of a Label element and a Link element that points to a corresponding follow-up page. As entries of the main menu are simple strings, it suffices to use simple labels to specify them. Of course, in some cases menu entries may be more dynamic, thus the usage of operation ports may be required to provide appropriate values and ContentItem elements may be necessary to represent them.

4.5.2.16 Form

The ObjectCreator and ObjectEditor elements of the Composition/Navigation model specify user interface components that allow the Web application's user to create and modify a content object respectively. These interface components utilize appropriate forms that contain fields for each attribute of the corresponding object, thus the user may specify initial attribute values for a new object or provide new values for an existing object. These values are used as input for the *add* and *edit* operations of the target object's operation class. However, the operation model of a Web application usually includes a set of further content management operations that may have alternative signatures. For example, *setter* operations require a single parameter and custom operations (see Section 4.4.2.7) may have an arbitrary signature. The integration of these operations into the Web application's user interface requires a customizable form, which may be adapted to the operation's requirements in a flexible manner. This problem is tackled with the Form model element, which allows the specification of a Web form with arbitrary fields, specified by the Field element, thus it may be adapted to the parameter requirements of an arbitrary operation of the operation model. Figure 4.39 depicts the general notation of the Form and Field elements and provides a combined usage example.

The left-hand side of the figure displays the Form and Field elements. The Form element, which is depicted at the bottom bears the name of the form in the element's header section. The body section contains an icon, an optional condition, a set of field areas, and an ActionLink element (see Section 4.5.2.6). Each field area may contain a single Field element that provides the specification of one field of the form. The label of the ActionLink element may specify an arbitrary string, however, values like "Save" or "Submit" are common. The operation call of the ActionLink element may be connected to an appropriate operation that uses the values provided by the form. Finally, the element's navigation link may be connected to a Page element that specifies the user interface page that should be presented to the user after activating the form.

The Field element may be used in conjunction with the Form element to specify the fields of the form. To this end, a single Field element is placed into each field area of the form, which may have an arbitrary number of fields. The body section of the Field element shows

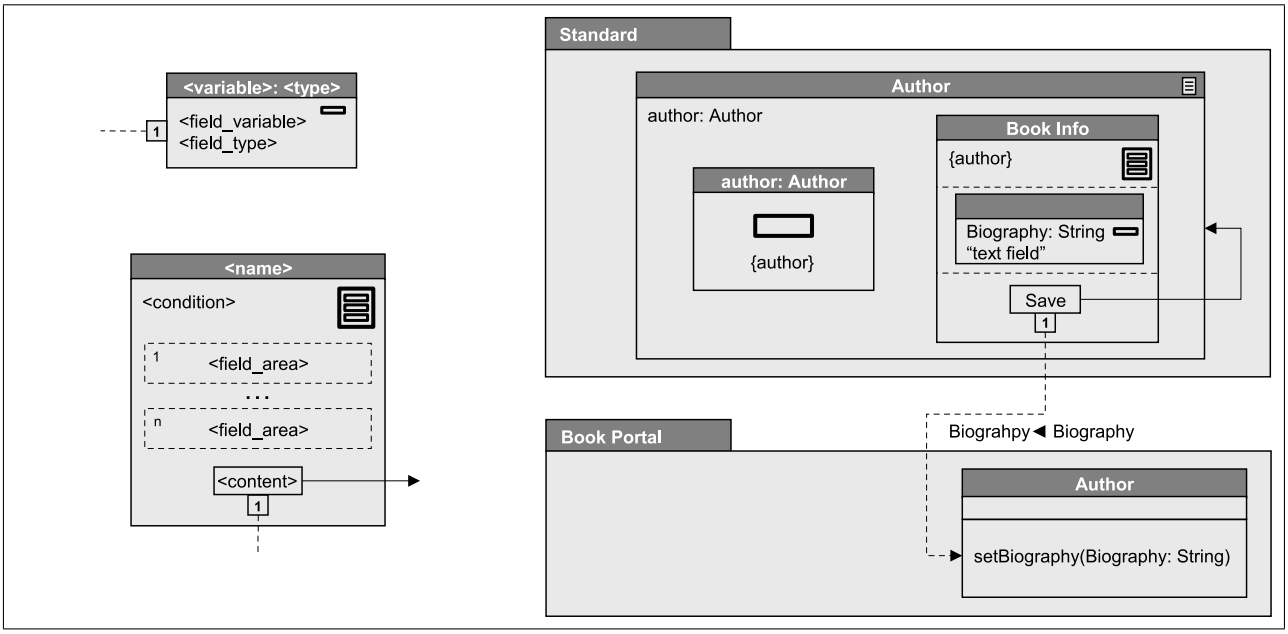

Figure 4.39: The `Form` Element

an icon and specifies the field variable and the field type. The *field variable* is used to map the field to a parameter of the connected operation. Note that the field variable has an appointed data type that specifies what kind of data is required by the field. The *field type* specifies the type of the field, i.e., whether a single value or multiple values are required or how the user may interact with the corresponding user interface element. Common field types are "input field", and "text field", which specify a single-value field for *String* data or "selection" and "multiple selection", which allow the Web application user to select one or more values from a predefined list. As a first possibility, the operation port of the `Field` element may be used to connect the element to an operation that delivers an appropriate value or list of values to be selected from. Alternatively, this context information may also originate from a variable of the parent element's namespace. In both cases, the header section of the `Field` element shows this context specification. If the header section is empty, i.e., no context is specified, the field has no predefined value, thus the user has to provide it.

The right-hand side of Figure 4.39 shows a simple usage example for the `Form` element. The *Author* page displays information about an author employing an `ObjectView` element that uses the *author* variable of the page as context. Additionally, this page includes a `Form` element that allows to update the author's biography. To this end, the form contains a single `Field` element, which has the type "text field". The field specifies the *Biography* field variable of the *String* type which is used in the mapping of the field's operation call to associate it to the *Biography* parameter of the *setBiography* operation. Note that the *author* variable from the page's namespace is provided to the operation as context. Finally, the navigation link of the forms `ActionLink` element points to the *Author* page, thus after activating the form the Web application's user remains on the same page.

4.5.2.17 Association Creator

Previous sections have introduced different model elements that support content management operations concerning the manipulation of a single object. The task of corresponding user interface components is to allow the Web application's user to view to create or to modify a single content object. Note that there is no specific model element that supports the deletion of an object. This task may be simply achieved by utilizing an `ActionLink` element and connecting it to the appropriate *delete* operation.

However, manipulating objects in an isolated manner is only one side of the coin if it comes to content management. Another important aspect is the management of relationships between objects. The first element of the Composition/Navigation Model that targets this problem is the `AssociationCreator` element. It specifies a user interface component that allows the Web application's user to associate two content objects regarding associations that have been defined in the Web application's content model. Considering the Book Portal application, an appropriate example is associating `Book` objects to `Author` objects, thus for each author a list of his books may be stored and vice versa. Figure 4.40 depicts the general notation of this element and also provides a usage example.

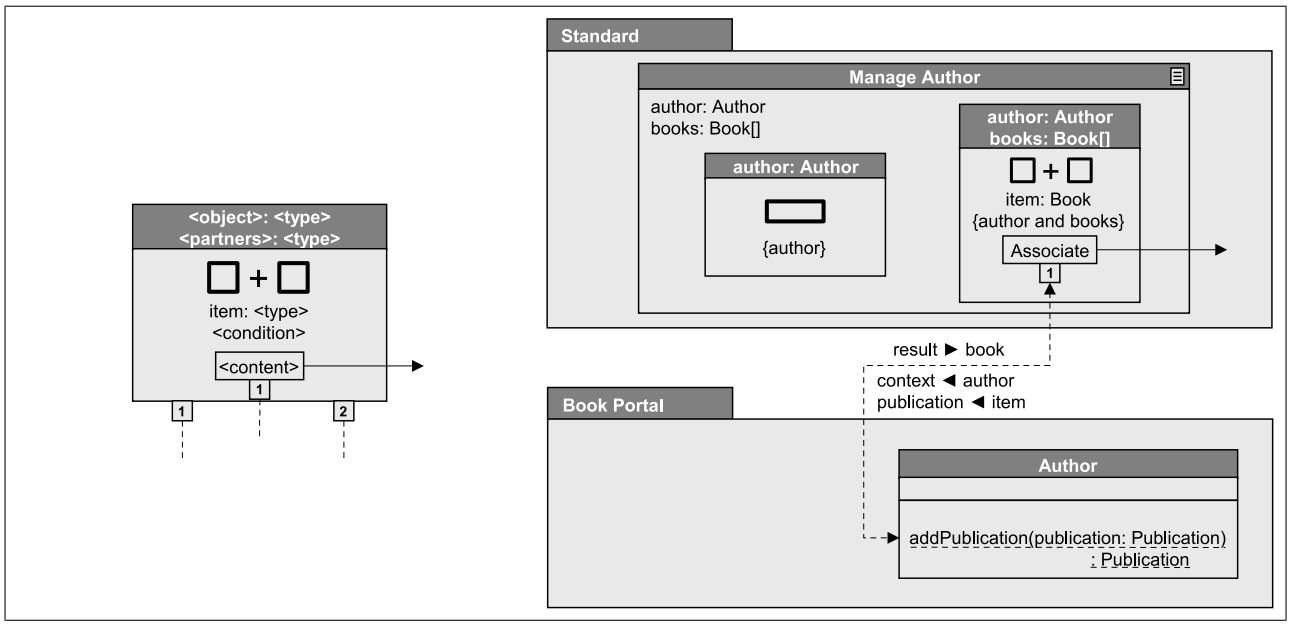

Figure 4.40: The `AssociationCreator` Element

The left-hand side of the figure shows the general notation of the `AssociationCreator` element. The element's header contains two context variables. The first one specifies an object for that an association is to be created. The second variable specifies a list of possible association partners. Correspondingly, this element possesses two operation ports that may be connected to operations, which deliver appropriate context objects.

The element's body section contains an icon (two rectangles with a plus sign between them), the *item* variable, an optional condition, and an `ActionLink` sub-element. The task of the *item* variable is to store an association partner that is selected by the Web application's user. It may be used as context or a parameter for an appropriate operation. The operation port of the `ActionLink` sub-element may be connected to an operation that creates the association

between the target objects. Its navigation link may be connected to a follow-up page that is presented after the execution of the operation. A possible rendering of the `AssociationCreator` element presents a short info about the context object and a form that contains a selection field of possible association partners and a field for each describing attribute of the association. The `ActionLink` sub-element is usually rendered as a button, for example, bearing the label "Associate".

The right-hand side of Figure 4.40 shows an example of how the `AssociationCreator` element may be used. The *Manage Author* page provides the functionality for associating `Author` and `Book` objects. To this end, the page contains the *author* and *books* variables. The origin of values for these variables is not important for this example. The `Author` object that is to be associated to a `Book` object is presented with a simple `ObjectView` element on the left-hand side. The `AssociationCreator` element on the right-hand side of the page uses the *author* and the *books* variables as context and also specifies a condition that checks whether these variables have valid values. The operation port of the `ActionLink` sub-element is connected to the *addPublication* operation of the `Author` operation class. The *author* variable is provided as context and the *item* variable, which holds the selected association partner, i.e., a selected `Book` object, is mapped to the *publication* parameter of the operation. The navigation link of the element points to a follow-up page, which is omitted in this example.

Note that the content model of the Book Portal example application (see Figure 4.3 in Section 4.3) specifies the `Book` class as a specialization of the `Publication` class, thus the corresponding association-level operation in the `Author` operation class is called *addPublication* and requires a parameter of the `Publication` type. Of course, any instances of a sub-class, e.g., `Book` instances, are also allowed as valid parameters.

4.5.2.18 Association Modifier

The previous section introduced a model element that supports the association of two objects. Of course in many cases, an association between two objects is described by one or more attributes. For example, the `Comment` association between the `Review` and `Publication` classes of the Book Portal application is described by the *CommentDate* attribute (see Figure 4.3 in Section 4.3). In some cases, it is necessary to update attributes of an association without having to redefine it, i.e., it should not be necessary to delete and recreate the association. To this end, the Operation Model defines for each association an appropriate *change* operation. The corresponding element of the Composition/Navigation Model that integrates this operation into the Web application's user interface is the `Association Modifier` element shown in Figure 4.41. A typical rendering for this element is a Web form that provides fields for each attribute of the association allowing the user to provide new values and issue the modification.

The left-hand side of the figure shows the general notation of the `AssociationModifier` element. The element's header contains a first context variable that specifies an object for that an association attribute is to be modified. The second context variable may contain the association partner. Accordingly, this element possesses two operation ports, which may be connected to operations delivering the context objects.

Note that the second context variable and also the second operation port are optional. It depends on the multiplicity of the association partner whether its specification is necessary or not. For example, an author may write several books. Thus, if the `Write` association had any describing attributes, the corresponding *change* operation of the `Author` operation class

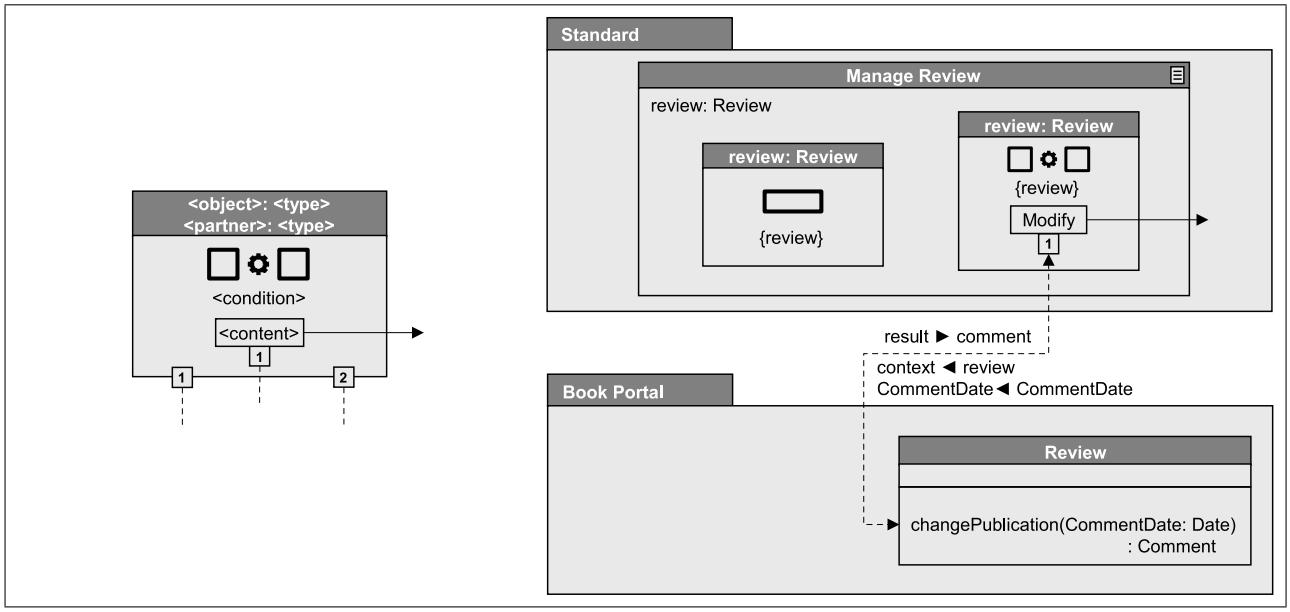

Figure 4.41: The `AssociationModifier` Element

and also any referring `AssociationModifier` elements had to provide a `Book` object as association partner in order to make clear to which book instance the modified association refers to. In contrast to that, if an object has only a single association partner, e.g., a review always refers to a single publication, it is not necessary to specify the association partner.

The element's body section contains an icon (two rectangles and a cogwheel), an optional condition, and finally an `ActionLink` sub-element. Similarly to other model elements that provide values as input for an operation, the `AssociationModifier` element includes an `ActionLink` sub-element. The operation port of this sub-element may be connected to an operation that modifies the attributes of an association between two objects. The element's navigation link may be connected to a follow-up page that is presented after the execution of the corresponding operation.

The right-hand side of Figure 4.41 shows the usage of the `AssociationModifier` element. This example demonstrates the modification of the *CommentDate* attribute of the `Comment` association between a `Review` and a `Book` object. To this end, the *Manage Review* page contains the *review* variable, which provides a `Review` object for that an association is to be modified. The origin of a value for this variable is not relevant for this example. An `ObjectView` element on the left-hand side of the page shows some supplemental information about the review. The focus of this example is on the `AssociationModifier` element on the right-hand side of the page. This element uses the *review* variable as context and the operation connector of the `ActionLink` sub-element is connected to the *changePublication* operation of the `Review` operation class. The *review* variable is provided to the operation as context along with the single *CommentDate* parameter. Note that there is no need to define this parameter explicitly as a variable in the element's namespace. Finally, the navigation link of the element targets a page that is omitted from this example for the sake of simplicity.

4.5.2.19 Association Remover

The `AssociationRemover` is the last model element that deals with associations between objects. As the name suggests, it specifies a user interface component that allows the Web application's user to delete an association between two objects. This is a common functionality in any content management system. Assume, for example, that an administrator of the Book Portal example application registers that he has associated the wrong author to a specific book. In this case, the false association has to be removed from the system and the correct one has to be created. A possible rendering for this element is a Web form that lists partner objects that have been already associated to the target object. The form allows the selection of one association partner that is to be removed from the list of associated objects and the initiation of the corresponding operation. Figure 4.42 depicts the general notation of this element and provides a usage example.

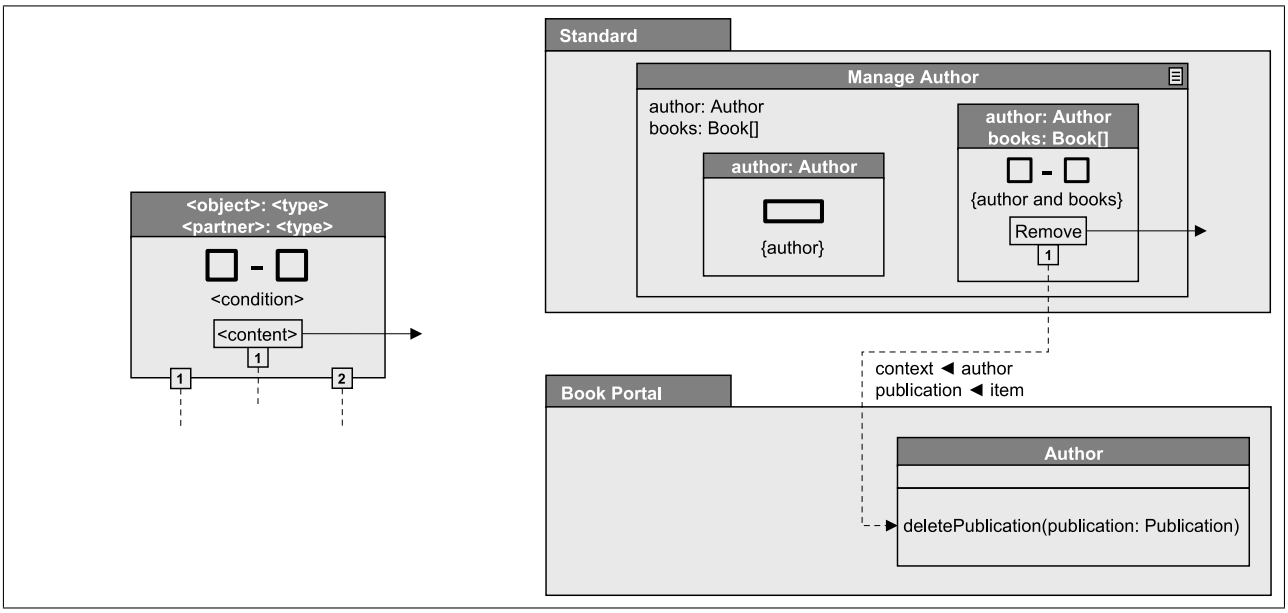

Figure 4.42: The `AssociationRemover` Element

The left-hand side of the figure shows the general notation of the `AssociationRemover` element. The element's header contains two context variables. The first one specifies an object for that an association is to be removed. The second one specifies a list of already associated partner objects. This model element possesses two operation ports that may be connected to operations, which deliver values for the context variables.

The body section of the element contains an icon (two rectangles with a minus sign between them), an optional condition, and an `ActionLink` sub-element. This sub-element's operation port may be connected to an operation that deletes an association between two objects. Finally, the navigation link may be connected to a page that is to be presented after the operation was executed.

The right-hand side of Figure 4.42 presents an example for the `AssociationRemover` element. The example shows how this element may be used to remove a *Write* association between an `Author` and a `Book` object. This is achieved by the *Manage Author* page that contains the *author* and the *books* variables. The origin of values for these variables is not of concern for this example, however, it is assumed that the *books* variable contains `Book` objects that are

already associated to the Author object in the *author* variable regarding the Write association. An ObjectView element on the left-hand side of the page presents the Author object for that an association is to be deleted. The AssociationRemover element on the right-hand side of the page uses the *author* and *books* variables as context, contains a simple condition that checks whether these variables have valid values, and connects its ActionLink sub-element to the *deletePublication* operation of the Author operation class. The *author* variable is provided as context for the operation and the special *item* variable that holds a selected Book object is provided as a parameter. The navigation link of the ActionLink sub-element points to a page that is omitted from this example.

Note that similarly to the AssociationModifier element, the multiplicity of association partners for an object also affects the usage of the AssociationRemover element. However, the implications are somewhat different. If an object is allowed to have multiple association partners regarding a given association, then the AssociationRemover element may be used as previously described in this section. However, if an object is only allowed to have a single association partner, then it is not necessary to use the AssociationRemover element to delete the association. In such a case, the corresponding *delete* operation does not require any parameters because the association partner is unambiguous. Thus, the usage of a user interface component that allows the selection of a potential association partner, for which the association is to be deleted, is unnecessary. In case of a single association partner, it suffices to use a ActionLink element that is connected to the corresponding delete operation.

4.5.2.20 Custom Item

Web applications have evolved from simple Web sites into complex systems that often have a similar complexity level to traditional software applications. Correspondingly, many Web applications possess comprehensive user interfaces and any model-based Web engineering method that aims to support the development of such complex applications has to provide a powerful set of model elements for user interface design. However, it is not reasonable to extend the model with one or more new model elements every time a problem cannot be solved with the existing set of elements. The flashWeb approach provides a powerful set of standard model elements for user interface design that have been introduced in previous sections. These elements may be combined in a flexible way to match the largest part of any user interface specification. User interface requirements that cannot be fulfilled with this set of elements are supported with the CustomItem element. It allows the specification of custom code utilizing (programming) languages of the target platform the Web application is deployed to. For example, if the target platform is Java EE the Custom Item element may be used to specify HTML and JSP instructions that solve the given problem. Similarly to the Custom Operation (see Section 4.4.2.7) of the Operation Model, this element may be used to insert custom code into the model, i.e., into the user interface specification. Figure 4.43 depicts the general notation of the CustomItem element and provides a usage example.

The left-hand side of the figure shows the general notation of the CustomItem element. The element's header section specifies a name, the body section contains an icon depicting a question mark, an arbitrary number of variables, and an optional condition. As many other elements of the Composition/Navigation Model, this element may have an arbitrary number of operation ports, thus it may be connected to an arbitrary number of operations that may provide appropriate input values. Note that the custom code, which is provided by the ap-

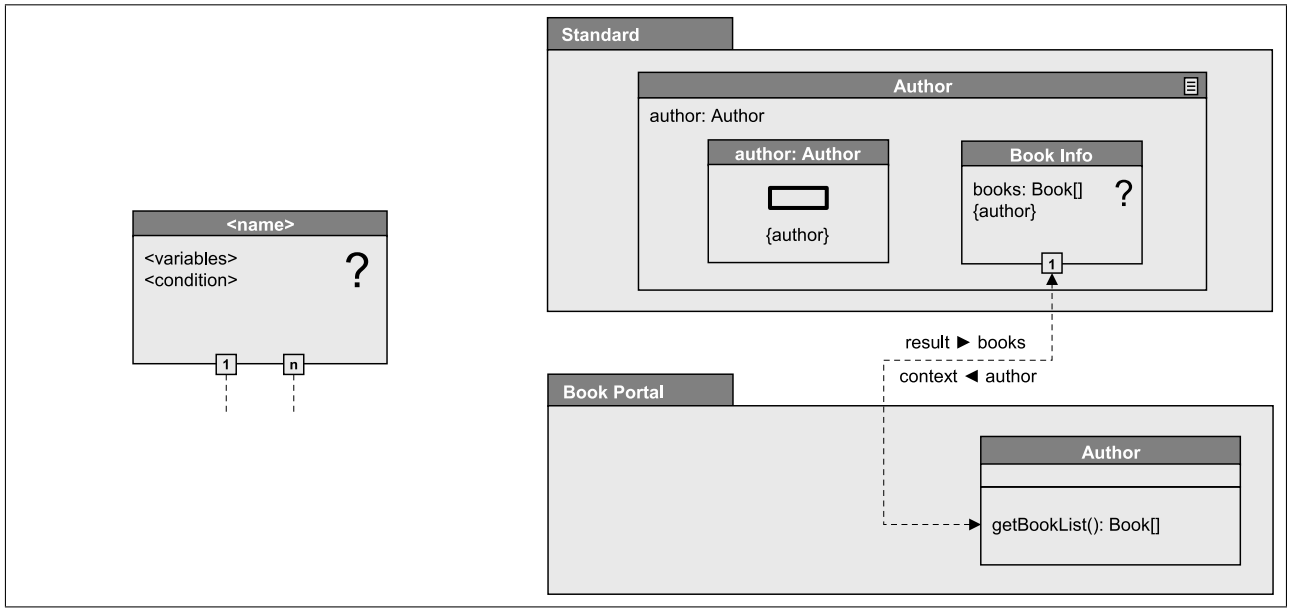

Figure 4.43: The `CustomItem` Element

plication developer, is not displayed explicitly. Of course, any CAWE tool that supports the flashWeb approach has to provide an appropriate way in order to allow the developer to attach the custom code to instances of this model element.

The usage example for the `CustomItem` element on the right-hand side of Figure 4.43 shows the *Author* page, including an `ObjectView` element to show author data and a `CustomItem` element that provides some additional information about the author's books. The `CustomItem` element has a single operation port that is connected to the *getBookList* operation of the `Author` operation class. The operation receives the author object as context and delivers a list of the author's books, which is mapped to the *books* variable of the `CustomItem` element.

The additional information about the author's books on the *Author* page should include the total number of books and the title of three books. To this end, the `CustomItem` element utilizes the code provided in Listing 4.5.

```
1  The portal has information about
2  <span tal:content="python:len(books)" />
3  books of this author.
4
5  Selected books:
6  <ul>
7   <li tal:repeat="book books[0:3]"
8       tal:content="python: book.getTitle()">
9   </li>
10 </ul>
```

Listing 4.5: Custom User Interface Code

The exemplary code that is shown in this listing may be used for the Zope [vW08] Web application framework. The example includes simple text and additionally utilizes HTML, TAL

and Python. The Template Attribute Language (TAL) of the Zope framework is introduced in Section 5.3 in more detail. However, the current example may be understood without any deeper knowledge of TAL. Line 2 shows the number of books that are stored in the *books* variable. To this end, the *len* Python method computes the length of the book list and the *tal:content* expression inserts the result as the content of the *span* HTML element. The lines 6-10 specify a simple listing of three book titles. To this end, the *tal:repeat* expression in line 7 specifies that the *li* HTML element should be repeated three times, once for each of the first three books of the book list. Finally, in line 8 the *tal:content* expression specifies that the *getTitle* method is to be called in order to insert the title of a book as the content of the *li* HTML element. Note that the presented code is fault tolerant and works also if an author has less then three books.

4.5.3 Content Management Patterns

Web applications that facilitate the management of non-trivial content usually support a variety of common content management patterns. Most of these patterns are seamlessly integrated into Web application interfaces and provide an improved user experience without getting actually recognized by the Web application's user. Perhaps the most basic but also the most important pattern is the presentation and management of one or more content objects in the context of another content object. Regarding the Book Portal application, a simple example for context specific presentation is showing all reviews of a user (see Section 2.2.3.3 and Section 4.3). An example for context-specific content management is creating, modifying, or deleting a review for a selected user. In all of these examples, the context is a user object.

Subsequently, the content management pattern of creating a new object in the context of another object is presented. The example in Figure 4.44 uses the `Review` and the `User` objects of the Book Portal application to demonstrate how easily content management patterns may be defined with the flashWeb mehtod.

The specification of a content management pattern includes two steps. The first step is to create an appropriate composite operation (see Section 4.4.2.2) that provides the required content management functionality at the operation level. To this end, in Figure 4.44 the `Book Portal` operation package contains the `Review` operation class, which defines the *createReviewForProvider* composite operation. This composite operation combines the *createReview* and *addReviewProvider* operations in order to create a new `Review` object and to associate it to an existing `User` object in a single step. Note that composite operations and this specific example are explained in detail in Section 4.4.2.2.

The second step of creating a content management pattern is the integration of the created composite operation into the user interface specification. To this end, the `Standard` user profile in Figure 4.44 includes the *User Reviews* page that provides the corresponding functionality. The page defines the *user* variable and contains two elements. Note that the origin of a value for the *user* variable is not relevant for this example. The first element of the page is an `ObjectView` element (see Section 4.5.2.8) on the left-hand side. It uses the *user* variable in order to present information about the `User` object, which plays the role of the context object for this example. The right-hand side of the page contains an `ObjectCreator` element, which actually integrates the *createReviewForProvider* operation into the user interface. To this end, the operation port of this element is connected to the operation and the connection provides all necessary mappings. The directly by the `ObjectCreator` element and the *user* variable is mapped to the *ReviewProvider* parameter of the operation. Finally, the result of the operation is

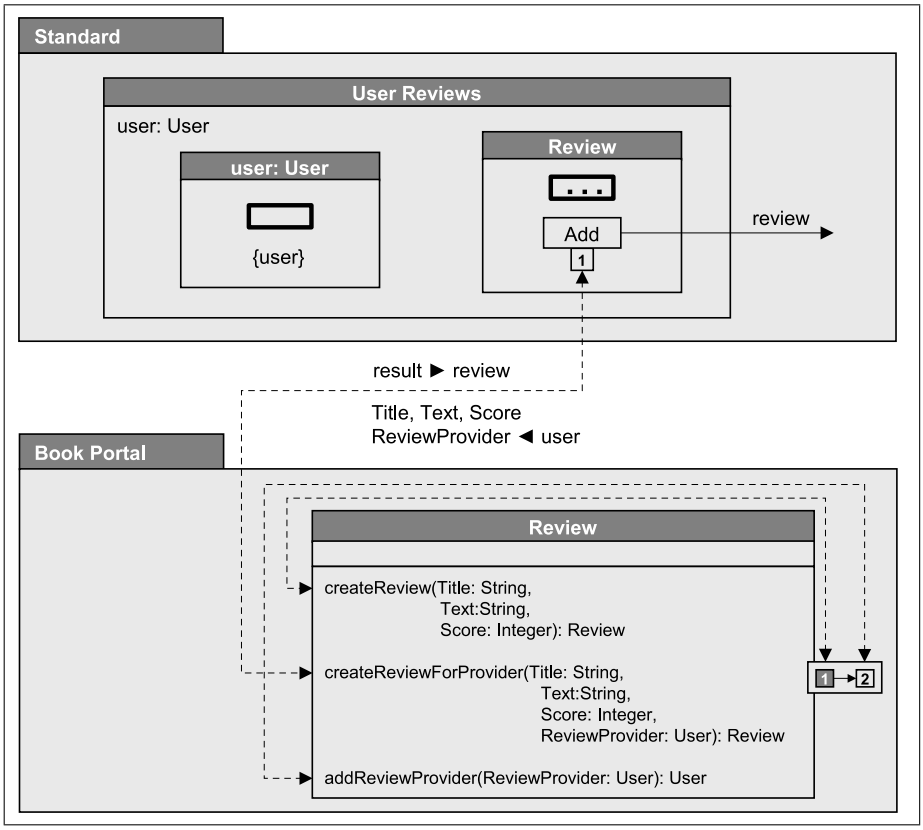

Figure 4.44: Example for the *Create in Context* content management pattern

mapped to the *review* variable and forwarded by the navigation link of the `ObjectCreator` element to a follow-up page that is omitted from this example for the sake of simplicity.

Note that this section provides merely a simple example of a content management pattern, which combines only a few model elements. Of course, the flexible nature of the flashWeb method allows for the composition of patterns with arbitrary complexity. This is due to the fact that composite and custom operations of the Operation Model (see Sections 4.4.2.2 and 4.4.2.7) are able to express arbitrary application logic and that elements of the Composition/-Navigation Model may be composed in a similar flexible manner to provide appropriate user interface components for these operations.

4.5.4 Notation Variations

Similarly to the Content Model and the Operation Model (see Sections 4.3 and 4.4) of the flash-Web Web engineering method, the Composition/Navigation Model also provides different alternative notations for all of its elements. The aim of these alternative notations is to hide details of an element that are not of concern to the Web application developer at a specific point in time during his modeling activity. A flashWeb CAWE tool that supports these simplified notations allows the Web application developer to focus on a certain part of the user interface without being overwhelmed by unnecessary details of other parts.

Basically, alternative notations for the Composition/Navigation Model offer two different ways to abstract from details. First, any model element may be *minimized*, thus only a minimal representation of the element is displayed. The minimized notation of an element hides the

element's body section entirely except for an icon. This is of course a major reduction of information, especially for elements that have subelements, as these are hidden as well. Second, operation calls of a Composition/Navigation Model element may be hidden explicitly. In this case, all connections of a given model element to any operation of the Operation Model are hidden.

Similarly to the simplified notations of other flashWeb models, all simplifications are indicated by icons, which are displayed in the header section of a given model element. A minimized element is indicated by an *arrow* icon and an element, for which operation calls have been hidden, bears a small circle with the letter "O" inside it. Figure 4.45 depicts an example using the Book Portal application scenario.

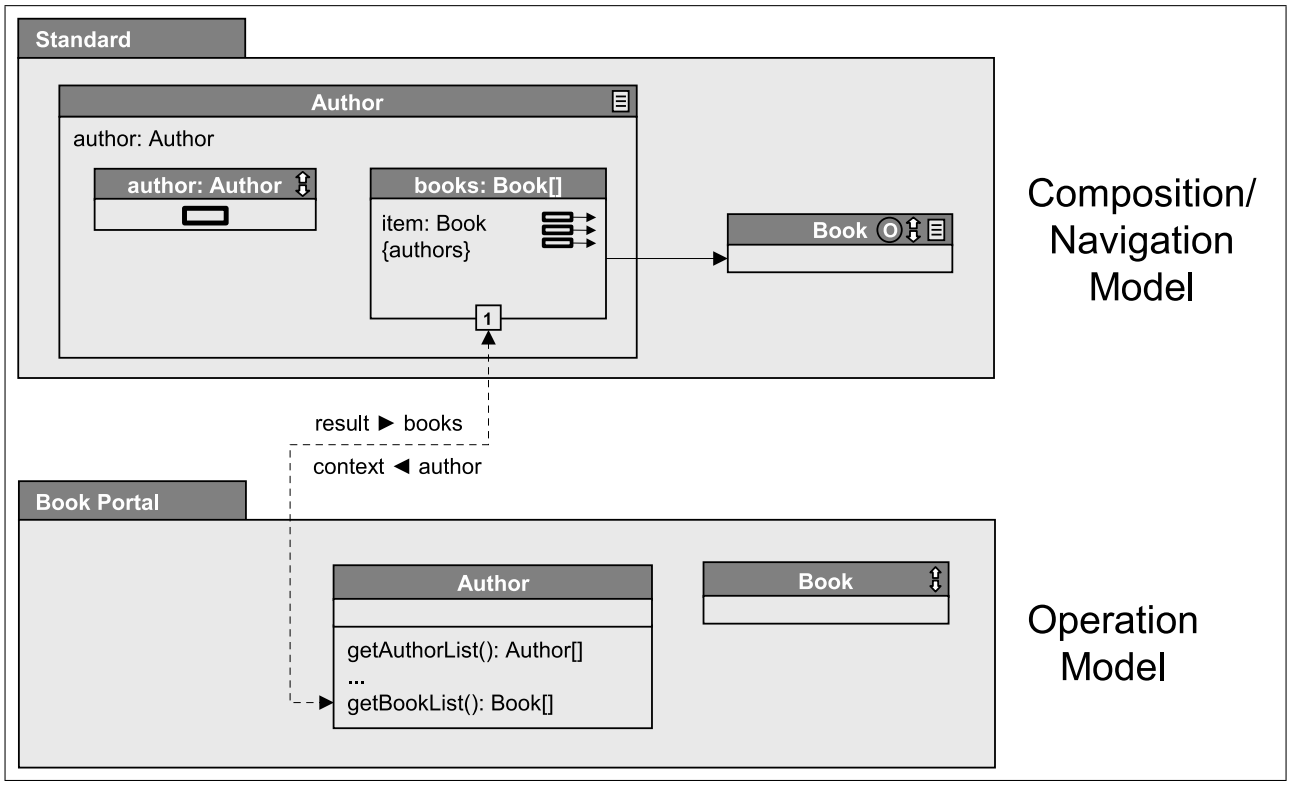

Figure 4.45: Minimized `Book` Page Example

This example specifies a part of the Book Portal application that shows information about authors and their books and also allows the Web application's user to request additional information about a selected book. It is used for a step-by-step demonstration of alternative graphical notations for included model elements. The first configuration depicted in Figure 4.45 specifies the *Author* and the *Book* pages for the Composition/Navigation Model and the `Author` and `Book` classes for the Operation Model. The `Book` page is already minimized as it is indicated by the *arrows* symbol in the element's header. Additionally, all operation calls that originate from this page are hidden as well, which is indicated by the *O*-symbol in the header of the page. Of course, one may assume that one or more hidden operation calls target the `Book` operation class, which itself is minimized.

The left-hand side of the *Author* page contains an `ObjectView` element. This element is minimized, thus the body section shows only a small icon. However, as the minimized notation of an element still displays the header, it is clear that this element refers to the *author* variable

from the namespace of the *Author* page in order to provide information about an `Author` object. The right-hand side of the page contains an `ObjectIndex` element that is represented with its standard, non-minimized notation. This index receives its data from the *getBookList* operation of the `Author` operation class. To this end, the elements operation port is connected to the operation and it provides the *author* variable as context. The result of the operation, a list of books, is provided to the index, which is indicated in the element's header.

However, the example presented so far still contains some elements that may be further simpified. Figure 4.46 presents the same scenario with two other elements using simplified representations.

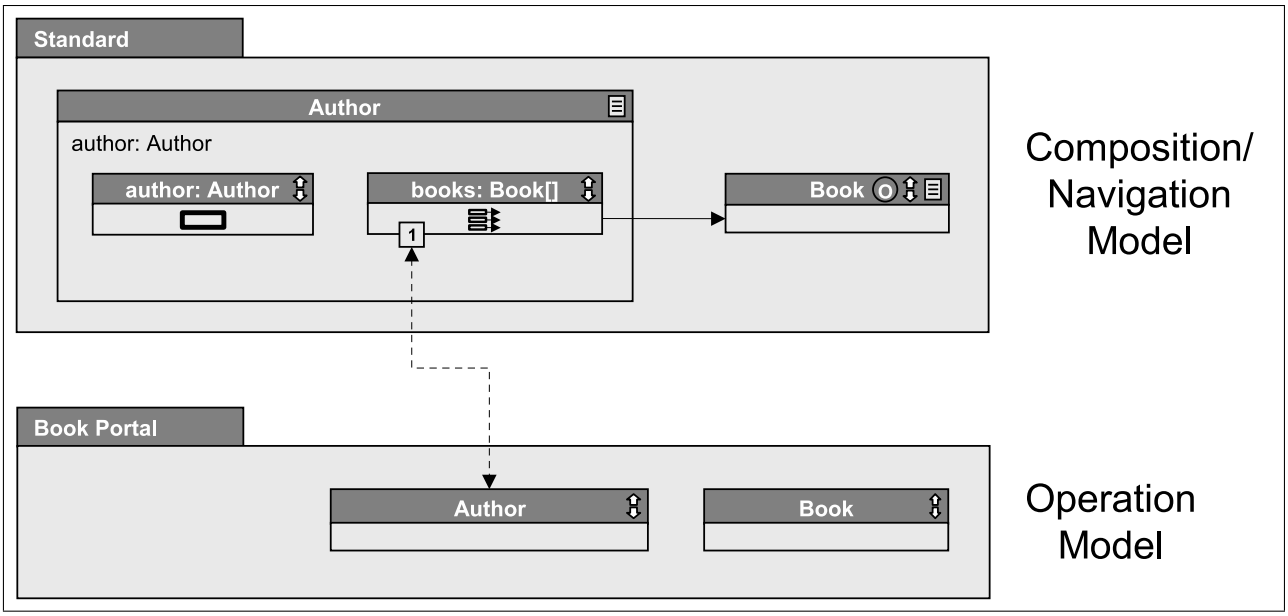

Figure 4.46: Minimized `ObjectIndex` Example

The first element that is represented with a minimized notation in contrast to the specifications in Figure 4.45, is the `ObjectIndex` element on the right-hand side of the *Author* page. The single content of the element's body section is a small icon. The header section displays the *arrow* symbol indicating that the element is minimized. Note however that the element's header does not contain the *O*-symbol and that correspondingly the operation call targeting the `Author` operation class is still displayed.

The second element that differs compared to Figure 4.45 is the `Author` operation class. It is minimized, i.e., all operation signatures of the class are hidden, which is indicated by the *arrow* symbol in the header section of the class. Notice however that the operation call that originates from the `ObjectIndex` element is now connected to the header of the `Author` operation class. This is due to the fact that all operation signatures are hidden and that the operation call cannot be associated to the right operation. Also note that the operation call does not specify any mappings between variables and operation parameters because the information is not relevant any more at this level of abstraction. Finally, Figure 4.47 depicts an almost minimal representation of the example scenario.

This representation additionally hides the content of the *Author* page. This fact is additionally indicated by the *arrow* symbol in the header section of the page. Note however that the operation call between the *Author* page and the `Author` operation class is still displayed. Of

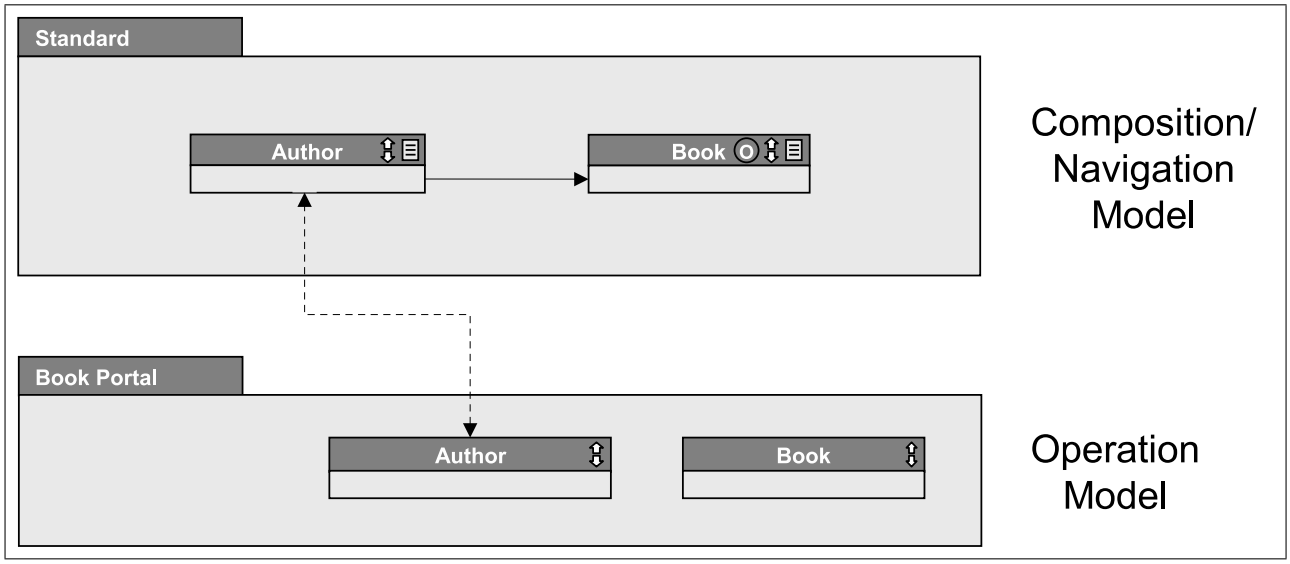

Figure 4.47: Minimized `Author` Page Example

course, the operation call is not connected to an operation port of the page because that would represent different semantics. Connecting the operation call just to the bottom of the page indicates that the operation call originates from a hidden sub-element of the page. Note that packages and user profiles also have simplified notations. Thus a final abstraction step for this example could be the minimized representation of the *Standard* user profile and the *Book Portal* operation package.

4.5.5 Summary

The Composition/Navigation Model that has been introduced in the previous sections allows the Web application developer to design the Web application's user interface in an effective and efficient manner. To this end, it provides model elements that may be combined freely to build an arbitrary user interface. The fine granularity of model elements ensures that an overwhelming part of the user interface may be constructed using standard elements and that only a minimal part has to be customized.

The explicit distinction between operations (Operation Model) and user interface components (Composition/Navigation Model) of a Web application has many advantages. First, the separation of concerns ensures a clear design and the reusability of components. Second, the Web application may be developed in a parallel manner. A developer that focuses on content management functionality may specify a set of content management operations and a user interface expert may concentrate on the development of the user interface. The generic nature of elements of the Composition/Navigation Model allows the smooth integration of content management operations into the user interface. Finally, the navigation structure of the Web application is defined in a single model and is not intermixed with other concepts, e.g., with content management patterns.

Unfortunately, most Web engineering methods neglect to define a powerful model for user interface design that is precise enough for the generation of the actual user interface. The only exception is the WebML approach that has been presented in Section 3.4.4. However, the WebML integrates basic content management operations into the user interface specifica-

tion and ,in contrast to flashWeb, it does not offer the advantages that have been mentioned previously.

4.6 Presentation Model

The aim of the Composition/Navigation Model that has been introduced in the previous section is to define the user interface of a Web application. However, as the name of the model suggests, its focus is on the composition of user interface pages and on the definition of the Web application's navigation structure. It does not define the actual appearance of each user interface component. The separation of content and presentation is an important paradigm that should be followed by any design method that aims to support the development of content-intensive Web applications. Following this paradigm, flashWeb's Composition/Navigation Model specifies what parts of the Web application's content are to be presented on which user interface page. As an addition to that, flashWeb's Presentation Model assigns to each user interface component an appropriate presentation. Figure 4.48 illustrates this approach by providing a simple example using the Book Portal scenario.

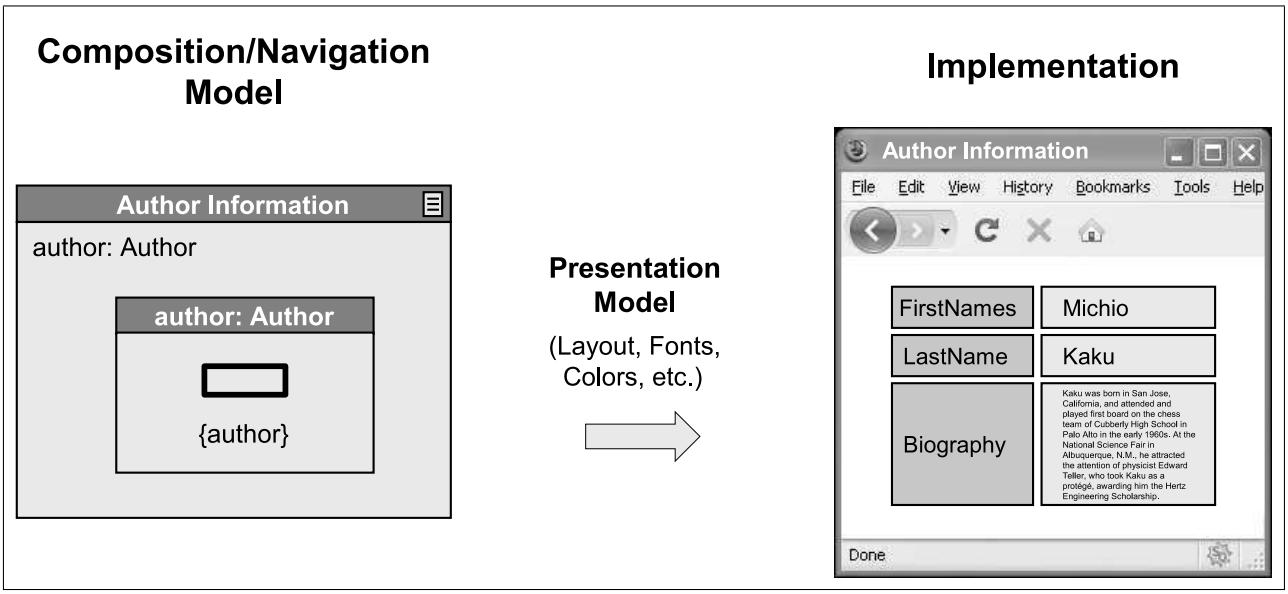

Figure 4.48: Motivation for the Presentation Model

The left-hand side of the figure depicts a partial Composition/Navigation Model. The *Author Information* page contains a single `Object View` sub-element that specifies a user interface component, which provides information about an `Author` object. However, this partial model does not specify how the corresponding information is to be presented. The right hand side of the figure shows a possible rendering for this model, which is a Web page with the title *Author Information* and a simple two-column table showing attribute names and corresponding attribute values for a single author. Of course, even for such a simple example, it is a long way from an abstract model to the actual presentation of a user interface component. It is the task of a presentation model to define all the necessary information to fill in this gap. Some aspects that have to be specified are the layout of all components, e.g., the presentation of an author object with a table-like structure, the positioning of components regarding their parents, e.g.,

the positioning of the author information inside the page, the specification of text fonts, text colors, background colors, etc. For the flashWeb approach, all this information is provided by the Presentation Model that relies on a few concepts, which are introduced in Figure 4.49.

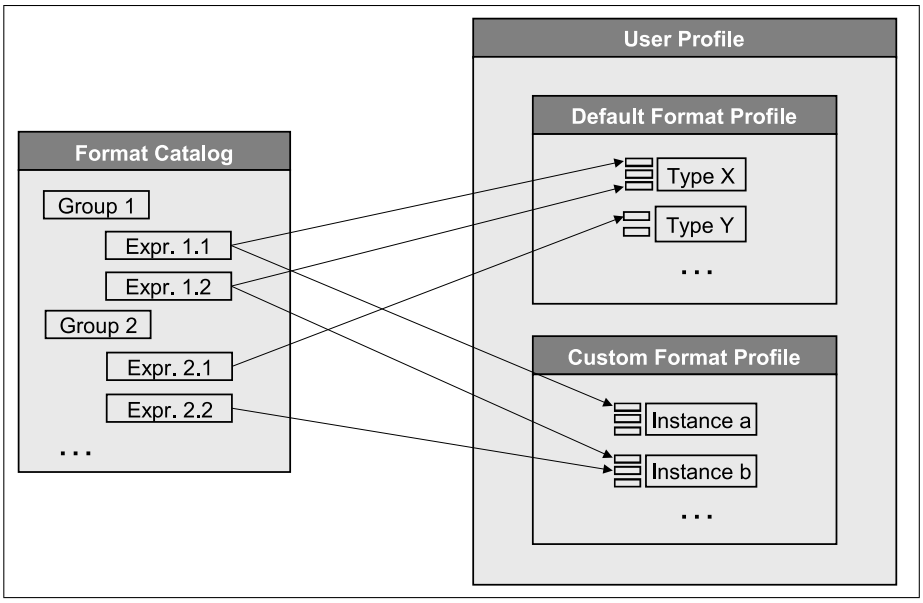

Figure 4.49: Presentation Model Concepts

Note that flashWeb's Presentation Model does not employ an explicit graphical notation like all the other models presented so far. It is rather a set of concepts that are to be implemented by a CAWE tool, which aims to support the flashWeb method. Consequently, the presentation in Figure 4.49 is not a graphical notation but merely an illustration of these concepts.

The Presentation Model relies on four basic concepts. A `Formatting Expression` is a piece of code that specifies a specific aspect of a model element's presentation utilizing an appropriate formatting language. The `FormatCatalog` is a hierarchical collection of formatting expressions. Note that in Figure 4.49 the format catalog has two groups, both of them containing two formatting expressions. The `DefaultFormatProfile` specifies a common presentation for all instances of a specific model element for a single user profile. Finally, the `CustomFormatProfile` defines the presentation of specific model element instances for a user profile. The following sub-sections describe each of these concepts in detail.

4.6.1 Formatting Expressions

A `FormattingExpression` abstracts from a concrete formatting language that may be utilized to define the presentation of a Web application for a specific target platform. For example, most Web application frameworks employ Cascading Stylesheets (CSS) for defining the final presentation of the user interface. This technology allows to define the presentation of a user interface component in a fine-granular manner but, of course, the language has its own syntax that has to be followed. The motivation for the concept of a formatting expression is to abstract from a specific formatting language and from the granularity of formatting constructs of the given language. To this end, a formatting expression is composed of two parts. The first part is a *name* that identifies the expression and the second part is a *code fragment* that contains the actual formatting directives in the target language. A simple example of a formatting example

may have the name “Small Red Text”. A corresponding code fragment using CSS is provided in Listing 4.6.

```
1 font-family: 'Palatino Linotype',Times,serif;
2 font-style: normal;
3 font-weight: normal;
4 font-size: 8pt;
5 color: red;
```

Listing 4.6: Example Code for a FormattingExpression

The listing contains five CSS formatting directives. The first line states that the preferred font family is “Palatino Linotype” and also defines some alternatives if this font is not available. The second and third lines state that the font style and font weight should be normal. The forth line provides a directive that specifies the font size, which is set to eight point in this example. Finally, the last line specifies the font color, which is red. Note that this is only a subset of formatting directives that are offered by CSS for formatting text. Thus in order to abstract from these directives, this code is assigned the name “Small Red Text” and may be used in format profiles that are introduced in subsequent sections.

4.6.2 Format Catalog

The aim of the `FormatCatalog` is to gather formatting expressions in a structured way. To this end, the format catalog employs the notion of a group that may contain an arbitrary number of formatting expressions or sub-groups. However, formatting expressions and sub-groups are not allowed to be mixed, thus a group either contains one or more expressions or one or more sub-groups. For example, “Text Formats” would be an appropriate name for a group that contains the “Small Red Text” formatting expression.

The general idea of the format catalog is for the user interface developer to create a structured repository of formatting expressions, which may be used to format different parts of the user interface in a simple an flexible way. The concept of a catalog of formatting expressions that abstracts from the actual syntax of the formatting language greatly facilitates parallel development. An art designer may create the format catalog independently from the actual user interface of a particular Web application. Simultaneously, a user interface designer may compose the user interface and, after the end of these activities, formatting expression may be assigned to elements of the user interface model as required. Of course, the art designer and the user interface designer should synchronize their efforts.

4.6.3 Default Format Profile

The `DefaultFormatProfile` allows to define a default presentation for each model element type of a certain user profile. A user profile, i.e., definitions that are contained in a `UserProfile` model element (see Section 4.5.2.1) define the user interface of the Web application for a certain group of users. Of course, it should be possible to format different user profiles differently. Therefore, every user profile has its own default format profile.

The default format profile assigns to every model element type a *presentation type* and a corresponding set of *formatting slots*. A formatting slot represents a certain part of a user interface

component's presentation, e.g., the formatting of a row in a tabular view. Each formatting slot of a presentation type may be associated to an arbitrary number of formatting expressions from the format catalog. Figure 4.50 shows an example of a partial default format profile and the association of formatting expressions to formatting slots.

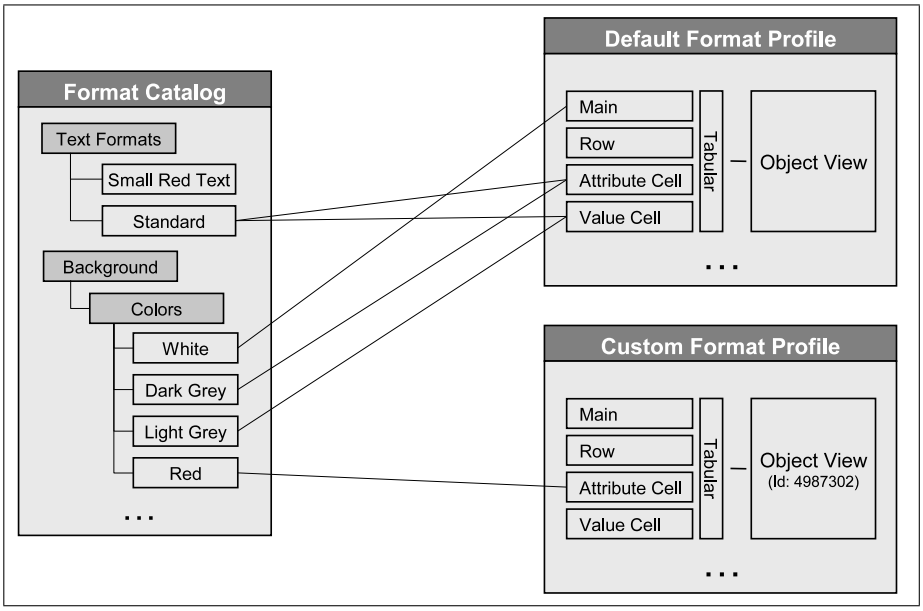

Figure 4.50: Format Profile Examples

The right-hand side of the figure illustrates a default format profile with a single formatting assignment. The `ObjectView` model element type is assigned the *Tabular* presentation type, e.g., every instance of this model type is to be presented with a tabular view as default. This tabular presentation type contains four formatting slots. The *Main* slot defines the apperance of the entire view. For example, this slot may be used to specify the position of the component inside its parent component. The current example in Figure 4.50 assigns a white background color to the entire component. To this end, the slot is connected to a corresponding formatting expression of the format catalog. The *Row* slot defines the formatting of rows of the tabular view. A possible formatting option for this slot is, for example, the definition of a distance between rows. However, the current example does not assign any formatting expressions to this slot. Finally, the *Attribute Cell* and *Value Cell* slots define the presentation of table cells that hold attribute names and attribute values, respectively. The current example assigns a standard text format to both slots, a dark grey background color to an attribute cell and a light grey background color to a value cell. Note that these formatting definitions may result in a presentation that is similar to that shown on the right-hand side of Figure 4.48.

A default format profile may contain for every model element type an appropriate formatting assignment, thus every user interface component of a corresponding user profile is represented in a fashionable manner. Of course, every model element type may be assigned only one formatting type from a list that is defined by the default format profile.

4.6.4 Custom Format Profile

The user interface of a modern Web application has to be tailored to the needs of its users. Important data and functionality has to be highlighted and made easily accessible, whereas

less important parts do not have to be emphasized. Accordingly, it is a common requirement that the presentation of selected user interface components may differ from the presentation of other components of the same type. However, the default format profile that has been introduced in the previous section only allows to specify a common presentation for all instances of a certain model element type. Therefore, the flashWeb method employs the `CustomFormatProfile` that facilitates the custom formatting of selected model elements.

The custom format profile relies on exactly the same concepts as the default format profile. However, instead of defining the presentation of every model element of a certain type, it does so for selected model element instances. The right-hand side of Figure 4.50 illustrates a custom format profile that defines the presentation of a single `Object View` model element. In this example, the `Attribute Cell` slot of the `Tabular` presentation type is assigned a red background color. Accordingly, the attribute names of the corresponding model element instance are highlighted with red color, whereas all other instances of this type have grey backgrounds for attribute names.

Note that the presentation of every model element is determined by formatting assignments of the corresponding default format profile and possibly from a custom format profile. Naturally, assignments of the custom profile overwrite assignments of a default profile. The granularity for overwriting formatting assignments is the slot level. Thus, if both the default and the custom profiles assign formatting expressions to a certain formatting slot, the definitions of the custom profile are used. However, if the custom profile does not define the formatting of a certain slot, but the default profile does, the assignments of the default profile are used.

4.6.5 Summary

In contrast to other flashWeb models, the Presentation Model does not employ a graphical notation that defines the appearance of user interface components. Instead, it is a set of concepts that are to be implemented by a CAWE tool, which supports the flashWeb development method. Note that such a tool is presented in Section 5.2.

The focus of flashWeb is clearly on the design of content, data management operations, and the composition of the user interface. The development of a graphical model that also defines the presentation of user interface components is future work. However, as the Composition/-Navigation Model already provides a presentation-near notation, it is advisable to design a graphical presentation model that is derived from it.

CHAPTER 5

Implementing Web Applications with flashWeb

The previous chapter introduced the flashWeb method that utilizes different graphical models to capture the entire functionality of a Web application. Different models may be used to design the Web application's content storage, content management functionality, and its user interface. From the developer's point of view, flashWeb's graphical models offer various benefits. They support a structured development process and serve as an excellent documentation of the Web application. However, the real advantage of the method comes from its interconnected models that allow the generation of a fully functional implementation. This code generation capability speeds up the development process of any Web application considerably. Note that Section 3.2.6 gives an overview of different code generation scenarios and explains their advantages for the Web application development process. The aim of this section is to present a set of technologies that have been developed to support the flashWeb Web engineering method. To this end, Section 5.1 outlines the overall implementation strategy, Section 5.2 describes the flashWeb CAWE tool, Section 5.3 introduces the Zope 3 implementation framework, and finally Section 5.4 provides details about the code generation process.

5.1 Implementation Strategy

The general strategy of the flashWeb Web engineering method for developing a Web application includes three main components. First, the developer uses the flashWeb CAWE tool, which is a graphical model editor to create all four flashWeb models that specify the complete functionality of the Web application. Second, a suitable implementation framework is utilized that usually provides a set of common features and a runtime environment. Finally, a code generator plug-in of the flashWeb CAWE tool uses the models to generate an implementation for the framework. Figure 5.1 depicts these components.

Of course, the CAWE tool and the code generator plug-in are both sophisticated pieces of software. The CAWE tool has to provide an intuitive and effective user interface that allows to create and manage the graphical models. Additionally, the code generator has to be able to transform the models into application code that fits the requirements of the implementation framework. To provide the maximal amount of flexibility, the architecture of the CAWE

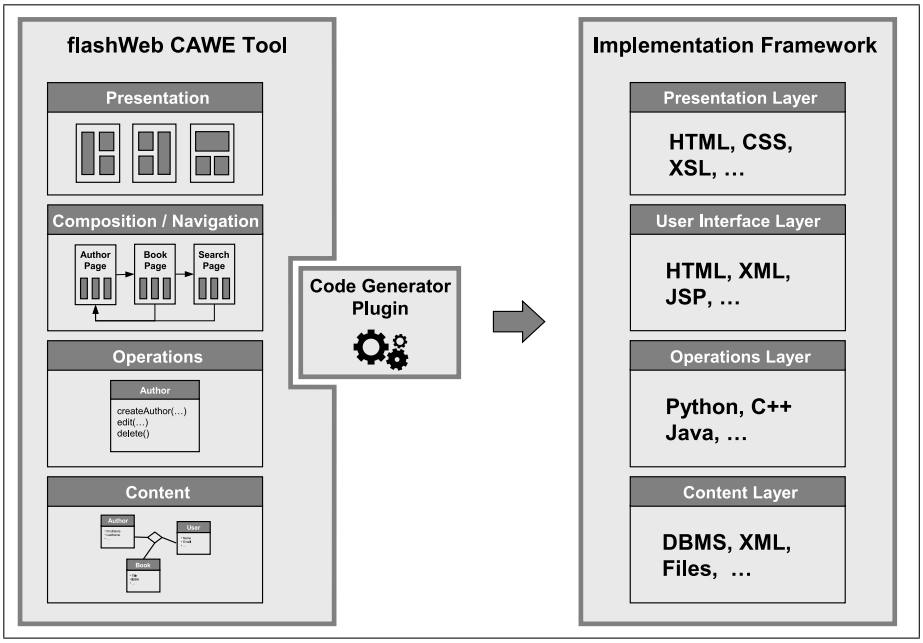

Figure 5.1: flashWeb's Implementation Strategy

tool supports an arbitrary number of generator plug-ins, thus for different target frameworks different plug-ins may be used.

The transformation of flashWeb models into a fully functional implementation is a non-trivial task. The first challenge is the complexity of the models. The code generator has to be able to incrementally traverse all models and transform them into an appropriate internal representation that suits the generation process. The second challenge is the complexity of the implementation framework. Web applications are usually implemented with a set of standard languages, e.g., HTML, CSS, JavaScript, and with a set of languages that depend from the runtime environment of the implementation framework, i.e., a server-side programming language like C#, Java, or Python and additional template languages like ASP or JSP. Additionally, most frameworks encourage or require the usage of a certain software architectural paradigm, e.g., the Model-View-Controller approach. Therefore, the actual implementation of the Web application usually includes a large number of different software artifacts, i.e., implementation files, configuration files, start-up scripts, etc. Accordingly, the code generator has to be able to handle all these different programming languages and different file types. The final challenge is to map the concepts of the modelled Web application to the concepts of the implementation framework. Of course, the complexity of this task is determined by the actual implementation framework and the chosen Web application architecture (see Section 2.3.4).

The flashWeb Web engineering method postulates the usage of a Web application architecture that corresponds closely to the design provided by its models. This is indicated in Figure 5.1, where each flashWeb model is mapped to a corresponding implementation layer of the implementation framework. Note however that most implementation frameworks do not distinguish between a *User Interface Layer* and a *Presentation Layer*. Usually, the latter one incorporates the functionality of both layers.

Of course, all components of the implementation strategy that have been introduced in this section may be implemented using technologies of an arbitrary framework, e.g., JavaEE, .NET, etc. In order to prove the validity of the flashWeb approach, the following languages and

frameworks have been used. The flashWeb CAWE tool is implemented in Java using the Eclipse framework. The Zope 3 Web application framework is used to implement the Web application and its built-in Web server is utilized as the runtime environment. The flexible nature of this lightweight framework allows to build Web applications with an arbitrary architecture. Therefore, Web applications developed with the flashWeb approach use all four implementation layers depicted in Figure 5.1. The functionality of the flashWeb CAWE tool is extended by a code generator plug-in, which is developed in Java and is an actual Eclipse plug-in that is seamlessly integrated into the CAWE tool. The following sections provide details about these three main components of the implementation strategy.

5.2 flashWeb CAWE Tool

The main component of flashWeb CAWE tool is the flashWeb Model Editor [JSSK07][Sch07], which allows the Web application developer to create, to modify, and to permanently store Web application models. It has been designed to support all graphical models of the flashWeb method and especially to support alternative notations of the models in order to facilitate efficient Web application development. The editor is implemented in Java and uses the Rich Client Platform of the Eclipse Framework [URI09] to run as a stand-alone application. Figure 5.2 depicts a screenshot of the editor's user interface.

The GUI of the Model Editor is kept as simple as possible and it relies on well known concepts for graphical user interface design. At the top of the application window, the *Menu* provides access to basic commands of the editor. The *Action Bar* is situated directly below the menu and includes icons for frequently-used features. Below the action bar you may observe the *Modeling Pane*, which provides a single modeling area of unlimited size. Note that this prototype does not have separate areas for different flashWeb models. Instead, all models are created in the same area and are separated by a simple color code. Content model elements are yellow, operation model elements are green, and finally elements of the user interface are blue. Next to the modeling pane on the right-hand side, there is the *Palette*, which contains a list of entries representing model elements.

The Web application developer may select an arbitrary element from the palette and create a new model element instance in the modeling area with a subsequent left click. Further customization of a model element is usually done through a context menu which is available through a right click if the corresponding model element is selected. The last component of the GUI is the *Properties Pane* at the bottom of the editor window. If a model element is selected in the modeling area, this pane shows all of the element's properties and allows the developer to manage them comfortably.

Dealing with graphical models is usually not an easy task. During development, the models get large quickly and the developer may find it difficult to keep an overview. To this end, all graphical models of the flashWeb approach provide alternative notations that substantially simplify those parts of the model that are not of concern to the developer at a given point in time during development. These alternative notations have been introduced in Sections 4.3.4, 4.4.4, and 4.5.4, respectively.

Additionally to these alternative notations, the editor provides further features that simplify working with large models. An important ability of the editor is to allow the fine granular configuration of views of model connections. Especially between the content model and the

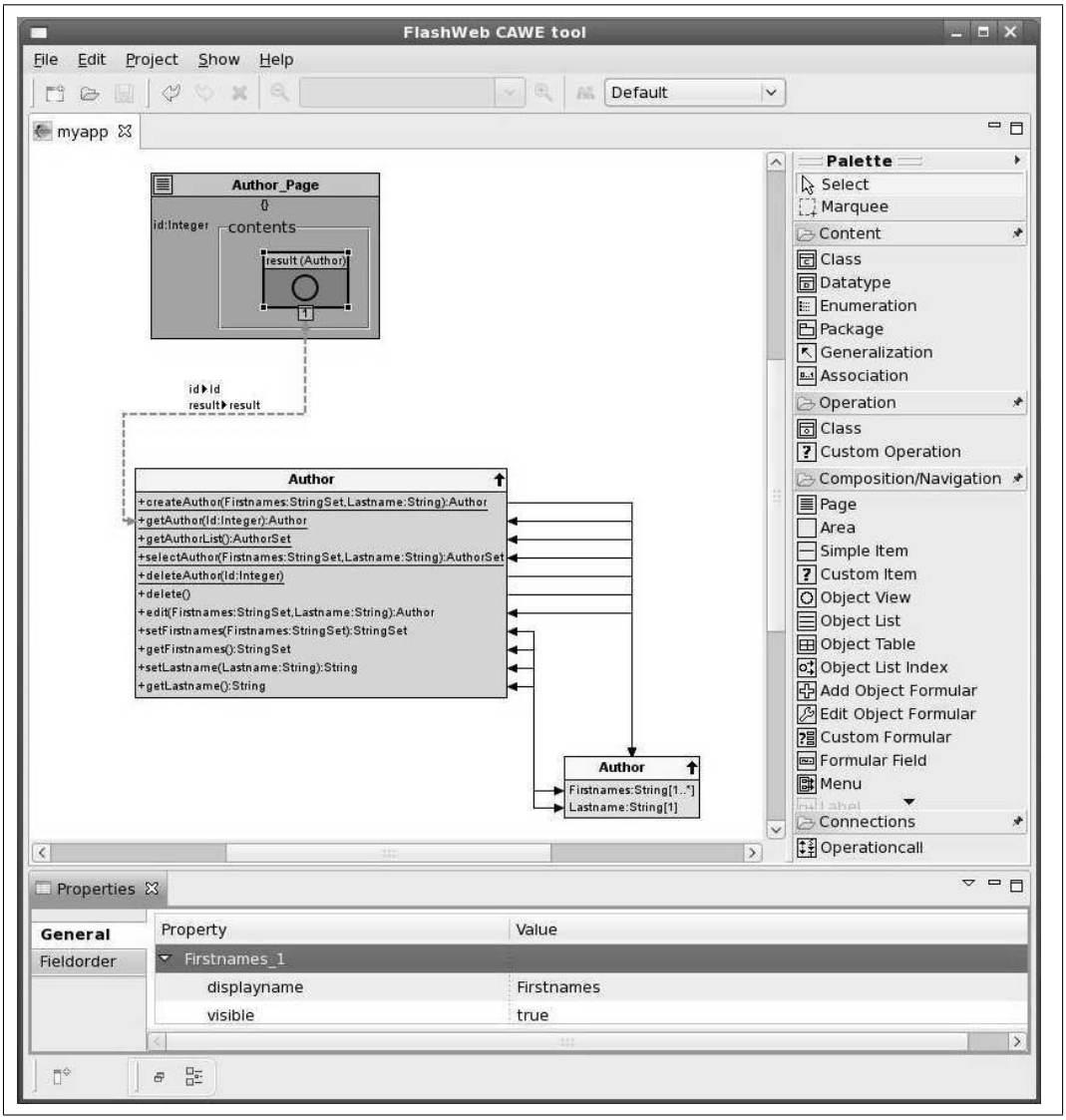

Figure 5.2: GUI of the flashWeb Model Editor

operation model of a Web application, the number of connections is usually high. Therefore, the editor allows to hide these connections temporarily and only show those that are relevant for the developer. Furthermore, the model editor automatically highlights all model connections that are attached to a selected model element. This feature allows the developer to get a fast overview of all model elements that are related to the currently selected element.

As mentioned before, the flashWeb CAWE tool may be extended by an arbitrary number of code generator plug-ins. Each plug-in supports one target framework or runtime environment for which code is to be generated. Generator plug-ins may be configured with the *Preferences* dialog, which can be reached from the *Edit* menu entry of the model editor. Figure 5.3 depicts the configuration of a generator plugin for the Zope 3 Web application framework.

The configuration options for a code generator plug-in are simplistic. As a matter of fact, there are only two options that have to be considered. First, the developer has to choose the *generation mode*, which may be "On-the-fly" or "On-demand". If the generator is set to the "On-the-fly" mode, it generates the Web application incrementally in the background without consulting the developer. Every time the developer changes a certain part of a model, the cor-

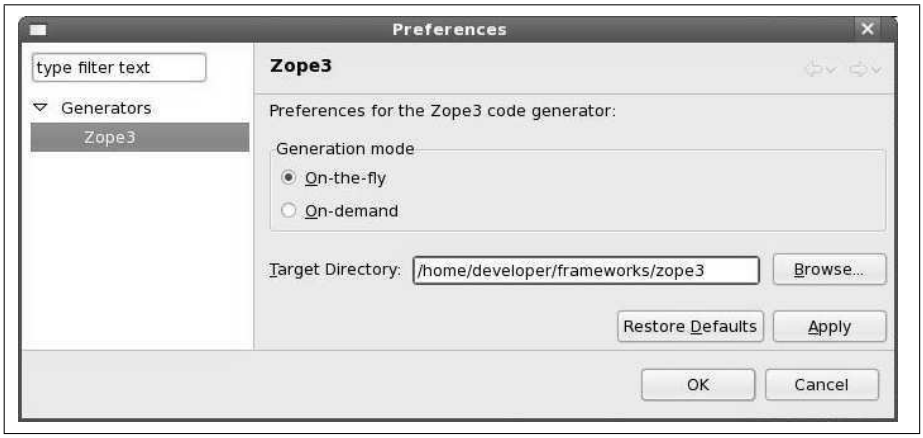

Figure 5.3: Code Generator Preferences Dialog

responding part of the implementation is updated automatically. If the generator is set to the "On-demand" mode, the developer has to trigger the generation process explicitly. The second configuration setting for a code generator plug-in specifies the *target directory* in which the generated code is to be stored. Note that here only this simple interface of the code generator is introduced. Section 5.4 provides actual details about a code generator plug-in for the Zope 3 Web application framework.

5.3 The Zope3 Web application Framework

The flashWeb Web engineering method is designed to be independent from any Web application framework or runtime environment. Except for the proposed separation of content, operational logic, user interface definitions, and presentation, the method allows to create Web applications with arbitrary architectures. Thus, theoretically an arbitrary number of target platforms may be supported with corresponding code generator plug-ins. However, an ideal target platform should provide a certain degree of flexibility, thus the proposed implementation layers (see Figure 5.1 in Section 5.1) can be realized. To this end, the flashWeb CAWE tool features a code generator plug-in for Zope 3, which is a powerful and extremely customizable Web application framework that is introduced briefly in the following sub-sections.

5.3.1 Introduction

Zope 3 [vW08] is an open-source, object-oriented, and component-based framework designed for Web application development. It's component architecture is extremely flexible and it allows to use the framework as a whole for publishing content and applications on the Web, but it also facilitates the utilization of single components of the framework to be included as part of any application that is written in Python. For example, Zope's Object Database does not have to be used for Web application development, it may be easily employed as the content storage component of a standalone desktop application.

Important features of the framework are object-oriented data storage, HTML/XML templating, form generation and validation, internationalization, support for strong security, cataloguing services, and support for testing. The framework also provides rudimentary support for

workflow management, access to relational databases and XML processing. However, these features are subject to further development.

Zope 3 relies on three main components if it is used as a server for running Web applications. First, the *Web Server* component handles the communication with Web clients via diverse protocols, i.e., HTTP, WebDAV, or XML-RPC. Second, the *Publisher* component deals with finding objects that are required by incoming requests and publishes them to the clients. Finally, the *Application* component handles all other aspects such as storing content, executing application logic, etc. Of course, the Zope 3 framework provides many further components, e.g., for storing data or ensuring security that may be used as a part of the application component.

5.3.2 The Component Architecture

The *Component Architecture* is the heart of the Zope 3 framework. It ensures that every component of the framework has a well-defined purpose and it also provides facilities for combining components in an extremely flexible manner. On one hand, concepts of the component architecture are used to tie together basic components of the framework itself. On the other hand, they may be used to develop component-based Web applications. Main concepts of the component architecture are introduced in the following sections.

5.3.2.1 Interfaces

An essential concept of Zope 3 is the notion of an *Interface*. This concept is known from different programing languages, e.g., from Java. In general, an interface defines certain functionality, which is to be provided by components that implement the interface. More specifically, an interface usually includes the list of attributes and the signature of methods of an object, as well as arbitrary documentation.

Note, however, that the usage of interfaces in Zope 3 is different from their usage in most programming languages, where they often serve as a solution for the lack of multiple inheritance. As Python does offer multiple inheritance, there is no need to use interfaces for that in Zope 3. Instead, interfaces have a more symbolic meaning and their sole purpose is just to identify and describe certain functionality.

However, in contrast to many other programming languages and frameworks, Zope 3 makes heavy use of interfaces. They can be used to register and to find components that provide certain functionality. Globally registered components are usually called *utilities*. They can be used to define so-called *adapters* that extend other components with certain functionality. Furthermore, interfaces are widely used for security purposes, i.e., to allow or restrict access to certain functionality.

5.3.2.2 Content Components

The aim of a *Content Component*, as the name suggests, is to handle content. Typical tasks of a content component are to allow the storage, the modification, and the retrieval of simple data. Data processing or data presentation are not the responsibility of a content component. Like any other component in Zope 3, content components are to be described with interfaces. In case of a content component, the interface specifies, which methods can be used to store, to modify, or to retrieve content, but of course it does not specify how the data is stored internally, thus the actual implementation of the content component remains replaceable.

5.3.2.3 Adapters

The *Adapter* concept is central to the component architecture of Zope 3. Adapters provide certain functionality for other components of the framework, e.g., for content components or for other adapters. Using this concept, the architecture of a Web application may be designed to use an arbitrary number of functionality layers.

The adapter concept can be easily explained by an example. Assume that a Web application has a content component that stores data about persons, e.g, the name, the birth date, etc. A requirement for the Web application could be to list all persons and to display for each person their age. Unfortunately, the content component does not have any knowledge about the age of a person. A typical solution for this problem is to use an adapter that is capable of computing the age of a person. Such an adapter would access the birth date of a person and acquire the current date from the system to calculate a person's age.

Adapters rely heavily on the concept of interfaces. The definition of an adapter has to specify which interface it adapts, i.e., for what kind of objects does the interface provide functionality and which interface it provides, i.e., what is the actual functionality of the adapter.

5.3.2.4 Utilities

The concept of a *Utility* is similar to the concept of an adapter. A utility provides certain functionality, which is described with an interface. However, utilities are independent of other components, i.e., they do not provide functionality for a certain type of components like an adapter does. Utilities rather provide a more general service that may be used by various other components of an application.

There are two basic types of utilities that are commonly used in Zope 3. Global utilities that follow the singleton software engineering pattern are registered and looked up only by their interface. Common examples are database connectivity or mail delivery. In contrast to them, the developer has the possibility to register utilities that are required several times. Examples for such utilities are object factories or different object views.

5.3.2.5 ZCML

The *Zope Configuration Meta Language (ZCML)* [vW08] is the glue that binds components of the component architecture together. The aim of ZCML is to be used by site administrators who maintain Web sites and Web applications. Using ZCML, the site administrator should be able to configure a certain application without having to understand or change the source code of included components. To this end, ZCML is based on XML and has simple semantics, thus the site administrator is not required to have programming skills. Main aspects that may be configured with ZCML are the registration of utilities and adapters, the registration of views, which are special adapters providing presentation for a component or the security setup of application.

5.4 flashWeb Code Generator Plug-in

Code generator plug-ins play a very important role for the flashWeb Web engineering method. Although the usage of graphical models offer various advantages for Web application development, it is the code generation capability of flashWeb that speeds up the development process

considerably and enables the method to be used for almost any Web project. The reader is referred to Section 6.3 for a discussion of flashWeb's support for different Web application development processes.

An essential aspect for any Web application is a clear and well-structured conceptual architecture, which builds the backbone of the application and may be extended with appropriate application modules. In Section 5.1, a four-layer Web application architecture is introduced that adheres to the four models of the flashWeb method. The aim of a code generator plug-in for the flashWeb model editor is to transform the graphical models to such a four-layered implementation using an appropriate implementation framework. Of course, the introduced four-layer architecture is merely an example and plug-ins supporting arbitrary Web application architectures are allowed. This section introduces the *Zope 3 Generator Plugin* that achieves this goal for the Zope 3 Web application framework.

5.4.1 Plugin Overview

Generating the implementation of a Web application is not a trivial task. There are many factors that add to the complexity of this objective. First, Web applications are usually implemented using several different languages. A Web application running on the Zope 3 framework uses HTML, JavaScript, and CSS at the client side, Python, ZCML, and the Template Attribute Language (TAL), a templating language for creating dynamic Web pages, at the server side. Correspondingly, the code generator has to be able to handle correctly the specifics of each language, e.g., the language syntax, indentation, and naming conventions.

Second, depending on the chosen implementation framework, a Web application component is usually implemented with a set of software artifacts, i.e., class files, configuration files, etc. For example, to define a content object the Zope 3 framework requires as a minimum that there exists a Python class file that contains the object's implementation, a Python interface file that contains the object's interface specification, and a ZCML configuration file that registers the object's interface in the system registry. Additionally, the configuration file has to be referred to from another configuration file in order to be loaded on start-up. Furthermore, the developer has the freedom to put several definitions, e.g., object definitions or object interface specifications into a single file. Correspondingly, there are many different ways to implement a Web application component regarding the partitioning of a component's specification, thus the Web application generator has to provide a solid strategy to deal with this problem.

Finally, the code generator provides diverse functionality that requires the coordination of components that fulfill a wide range of different tasks. The generator handles an internal representation of a Web application's model and reacts to events that signal a change in the models. It manages different implementation artifacts and maps them to model components. It handles code templates that help to generate reoccurring code fragments. Last but not least, it writes source code artifacts to the file system. This complex functionality is incorporated into the code generators architecture, which is presented in Figure 5.4.

The code generator has six main modules that depend on each other in various ways. Each module is represented by a manager component that coordinates the tasks executed by the corresponding module and also handles communication with other managers. The *Preference Manager* provides preference pages for the model editor GUI (see Figure 5.3), handles the permanent storage of preference settings, and provides access to them for other managers. The task of the *Dispatch Manager* is to handle events (e.g. create, change, delete) that originate from

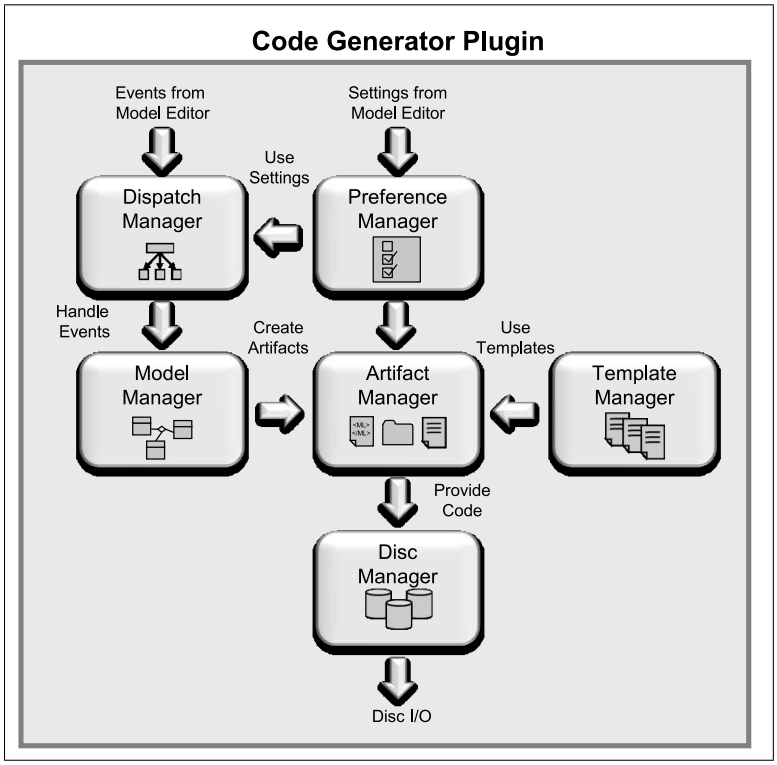

Figure 5.4: Architecture of the Code Generator Plugin

the model editor and to trigger appropriate actions of the generator. To this end, it maintains a registry of generator components that are capable of handling events. Depending on the event and the involved model component, the dispatch manager identifies the appropriate handler and signals the need for code generation to the corresponding handler. The *Model Manager* is responsible for the coordination of the code generation process. To this end, it provides handler components that know exactly what is to be generated for which model element in case of which model change event. To this end, this manager assigns to each model element a set of implementation artifacts and signals them the need for code generation. The *Artifact Manager* handles different implementation artifacts that are responsible for assembling the source code of an implementation file. To this end, each artifact has access to a set of code templates that help to create reoccurring code fragments. After assembling the code for a certain implementation artifact, the artifact manager forwards the code to be written to the file system. The *Template Manager* controls the look-up of code templates and the execution of the template based generation of code fragments. Generated code fragments are provided to the appropriate implementation artifact. Finally, the *Disc Manager* writes the generated code, which it receives from the artifact manager to the file system.

The described architecture of the code generator is of course independent from the actual implementation framework and also from the architecture of the implemented Web application. However, to be able to generate for a certain framework and to realize a certain Web application architecture, some sub-components of the introduced code generator modules have to be specific. On the other hand, it is desirable to be able to have several code generator plug-ins that are designed to generate Web applications for different implementation frameworks. Therefore, the code of the code generator is actually divided into two plug-ins. A generic code generator plug-in contains everything that is independent from the framework and from the

architecture of the generated Web application. This base plug-in can be used as a starting point to implement a specific code generator plug-in for an arbitrary Web application framework. The second plug-in is framework- and Web-application-specific, thus in this case it contains components that are specific to the Zope 3 framework and adhere to the four-layer Web application architecture introduced in Section 5.1.

Despite the separation of the code generator into two plug-ins, the architecture presented in Figure 5.4 is still valid. The generic plug-in provides abstract classes for all introduced managers and also for a set of further sub-components that may be specialized in the framework-specific package. Usually there are three main areas of a code generator that have to be tailored. First, the model manager has to provide handlers that are able to construct the Web application adhering to the intended application architecture. Second, the artifact manager has to provide artifacts that correspond to the requirements of the implementation framework. Finally, the code generation templates handled by the template manager have to generate code using the appropriate implementation languages.

5.4.2 The Content Layer

As indicated in Figure 5.1, for each of the four flashWeb models, the code generator is supposed to create a corresponding implementation layer of the Web application. This section describes how the Content Model of a Web application is implemented with the Content Layer.

The Content Model of flashWeb (see Section 4.3) facilitates the design of a Web application's content storage. To this end, the most important element of the model is the `Content Class`, which defines an object that is capable of storing data. A simple content class example is the `Author` class from the Book Portal (see Section 2.2.3) example scenario.

The Zope 3 Web application framework provides explicit support for the implementation of content objects. To this end, it offers the concept of so-called *content components* (see Section 5.3.2.2). A content component in Zope 3 is implemented as a sub-class of the `Persistent` class, which is an important component of the framework providing persistent storage for objects. Additionally, a content component usually implements the *IContained* interface, which declares that the component is contained at some storage location. Such a storage location is usually a so-called *container* that contains objects of a certain type. Figure 5.5 depicts an overview of generated artifacts that implement the `Author` content object and the corresponding `AuthorContainer` container.

The right-hand side of the figure depicts all implementation artifacts that are necessary to store `Author` objects. The artifacts are modules and files that contain the implementing source code. The implementation includes the two base modules "containers" and "objects" that contain container definitions and object definitions, respectively. These modules exist once for each content package and hold the specifications of objects and object containers of the package. The container for `Author` objects is implemented by the "authorcontainer" module, the "container.py" file that includes the container specification, the "interfaces.py" file that includes the interface specification of the container, and the "configure.zcml" configuration file that registers these components with the system. The implementation of the actual `Author` object is structured similarly to the implementation of the container. It consists of the "author" module that contains the "object.py" object definition file and the "interfaces.py" interface file. Additionally, it also includes the "events" sub-module which holds event definitions in the "events.py" file and the corresponding interface definitions. Finally, the "configure.zcml" con-

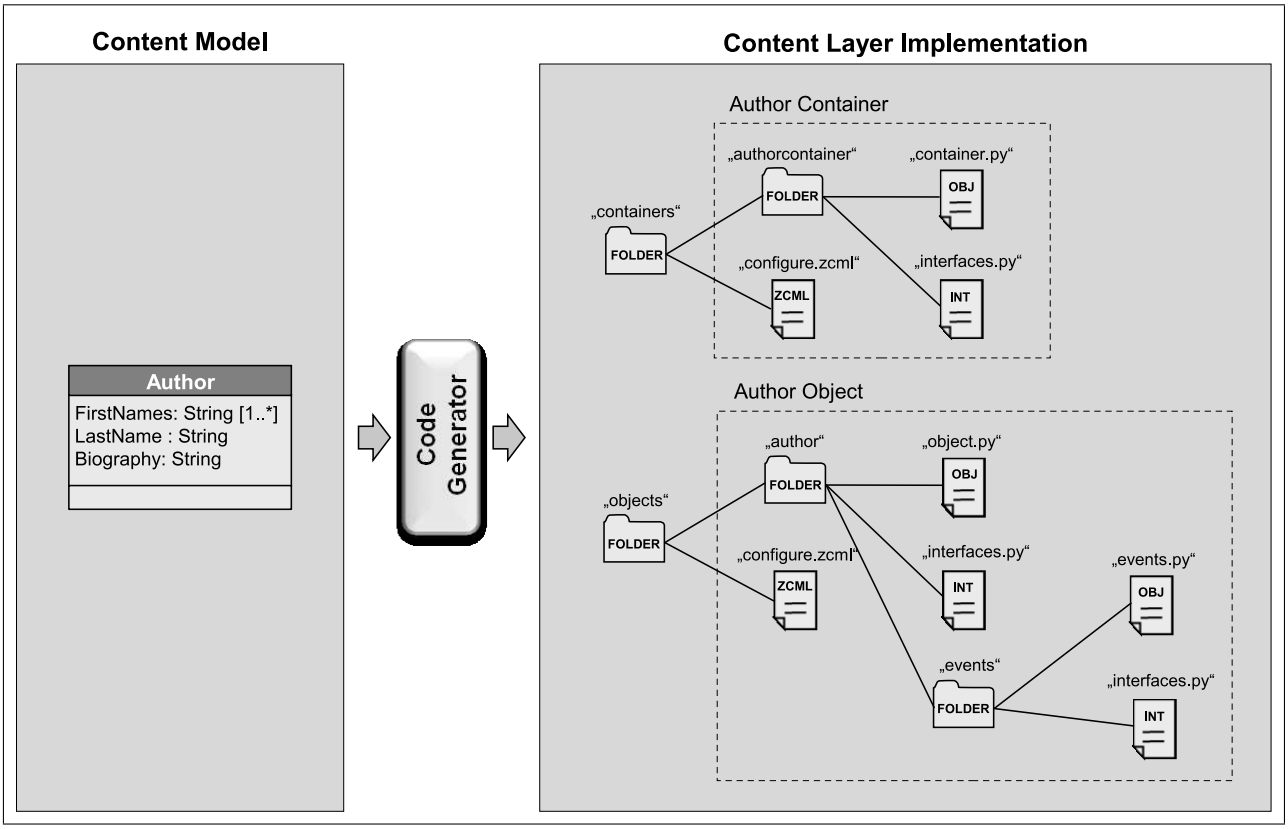

Figure 5.5: Implementation of the Content Model

figuration file registers all components with the system.

Note that there are some further concepts of the Content Model, e.g., content packages and associations that are not included in this example explicitly. Those concepts are implemented very similarly to the presented example therefore their description is omitted here.

5.4.3 The Operation Layer

The Operation Model of flashWeb (see Section 4.4) defines for each content object a corresponding `Operation Class` that provides a range of content management operations. Using the example from the previous section, this section describes the implementation of the `Author` operation class.

Components of the Operation Layer are implemented with *adapters* (see Section 5.3.2.3) of the Zope 3 Web application framework. Adapters allow to extend the functionality of a base component in a simple manner. A content object is merely a vessel for data storage; thus, for each content object, an adapter object is defined that provides access to the actual data. In case of the `Author` content class, this means the definition of an `AuthorOperations` adapter class that is described by the *IAuthorOperations* interface. Figure 5.6 depicts an overview of generated artifacts that implement the adapter.

The Zope 3 Web application framework offers the developer arbitrary flexibility considering the implementation structure of a component. To keep the generated implementation simple and uniform, the artifact structure that is used to implement an adapter is analogous to the structure, which was used to implement containers and content objects. Every operation

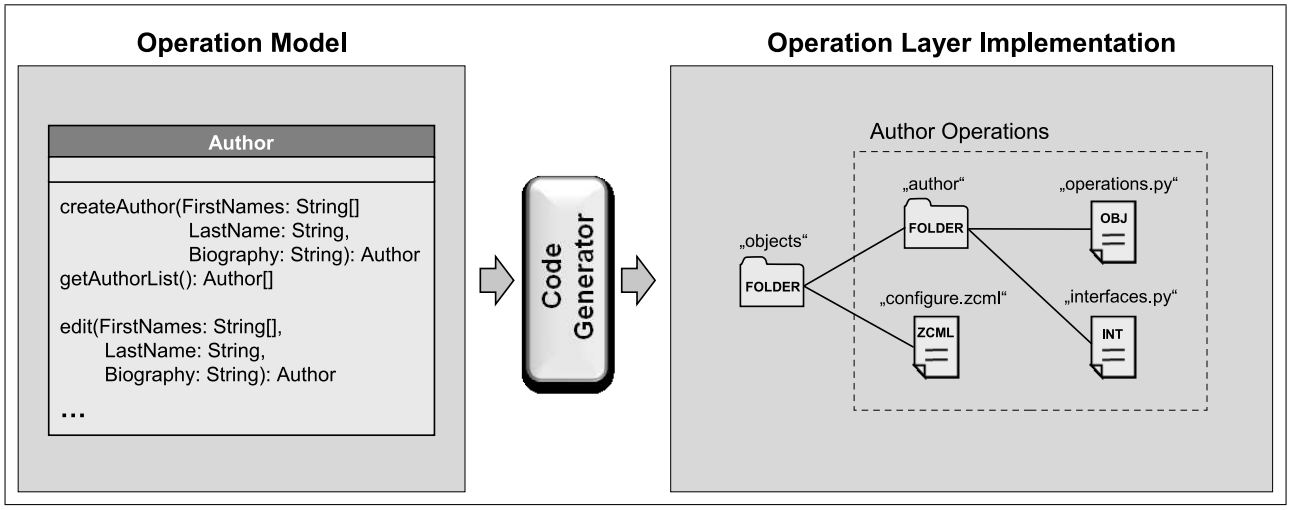

Figure 5.6: Implementation of the Operation Model

package contains an "objects" module that holds adapter specifications for the package. The `AuthorOperations` adapter is implemented with the "author" module, the "operations.py" file that contains the specification of the adapter, the "interfaces.py" file that contains interface definitions for the adapter and the "configure.zcml" configuration file that registers all components with the system.

5.4.4 The User Interface Layer

The implementation strategy for the User Interface Layer differs from the strategy of previously introduced implementation layers. A content object and the corresponding adapter that provides content management operations are implemented with dedicated classes, for which the implementing source code is generated from scratch. Of course, the generation of reocurring code is supported by source code templates, nonetheless the code of each content and operation class is unique. Instances of these classes are only created if the Web application's user executes a corresponding action. For example, a new `Author` object of the Book Portal application is only created if a system administrator enters the required data into a corresponding form and submits the data to execute the corresponding create operation.

In contrast to that, the implementation of the user interface is built by a set of standard components that correspond to the elements of the Composition/Navigation Model introduced in Section 4.5. This means for example that an `Area` element is implemented with a predefined `Area` class. Thus, the implementation of the user interface is build through the appropriate combination (nesting) of pre-defined implementation objects. Of course, this structure reflects exactly the structure of the corresponding model. Also, instances of these implementation objects are created at initialization time of the Web application as they are also used for creating the actual presentation of the user interface (see next section). Figure 5.7 depicts implementation artifacts for a small user interface fragment.

The left-hand side of the figure shows a partial Composition/Navigation Model. The model consists of the *Authors* page, which contains two `Area` elements. The *Authors Area* on the left-hand side also includes an `Object List` element, which presents a list of authors.

The implementation of this model is based on two main components. First, the *flashWeb UI*

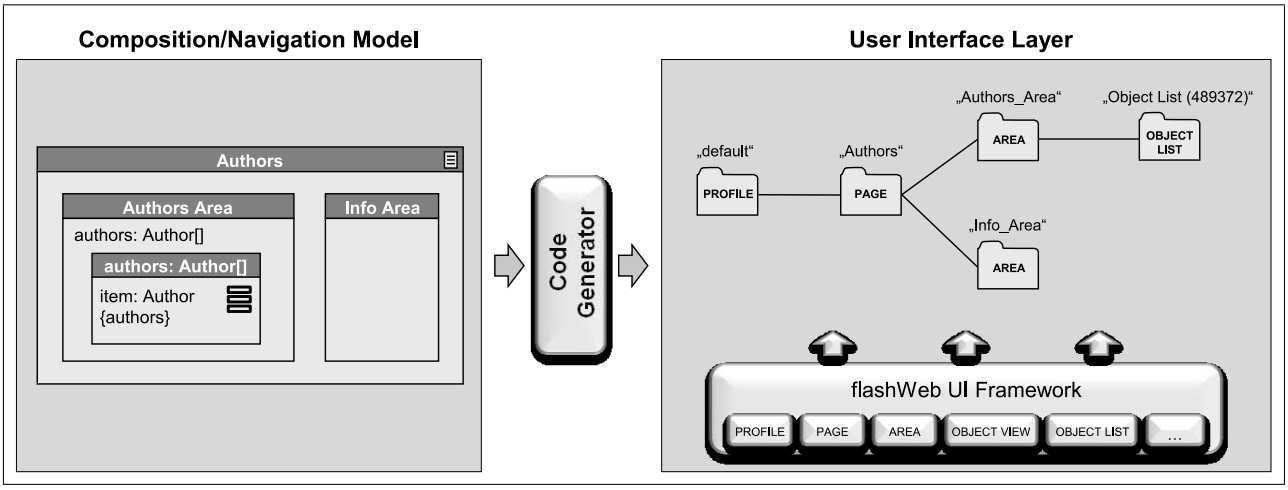

Figure 5.7: Implementation of the User Interface Structure

Framework provides for each model element an appropriate implementation class, for which an arbitrary number of implementation objects may be instantiated. Note that this framework is static and may be used for an arbitrary number of different Web applications. Second, the code generator creates a module structure that corresponds to the structure of the model. Each module consists of a folder and an "__init__.py" implementation file. Note, however, that these details are omitted in Figure 5.7. A task of a module is to provide initialization code that creates a corresponding implementation object at system startup and to serve as a container for further sub-modules.

In case of the current example, the *Authors* page is implemented with a *Page* module that is placed into the "default" user profile. This module specifies initialization code, which is executed at system startup and achieves three goals. First, it uses the `Page` implementation class from the flashWeb UI framework to create a corresponding implementation object. Second, it sets up this object with the data that is provided by the `Page` model element, e.g., the name of the page. Finally, it makes sure that initialization code of all sub-modules are executed recursively. Ultimately, the code generator does not create the implementation of the user interface directly but it creates an auxiliary structure of initialization modules that instantiates implementation objects at system startup.

5.4.5 The Presentation Layer

The last layer that is to be generated is the Presentation Layer. It defines for each user interface component its exact appearance to the Web application's user. The separation of presentation from content is an important content management paradigm. Especially the large number of potential client platforms (PC, PDA, mobile phones, etc.) requires that the Web application may be presented with different user interfaces.

In order to provide the highest degree of flexibility, the implementation of the presentation layer relies on dynamic page templates that are used to construct the presentation of user interface components at runtime. To this end, the *flashWeb UI Framework* provides for each user interface component, i.e., model element like `Page`, `Area`, etc., a presentation template that determines the basic rendering of the element. Note, however, that this basic representation is not generated and may be further refined by formatting expressions of the Presentation Model.

Figure 5.8 depicts implementation artifacts for the presentation of the user interface fragment from the previous section.

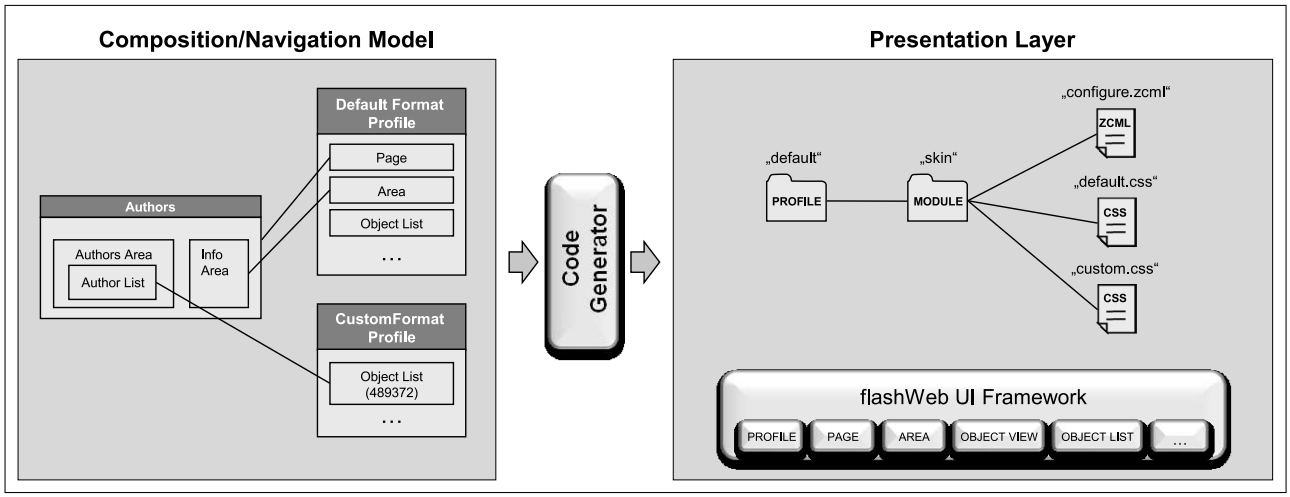

Figure 5.8: Implementation of the User Interface Presentation

The left-hand side of the figure illustrates that the Presentation Model of flashWeb uses two format profiles to define the fine-grained calibration of the user interface. Note, however, that the Presentation Model is not a graphical model. The reader is referred to Section 4.6 for more details.

The task of the code generator is to convert formatting expressions that are assigned to user interface components into implementation artifacts. To this end, it creates a simple artifact structure. As a matter of fact, all formatting expressions for user interface components of a certain user profile are generated into a so-called “skin” module. The module contains the two CSS files “default.css” and “custom.css”. As the names already suggest, these artifacts contain formatting expressions from the default and the custom format profiles, respectively. The “configure.zcml” configuration file registers the module with the system.

Ultimately, the presentation of a user interface component is determined by its basic presentation template and the assigned formatting expressions. However, the presentation of a nested component, e.g., a page, is of course composed of a number of sub-elements. Therefore, the nesting structure of user interface components, which is implemented by the User Interface Layer is necessary for the construction of the presentation of nested user interface components.

In case of the *Authors* Page example, the complete modeling, generation, and runtime process can be described as follows. The Web application developer uses the model editor to define the `Page`, the two `Area` elements, and, finally, the `Object List` elements. He also assigns formatting expressions using the format profiles. The code generator uses this model to generate a module structure for the User Interface Layer and a skin module for the Presentation Layer. At system startup the module structure is used to instantiate a corresponding object structure representing the user interface. If a Web application user accesses the *Authors* page, the runtime environment constructs the basic presentation code recursively using the presentation templates for the page and for all sub-elements. The presentation templates also integrate the formatting expressions from the skin module that together with the templates determine the final presentation of page.

5.5 Summary

Implementing Web applications with the flashWeb method is an easy task, because a huge part of the Web application's code is automatically generated from the models. A usually minor part of the application code, e.g., specific algorithms that cannot be modeled with constructs of the operation model, has to be coded by the developer and added to the model using custom model elements (see Section 4.4.2.7). After adding the custom part, a ready-to-run Web application can be generated from the models in a matter of seconds.

The flexible plug-in architecture of the flashWeb CAWE tool ensures the usability of the method with arbitrary target implementation frameworks. Regarding code generation, it does not matter which frameworks are supported by flashWeb code generator plug-ins. For each framework, different artefacts have to be generated that implement the Web application adhering to a chosen Web application architecture. Of course, each target framework requires a separate code generator plug-in. As a proof on concept, this chapter has introduced a code generator plug-in for the Zope 3 Web application framework. As previously described, generic components of the code generator reside in a basis plug-in that can be extended in a straightforward manner to create a specific code generator plug-in for alternative implementation frameworks.

CHAPTER 6

Evaluation and Conclusions

Throughout this work, many pros and cons were presented for model-based Web application development and, especially, for the flashWeb Web engineering method. Section 3.2 introduced the role of models for Web application development and Section 3.4 demonstrated how four prominent Web engineering methods apply them in the development process. The capabilities and limitations of these methods were discussed in concluding subsections, respectively.

The focus of this work is on the introduction of the flashWeb method, which of course requires legitimation. To this end, this chapter provides a detailed evaluation of the flashWeb method. Sections 6.1 and 6.2 describe some of the method's general advantages and limitations. Additionally, Section 6.3 provides a detailed discussion of flashWeb's support for different Web application development process models. Section 6.4 compares the flashWeb method with other Web engineering solutions and Section 6.5 evaluates the code generation process. Finally, Section 6.6 concludes this chapter by providing an outlook of possible future work.

6.1 Advantages of flashWeb

The flashWeb method does not require the employment of a certain Web application development process. To the contrary, the increased development speed through code generation enables the method to be used for both heavy-weight and light-weight development processes (see Section 6.3). To this end, it supports different development activities that are usually part of most development processes. The method's graphical models and of course the flashWeb CAWE tool may be used for the requirements gathering as well as for the design phase. Code generator plug-ins of the CAWE tool eliminate the need for a lengthy implementation phase. After potentially necessary refinements of design models, implementation artifacts are generated automatically. Finally, the advantages of graphical modeling and of the generative approach greatly facilitate any maintenance activities after product deployment.

A cornerstone of the flashWeb method is the utilization of graphical models, which are introduced in detail in corresponding sections of Chapter 4. A great advantage of flashWeb's graphical models is the excellent overview of the developed application. Furthermore, the utilization of different models to capture different aspects of the application ensures an appro-

priate separation of concerns, which is an important paradigm for the development of content management applications. The flashWeb method is also supported by a CAWE tool that allows to create, manage, and store the graphical models permanently.

An outstanding and unique feature of the flashWeb method is the Operation Model, which captures content management operations of a Web application. These content management operations may be combined freely inside the model and associated to components of the user interface through graphical connections. These connections seamlessly integrate arbitrary content management functionality into the Web application's user interface.

Another important advantage of flashWeb is the capability of the flashWeb CAWE tool to generate a fully functional implementation of the modelled Web application. This is ensured by two important characteristics of flashWeb's models. First, the models capture the entire functionality of the Web application. Application parts that cannot be expressed with standard elements are captured with custom elements (see sections 4.4.2.7 and 4.5.2.20). Second, semantic relationships between the models are clearly defined. Furthermore, the models may be combined graphically in order to create a cohesive specification of the Web application. This explicit model weaving technique enables the flashWeb CAWE tool to generate a ready-to-run implementation. The code generation capability of flashWeb is not only useful for speeding up the development process but also facilitates fast prototyping.

Last but not least, a further advantage of flashWeb is the clear structure and uniformity of the generated implementation. As described in Section 5.4, the four models of flashWeb are converted into four implementation layers. The architecture of the Web application is clearly structured and all generated components are uniform. Thus theoretically, the application can be maintained and extended easily without the flashWeb CAWE tool. A more practical advantage of a well-structured implementation is its support for debugging custom components. The flashWeb CAWE tool does not have an integrated runtime environment, thus a Web application that contains custom components has to be generated and then executed in a separate runtime environment that supports debugging.

6.2 Limitations of flashWeb

Like all other methods for Web application development, flashWeb has its advantages and limitations. This section introduces the most important problems of the approach but also offers partial solutions.

The first problem that a Web application developer using the flashWeb method encounters is the fact that the graphical models can get very large. This is of course a well-known problem of graphical modeling, which is inherent in all other model-based methods for Web application development. However, in case of the flashWeb method, this problem is probably even more intense. First, flashWeb employs a model weaving technique that uses explicit graphical connections between components of different models. Correspondingly, all three graphical models of flashWeb that are interconnected by these connections have to be displayed together. Second, the number of graphical connections between the Content Model and the Operation Model gets usually rather large.

Fortunately, the flashWeb method and the flashWeb CAWE tool offer a set of features that counteract this problem. First, each graphical model defines alternative notations (see Sections 4.3.4, 4.4.4 and 4.5.4) that allow the developer to focus on a certain part of the model.

To this end, model elements that are not of importance can be minimized. Second, the flashWeb CAWE tool provides a set of further features like zooming, hiding, or highlighting model connections, which make it easier to cope with large models.

Another aspect of flashWeb that may be considered as a problem are custom elements that integrate actual implementation code into the models. An example for a custom model element is a custom operation (see Section 4.4.2.7) that may be used to specify an arbitrary content management operation using a programming language of the target runtime environment. Of course, storing implementation code as part of the model is a mixture of two worlds, design and implementation.

Unfortunately, there is no ultimate solution for this problem. Almost every Web application, regardless of the application size, includes some kind of custom functionality that cannot be modeled with standard model elements. Of course, there exist several alternative solutions to deal with custom functionality. One is to generate the Web application and to extend the generated code afterwards. However, this and other solutions usually have more disadvantages than advantages. A more detailed discussion of different code generation strategies is provided in Section 3.2.6.

A final problem, that results from supporting custom model elements and for which the flashWeb CAWE tool does not provide a sufficient solution, is the cumbersome development of custom code. The general problem is that the flashWeb CAWE tool is not able to run the generated code directly and therefore it does not provide any support for debugging. Therefore, if debugging capability is required, another tool has to be employed. Ultimately, the development pattern for custom code includes four steps. First, a custom element containing custom code is added to the model with the model editor. Second, the model is used to generate the Web application. Third, the generated code is tested in the runtime environment and possibly debugged using an additional tool. Finally, if errors are found, the code is adjusted to eliminate them and the adjusted code is added to the model, which of course requires to repeat steps two to four all over again, alternatively, the adjustments can be added directly to the generated implementation to skip step two, in which case the developer should not forget to store the final code in the model after eliminating all errors.

6.3 flashWeb's Support for Engineering Web Applications

The flashWeb approach may be used in conjunction with most development process models. This section explains how flashWeb may be combined with heavy-weight and light-weight process models for conventional software development and, more importantly, it describes how flashWeb enables these process models to be used for Web application development in the first place.

6.3.1 Support for Heavy-weight Process Models

The flashWeb approach works very well in conjunction with heavy-weight process models. This category of process models usually divide the development process into a set of phases, e.g., specification, design, implementation, etc. The flashWeb method explicitly supports two common development activities, design and implementation.

Its support for the design phase is twofold. First, it facilitates the planing of the Web application's architecture. To this end, graphical models provide a *package* construct that allows

the developer to divide the Web application into different modules. The flashWeb method uses the basic paradigm of separating content, operations, and presentation and, correspondingly, it provides appropriate models to capture these aspects separately. Thus the architecture of the system must adapt to this architectural pattern. Second, flashWeb supports detailed design relying on well-known object-oriented principles, e.g., modeling with data types, classes, attributes, and operations. To this end, it provides three graphical models, the Content Model (see Section 4.3) the Operation Model (see Section 4.4), and the Composition/Navigation Model (see Section 4.5), and appropriate fine-grained model elements in all models. Note that the flashWeb approach utilizes a CAWE tool that includes a graphical editor for the comfortable creation of the models. The editor also supports different integrity checks so that the integrity of the design is guaranteed.

A highlight of the flashWeb approach is its ability to produce a fully functional implementation of the Web application. To this end, a code generator component of the flashWeb CAWE tool uses the models that have been created in the design phase to automatically generate the source code of the Web application. A considerable part of the source code is uniform as it is generated from standard model elements. The generated code provides standard functionality such as the definition of objects, full access to object attributes, full access to the instances of a certain type (e.g. filtering, sorting, etc.), and the management of relationships between objects. Given a reasonable design, the portion of code that can be automatically generated may be up to 90% [JSKM06a]. A usually minor part is *custom* code that has to be programmed and stored together with the models as input for the generator. Programming routines that implement non-standard behaviour belong to this category. To this end, the models provide so-called custom elements that allow to attach custom code to the design. Ultimately, using the flashWeb method and the flashWeb CAWE tool the development team may avoid a large part of the implementation effort and concentrate on the custom part of the code.

A vital aspect for any heavy-weight development process model is thorough documentation. Documents are used for communication between customers and developers and also among members of the development team. For example, in most heavy-weight development processes, the requirements specification is a central document that is used by customers and developers to agree on the functionality of the developed system. Similarly, the design document is used by all team members who participate in the design or the implementation of the system. The design document usually employs some kind of graphical notation and includes a considerable number of figures and diagrams. This is certainly the case if the RUP is used as process model as it heavily relies on UML to capture design. The flashWeb approach satisfies the need of heavy-weight processes for extensive documentation in an excellent way. Its graphical models constitute a comprehensive documentation of the entire system. In contrast to many other methods, flashWeb provides custom model elements for extending the system with custom code, thus the generated source code has never to be modified. This approach ensures that the models and the source code of the system are always synchronized.

Testing is a fundamental activity in every serious process model for Web application development. The flashWeb approach minimizes the necessary testing effort of Web applications considerably. As mentioned before, with this approach a large part of the Web application's implementation is generated and these components should be without any defects, thus there is no need to test them. Of course, custom components must be tested as usual. However, the overall testing effort is considerably reduced as the portion of custom code is small compared to the amount of uniform generated code.

6.3.2 Applying Heavy-weight Process Models to Web Application Development

Heavy-weight process models are usually not adequate for Web application development. The Web application development process bears some characteristics (see Section 3.3.3.2) that are not well supported by heavy-weight process models. However, employing the flashWeb method in conjunction with a heavy-weight process can eliminate some of these incompatibilities, thus the corresponding process model may be applicable for Web application development after all.

One of the most predominant characteristics of Web application development is the frequent change of requirements. Usually, heavy-weight process models cannot effectively deal with changing requirements and the cost of late modifications is very high. Heavy-weight process models rely on thorough design and documentation that is always kept up-to-date. Thus, if requirements change or new requirements emerge, besides the implementation also the design and all corresponding documentation artifacts have to be updated. The flashWeb approach alleviates some of these insufficiencies as the approach ensures that design and implementation are automatically kept synchronized. Furthermore, the capability of generating most of the implementation saves a lot of time that may be used to update additional documentation, e.g., the specification document or the user guide. Using the flashWeb approach, the overall adjustment effort that is necessary to react to changing or new requirements is considerably lower, thus heavy-weight process models may be better applicable to Web application development.

Another typical characteristic of Web application development is the necessity of short development cycles, which ensure that a new version of the application may be shipped to the customer after a short period of time. It is of course difficult to fulfill this requirement if, besides the implementation effort, a range of other activities that are required by a heavy-weight process have to be executed. As described previously, flashWeb accelerates the development process considerably and ensures that design and implementation are always synchronized. Thus the approach not only ensures a improved response to changing requirements but also facilitates shorter development cycles.

Note that the described aspects address only the most important problems that occur, if a heavy-weight process model is to be employed for Web application development. Of course, there are many further challenges (see Section 3.3.3.2) that have to be considered. For example, incremental and parallel development may or may not be supported by a particular heavy-weight process model, thus it is not appropriate to make general assertions regarding these requirements. Note that the flashWeb approach supports many of these requirements (see Section 6.3.5).

6.3.3 Support for Light-weight Process Models

Light-weight or agile software development processes concentrate on people, collaboration, and communication instead of on planing, tools, and documents. Thus at the first glance, it may not seem obvious why to employ a method together with an agile development process that relies on thorough design and a CAWE tool. However, as the following paragraphs describe, there are many typical aspects about light-weight development processes that are well supported by flashWeb.

A common characteristic of agile process models is incremental development. The applica-

tion is not planned and designed completely during a early phase of the development process but divided into several increments that are designed and implemented consecutively. An initial increment of the system usually implements a basic architecture, which is gradually extended by subsequent increments. The flashWeb method provides excellent support for incremental development. All flashWeb models provide structuring elements that, on the one hand, allow to create a basic architecture of the system and, on the other hand, enable the developer to define arbitrary system modules. Furthermore, the models provide additional fine-grained model elements that may be used to fill the modules with detailed functionality. Ultimately, the developer enjoys complete freedom to split up the functionality of the system into an arbitrary number of increments.

An important advantage of light-weight process models compared to their heavy-weight counterparts is their openness to changing requirements. Agile process models try to minimize the additional overhead of detailed planing and extensive documentation. Consequently, additional requirements for the application may be incorporated into the development process more smoothly. One way to anticipate and to respond to changing requirements is certainly incremental development, which is well supported by flashWeb. Another advantage of the method is the utilization of a CAWE tool that supports the generation of the implementation. The increased development speed is of course independent from the process model, nevertheless it greatly facilitates fast response to any change of requirements.

Simplicity is a core value of most agile development process models. Of course, a simple solution usually causes less problems than a more complex one, because it is less probable that it contains defects and it is easier to create and to maintain. The flashWeb approach is all about uniformity and simplicity. The provided set of standard model elements encourage the developer to create a standard solution before thinking about a possibly more complicated custom approach. Additionally, source code that is generated from the models is highly uniform. Thus, it is easy for the developer to integrate custom code into the implementation.

Another core value of agile software development is its focus on people and communication. As a method for Web application design and implementation, flashWeb does not explicitly boost communication between members of the development team, however, it doesn't restrain it either. As a matter of fact, it is the responsibility of the process model to stress the importance of communication and encourage team members to communicate. For example, an important practice of the XP process model is programming in pairs (see Section 3.3.3.3), which undeniably facilitates communication between developers. Of course, the flashWeb CAWE tool may be used by pairs of individuals to develop the models and generate the implementation and, thereby as a side effect, facilitate communication.

6.3.4 Applying Light-weight Process Models to Web Application Development

Generally, light-weight process models suit Web application development very well. There are several key characteristics of Web application development (see Section 3.3.3.2), e.g., short development cycles or changing requirements that are explicitly supported by agile process models. As described in previous sections, the flashWeb method facilitates fast Web application development and flexible response to changing requirements, thus it may easily be integrated into light-weight process models. Therefore, regarding these aspects the flashWeb method does not have to act as an enabler, which is certainly the case with heavy-weight process models (see

Section 6.3.2).

However, there is one aspect about Web application development that is difficult to accommodate by an agile process model. The complexity level of the Web application may increase considerably during the development process. Light-weight process models are primarily suitable to develop small and mid-size applications. However, if the complexity level of the Web application gets too high, it may be necessary to assign more resources to planning and design, which is usually a strength of heavy-weight processes. The flashWeb method provides the necessary thoroughness, which is required to build a complex Web application, without slowing down the development process. It relies on graphical modeling and the generation of the Web application's implementation. On the one hand, the modeling approach facilitates the definition of a system design that is flexible enough to adapt to increased complexity, on the other hand, the generative approach compensates for the necessary modeling effort. After all, the flashWeb method may enable a light-weight process model to better adapt to increasing level of complexity.

6.3.5 Further Support for Web application Development

Previous sub-sections concentrated on flashWeb's general support for different process models and on its characteristics that enable the employment of these models for Web application development. However, the flashWeb approach has some further characteristics that are not process-model-specific but are still relevant for Web application development.

The first aspect to be mentioned is component reuse, which is one way to deal with the enormous time pressure that is typical for Web application development. Of course, the initial development of reusable components requires additional effort. One needs to identify functionality that is reusable and partition it into appropriate modules. Furthermore, each module needs a simple, easy to use, and properly documented interface. Of course, the initial effort pays off if these components are employed in the same or in follow-up development projects.

The flashWeb approach supports component reuse at two levels. First, all flashWeb models define fine-grained standard elements that are small components to be used repeatedly to build the application. For example, the Composition/Navigation Model (see Section 4.5) provides several generic view components that can be flexibly combined with other model elements to present a uniform view over data originating from different sources. This may be described as component reuse in the small. Second, flashWeb models provide different structure elements that allow the definition of reusable modules. The graphical models provide an excellent module documentation that is always up-to-date. Furthermore, the flashWeb CAWE tool allows the easy definition and management of models, thus with its help, it is even possible to create different skeleton applications that may be reused in different development projects. This may be described as component reuse in the large.

Another characteristic of Web application development is the parallel execution of activities. As described in Section 3.3.3.2, parallelism is natural for Web application development for two reasons. First, usually there are several disciplines that are involved in the development process, e.g., system architects, programmers, graphic designers, etc. Second, parallel execution of similar tasks is often used to speed up the development process.

The flashWeb approach facilitates both forms of parallelism. To this end, it provides different models that may be created by developers in parallel focusing on different tasks. First, a modeling expert may create a Content Model (see Section 4.3) and an Operation Model (see

Section 4.4) that define the content structure and basic content management operations of the system. Second, at the same time a programmer may develop a custom module of the Operation Model, which provides functionality that is not covered through basic content management. Finally, parallel to the first two developers, a user interface designer may create a Composition/Navigation Model (see Section 4.5) that defines the user interface of the system. Additionally, each of these tasks may be split into several sub-tasks that can be executed by different members of the development team in parallel.

This high level of parallelism is explicitly supported by the flashWeb approach. To this end, different flashWeb models are employed to capture different aspects and the models may be flexibly combined to yield the overall system. However, because of the proper separation of concerns, this combination may also occur after significant progress has been achieved in each of the parallel activities. Of course, the developers have to agree on appropriate interfaces that ensure a smooth integration of components that have been developed in parallel.

6.4 Comparison with other methods

The flashWeb method has been thoroughly described in Chapter 4 of this work and, additionally, Sections 6.1 and 6.2 highlight some of the method's advantages and limitations. Additionally, this section compares the flashWeb method to other Web engineering methods, some of which have been introduced in Section 3.4.

6.4.1 Feature Comparison

The most prominent characteristics of the flashWeb method are described in Section 4.1. Many of these characteristics are superior to equivalent features of other Web engineering methods as described subsequently.

A common characteristic of model-based Web engineering methods is the utilization of graphical models for Web application design. Different models are applied to capture different aspects of the Web application (see Section 3.2.4). Unfortunately, many Web engineering methods provide model elements that have a rather high level of abstraction. Such model elements cannot be used to capture fine-grained functionality and therefore these Web engineering methods are only partially suitable to develop current day Web applications that usually provide sophisticated functionality. Note that most Web engineering methods introduced in Section 3.4 employ model elements that have a rather high level of abstraction or simply lack model elements to capture specific functionality (e.g. content management operations). Detailed comments are provided as conclusions of the corresponding sections. In contrast to that, the flashWeb method aims at providing fine-grained model elements in all its models to achieve a maximal coverage of the Web application's requirements.

An important paradigm for the development of any complex system is the separation of concerns. This paradigm is respected by most Web engineering methods by providing different models to capture different aspects of a Web application. For example, all methods that were introduced in Section 3.4 provide a separate model for capturing the content structure and the user interface of the Web application. This enables different experts to work on different aspects according to their expertise. However, as described in Section 3.4, Web engineering methods (e.g. the UWE method) often utilize a dominant content model that has

a great influence on what other models may specify. Another example is the WebML, which defines content management operations as part of the user interface specification. Dependencies between models or merging models into each other result in unnecessary constraints for the modeller and it restricts the parallel development of different application parts. In contrast to that, no flashWeb model constrains any other flashWeb model. Similarly to other Web engineering methods, flashWeb models build on each other, i.e., the Operation Model provides content management operations for objects that have been defined by the Content Model, etc. However, models are not restricted to those concepts that have been defined by lower level models. For example, flashWeb's Operation Model may define arbitrary utility classes that are not related to the Content Model in any way. As a matter of fact, flashWeb provides the greatest amount of independence possible between it's models by providing graphical references between different models.

A very important objective of Web engineering is code generation (see Section 3.2.6). The development speed-up that results from generating a part or the entire implementation of a Web application instead of producing it by manual programming, provides additional time and resources that can be used to create a thorough design and an appropriate documentation. In contrast to flashWeb, most Web engineering methods do not provide code generation support. From the set of methods introduced in Section 3.4, the WebML is the only one capable of generating at least a partial implementation of the modeled Web application. The lack of ability to generate a functional implementation can be ascribed to two common problems. First, Web engineering methods usually fail to create sufficient connections between their models. Thus, even if they could generate code using their different models, a significant programming effort would be necessary to merge the generated code manually after the generation step. As described in the previous section, flashWeb solves this problem by providing graphical connections between its models. These graphical connections define precise relationships between model elements that can be transformed into working code. Second, those Web engineering methods that are capable of code generation often support only partial generation, i.e., after the Web application is generated, the code must be adjusted manually to include custom components which cannot be generated. After that no further code generation is possible, thus an incremental development approach is not supported. For example, this is the case with the WebML method. In contrast to that, flashWeb models provide extension mechanisms (see Section 4.1) that allow to specify custom components and integrate them into the models. Accordingly, the models and the generated code are always synchronized and the Web application can be regenerated easily after the models have been changed. Thanks to its extension mechanisms, the flashWeb method is capable of generating a fully functional Web application from its models. Further details on code generation, especially on the distinction between full and partial code generation capability, are provided in Section 3.2.6.

Many Web engineering methods are database-driven, thus they rely on modeling techniques from the world of relational database management, e.g., Entity-Relationship modeling. In contrast to that, many other methods use UML or proprietary graphical notations that support object-oriented design. In Section 3.4, Web engineering methods from both areas were introduced. Similarly to the latter ones, the flashWeb method relies on object-oriented modeling. To this end, it uses the UML and UML-like syntax for all models.

6.4.2 Summary

Gu et al. argue [GHSL02] that there is a considerable gap between the informational design and the functional design in models of existing Web engineering approaches. While informational design is well understood and supported, the functional design is usually covered poorly. A comparison of selected Web engineering methods (e.g. OOHDM, WebML, UWE, etc.) with a set of requirements established by the authors show that the "Ability to Model Sophisticated System Functionality" and the "Ability to Link Information Architecture with Functional Architecture" of these modeling approaches are insufficient. Figure 6.1 (adjusted from [GHSL02]) shows a comparison of several Web engineering methods including flashWeb, which has been additionally inserted into the original diagram.

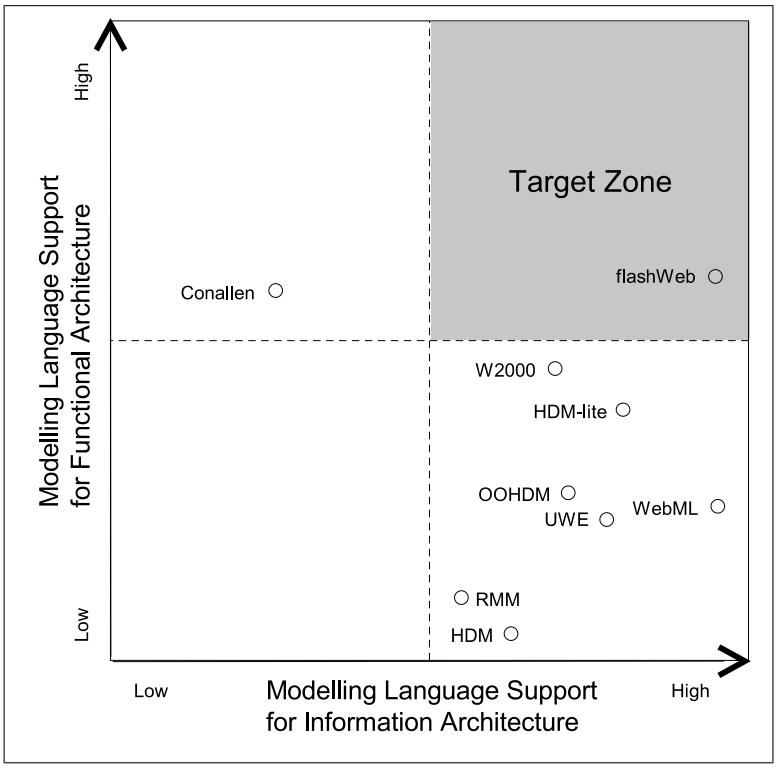

Figure 6.1: Web Engineering Methods Comparison

The x-axis indicates a method's capability to model a Web application's information architecture. As mentioned before, most Web engineering methods support this task well. This is also the case for flashWeb. It's content model allows the exact definition of a Web application's information requirements and standard content management operations integrate any content into the user interface seamlessly. To this end, flashWeb employs model elements that have similar expressive power as corresponding model elements of the WebML. Therefore, these two methods are at the same level considering the x-axis of Figure 6.1.

The y-axis depicts a method's capability to model a Web application's functional architecture. As Figure 6.1 suggests, this task is generally not well supported by the listed methods. They do not provide a solid strategy for the definition of functional characteristics. Some methods integrate operations into existing models, e.g., the content model and, in most cases, operations are not well connected to the model defining the user interface, thus it is difficult to integrate application logic into the user interface.

This is exactly the main challenge that is addressed by the flashWeb method. The Operation Model of flashWeb provides an essential basis for a Web application's functional architecture. It specifies basic content management operations that provide a solid connection between informational design (i.e., the Content Model) and functional design (i.e., the Operation Model). Furthermore, the method's model-weaving strategy allows an easy integration of functionality into the user interface. Therefore, the flashWeb method is higher ranked on the y-axis than other methods in this comparison. Note, however, that even flashWeb is far from the top end of the y-axis. This is due to the fact that it is yet to support higher level concepts for modeling functional characteristics, i.e., design patterns or business processes.

6.5 Code Generation Statistics

An outstanding feature of the flashWeb Web engineering method is its capability to generate a fully functional Web application for a selected implementation environment. To this end, the developer models the Web application with the flashWeb CAWE tool employing two types of model elements. Standard elements satisfy common Web application functionality and custom elements can be used to allow for application-specific requirements. An important question is of course, how large is the amount of custom code in a real Web application? Lets formulate this question another way. Is the ratio of standard model elements to custom model elements large enough to justify the usage of flashWeb and does the code generation capability really provide a speed-up of the development process? To answer these questions the flashWeb development approach has been tested in a real Web application development project.

6.5.1 Test Subject

The aim of the three-year research project *nova-net* financed by the German Federal Ministry of Education and Research was to develop innovative methods and tools to support innovation management process in enterprises. Main activities of the project were focused on two major areas. The first area was dealing with the development of methods for finding information on the WWW that can be utilized to support the innovation management process. Expert knowledge has been identified as a crucial asset, thus, the goal of the information retrieval process was to identify experts on the Web. Based on previous work [Jak03][JGHN05][JGNM05], an information retrieval tool [KSJ06][KJW07][KSJ07] has been developed capable of identifying similar Web documents and extract their authors, who have been identified as potential experts.

The second focal point of the research project was the development of methods and tools for managing and evaluating information in order to support decision making processes. To this end, a method [JKK+07] and a corresponding framework called the Scenario Management Framework (SEMAFOR) [JKS05] have been developed. The framework has been implemented as a Web application for the Zope 3 application server. It has four basic modules that can be combined in a flexible manner to compose and evaluate business scenarios. The *Descriptor Module* allows to gather business factors and their development over time. The task of the *Expert Module* is to manage experts that deliver assessments about business factors. The *Questionnaire Module* provides functionality to conduct Web-based expert surveys about arbitrary business factors. Finally, the *Scenario Module* allows to combine business factors into business

scenarios and use expert knowledge to automatically compute probabilities and assessments for the scenarios. Figure 6.2 shows a screenshot from the descriptor module of SEMAFOR.

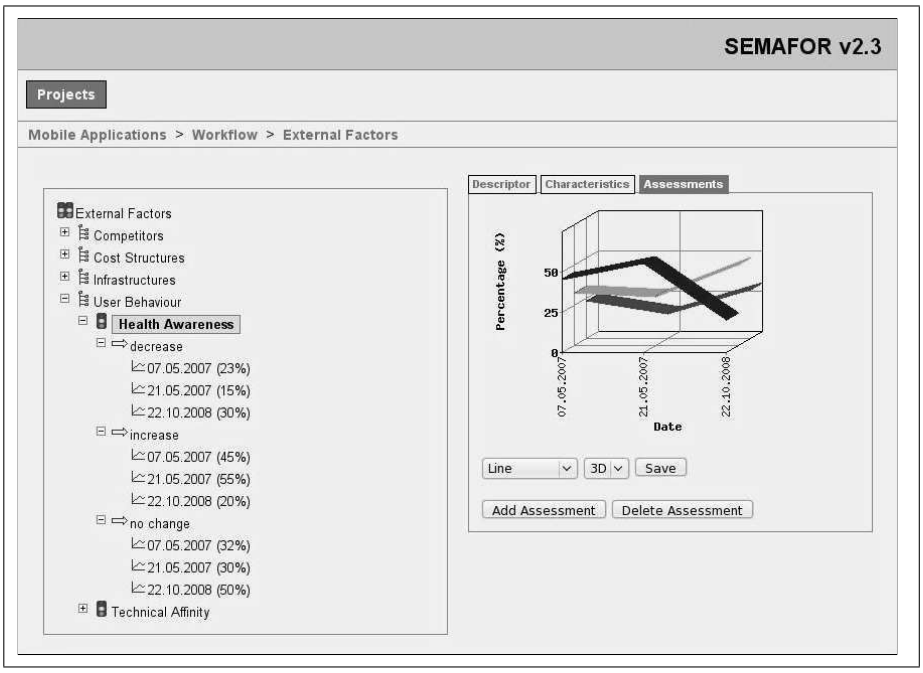

Figure 6.2: SEMAFOR Screenshot

User interface screens of the SEMAFOR tool have a uniform layout. Each screen is divided into two areas. The left-hand side of the screen is occupied by the *explorer* area, which usually shows objects of a certain module with a tree view. The right-hand side of the screen contains the *information* area, which provides details about the object that is selected on the left. Additionally, a breadcrumb above the areas allows fast access to the object tree of the explorer area.

The example in Figure 6.2 shows a screen from the descriptor module. The left-hand side of the screen shows a tree of business factors, also called *descriptors*. In this case, the right-hand side of the screen shows the development of the “Health Awareness" descriptor, which is selected on the left. This descriptor has three possible directions for development, i.e., “decrease", “increase", and “no change". The diagram on the right-hand side shows how an expert judged the probability for these development directions for three different moments in time.

It is obvious that SEMAFOR is not a trivial Web application. It provides a sophisticated user interface that allows the user to view and manage content in various ways. Accordingly, it is very well suited to test a model-based Web application development approach. On the one hand, it provides functionality that can be easily captured with standard model elements, e.g., objects that are represented with uniform tabular views. On the other hand, it integrates custom functionality like the computation of scenario probabilities or the generation of expert questionaires.

Therefore, SEMAFOR has been selected as a test subject for the flashWeb method. Note, however, that SEMAFOR has been modeled and generated with an early version of the flashWeb CAWE tool [JSKM06a]. As a prototype, this tool did not have a graphical editor but required models expressed in XML as input to generate a Web application. However, this early version already used the same models as flashWeb, thus, the results of the code generation process that

are presented in the next section are also representative for the code generation process of the graphical flashWeb CAWE tool. The idea of using graphical models instead of XML has been published in [JSKM06b].

6.5.2 Test Results

The generated implementation of SEMAFOR [JSKM06a][JKSB06] adheres to the principles of the flashWeb Web engineering method and employs the four implementation layers introduced in Section 5.1. Table 6.1 shows statistics about the generated code of SEMAFOR for each of these implementation layers. The term *standard code* means that the code has been completely generated from the graphical models. The term *custom code* means that the code has been added manually.

Layer	LOC	Percentage
Content Layer	3455	30.01%
Operation Layer (standard)	2601	22.58%
Operation Layer (custom)	682	5.92%
Composition/Navigation Layer (standard)	4247	36.88%
Composition/Navigation Layer (custom)	456	3.95%
Presentation Layer	76	0.66%
Total	11519	100%

Table 6.1: Code Generation Results

The Content Layer of a Web application developed with flashWeb contains only generated standard code. In case of SEMAFOR, the code of this layer amounts to 30.01% of the entire source code. The Operation Layer includes standard code as well as custom code. Standard code of this layer amounts to 22.58% of the entire code. The fraction of custom operations is 5.92% of the total, which is about one fifth of the Operation Layer. The Composition/Navigation Layer also has standard code, which makes up 36.88% and custom code, which amounts to 3.95% of the total. Finally, the tiny portion of the Presentation Layer makes up 0.66% of the entire code.

These figures characterize the implementation of a single Web application. Of course, the source code distribution between the different implementation layers and especially the ratio between standard and custom code may vary from application to application. As a matter of fact, SEMAFOR provides a great amount of specialized functionality that is difficult to express with standard components. The percentage of custom code is about 20% in the Operation Layer and 10% in the Composition/Navigation Layer adding up to a total of 10% of the entire source code of the application. This ratio is considered to be above the average. However, even in this case, 90% of the application was modeled with standard elements, which is still an overwhelming fraction of the application.

6.6 Future Work

An extension that would bring an important enhancement to the flashWeb method and to the flashWeb CAWE tool is the support for change management. Web application development is usually characterized by a frequent change of requirements, thus the application's implementation has to be adjusted often. This is not a problem during the actual development phase because the application does not administer real data that is to be treated carefully. Application prototypes may be tested with test data, which in many cases can be simply generated. If changes become necessary, the data and the prototype may be replaced or even thrown away.

However, the circumstances are very different after the Web application has been deployed and taken into service. The most sensitive part of the Web application regarding new functionality is the content storage. Content created by the Web application is usually valuable and cannot be simply thrown away as changes occur. The most difficult situations emerge if the content model of a Web application has to be changed. In such a case, it has to be ensured that content is transferred from the obsolete content storage structure to the new structure. This is usually achieved by running both structures in parallel and transferring the data via manually written scripts. After data transfer, the old structures can be deactivated and the custom scripts become useless. Note however that this process is usually error prone and tiresome.

Therefore, an extension of the flashWeb CAWE tool that provides support for change management would be very valuable and is an object for possible future work. An appropriate solution for this problem could be the support for fine-grained versioning of the models. In case of a necessary adjustment for a particular model element, a new version of the element and an additional mapping between features of the old and the new version could facilitate a smooth migration to the new functionality. To this end, the model editor has to be extended with functionality that ensures an efficient management of different versions of model elements. Furthermore, the code generator has to be able to handle the generation of different implementations according to these different versions and also to produce appropriate migration scripts.

Another feature that is not supported explicitly by flashWeb is the workflow-based design of Web application functionality. Web applications are usually complex systems and, in most cases, they implement some kind of a content management process or offer services that are composed of several tasks executed by the Web application's user. For example, e-business Web applications like online shops and auction portals always use typical workflows that guide the user through an ordering or a bidding process. These processes are usually implemented with a set of user interface screens that are presented to the user in a specific order.

Unfortunately, the flashWeb method does not provide high-level support for the modeling of such processes yet. Of course, it is possible to implement every user interface page of a process using different flashWeb model elements but the navigation between the pages has to be created manually. Also a status indicator that informs the user about the current state of the process needs to be integrated into each page manually.

The last approach for a possible flashWeb extension that is presented here considers the capabilities of the generated application, which may be improved by modifying the code generator plug-in. The goal of this extension is to generate a Web application that allows the user to customize the Web application's user interface in a model-based manner.

Current-day Web applications usually allow its users to customize the user interface of the application in different ways. Typical examples for user customization are the definition of

how many information items of a certain kind are displayed, how this information is presented, or whether the component presenting this information is to be included into a certain user interface page at all. A simple example demonstrating these options is the summary page of an online auction system that informs the user about his current activity. Assuming that this summary page usually includes information about products that have been acquired recently by the user over the auction platform, he may define the number of products that are to be shown, the ordering of these products, e.g., ordered by date or price, or he may even choose not to include this information into the summary page.

Such Web application functionality is of course an added value, however, it also means an additional development effort for the Web application developer. The flashWeb method, which uses a detailed user interface model, offers a unique opportunity to eliminate this development overhead. The Composition/Navigation model of flashWeb may be used to generate a highly dynamic user interface that allows the user to perform a model-based customization of the interface. To this end, the code generator plug-in of the flashWeb CAWE tool must not generate the source code of the user interface as simple HTML with some dynamic content inclusion through scripting, but rather it has to create a sophisticated object structure that represents user interface elements at runtime, which are capable of producing the HTML source code on the fly. With this approach, each user interface component is represented by an appropriate runtime object at the server side that stores all the necessary information to produce the component's presentation. Such a runtime object may be easily replicated to create customized versions of the user interface component and may be easily manipulated not only from the server side but also from the client side through AJAX technology.

Bibliography

[ALW99] C. Anderson, A. Levy, and D. Weld. Declarative web site management with tiramisu. In *ACM SIGMOD Workshop on The Web and Databases (WebDB99), Philadelphia, Pennsylvania, USA, June 3-4*, 1999.

[AML+93] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Alberti. *The Internet Gopher Protocol*. Internet Engineering Task Force - Network Working Group, 1993.

[AMM98] P. Atzeni, G. Mecca, and P. Merialdo. Design and maintenance of data-intensive web sites. In *Proceedings of the 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27*, 1998.

[AP03] P. Atzeni and A. Parente. Specification of web applications with adm-2. *Information modeling for internet applications*, pages 127–143, 2003.

[BCCF03] M. Brambilla, S. Ceri, S. Comai, and P. Fraternali. Specification and design of workflow-driven hypertexts. *Journal of Web Engineering*, 1(2):163–182, 2003.

[BCFC06] A. Bozzon, S. Comai, P. Fraternali, and G. T. Carughi. Conceptual modeling and code generation for rich internet applications. In *Proceedings of the 6th International Conference on Web Engineering (ICWE 2006), Palo Alto, California, USA, July 11-14*, 2006.

[BCFM00] A. Bongio, S. Ceri, P. Fraternali, and A. Maurino. Modeling data entry and operations in webml. In *Third International Workshop on The World Wide Web and Databases (WebDB2000), Dallas, Texas, USA, May 18-19*, 2000.

[BCFM06] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in web applications. *ACM Transactions on Software Engineering and Methodology*, 15(4):360–409, 2006.

[Bec04] K. Beck. *Extreme Programming: die revolutionäre Methode für Softwareentwicklung in kleinen Teams*. Addison Wesley, 2004.

[BGP00] L. Baresi, F. Garzotto, and P. Paolini. From web sites to web applications: New issues for conceptual modeling. In *Proceedings of the 19th International Conference on Conceptual Modeling (ER2000), Salt Lake City, Utah, USA, October 9-12*, 2000.

[BGP01] L. Baresi, F. Garzotto, and P. Paolini. Extending uml for modeling web applications. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34), Maui, Hawaii, January 3-6*, 2001.

[BKM99] H. Baumeister, N. Koch, and L. Mandel. Towards a uml extension for hypermedia design. In *Proceedings of the Second International Conference of The Unified Modeling Language - Beyond the Standard (UML'99), Fort Collins, USA, October 28-30*, 1999.

[BL89] T. Berners-Lee. *Information Management: A Proposal*. World Wide Web Consortium, 1989.

[BL91] T. Berners-Lee. *HTTP Specification 0.9*. World Wide Web Consortium, 1991.

[BL94] T. Berners-Lee. *Universal Resource Identifiers in WWW*. Internet Engineering Task Force - Network Working Group, 1994.

[BL00] T. Berners-Lee. *Weaving the Web*. Harper Collins, 2000.

[BLC93] T. Berners-Lee and D. Connolly. *Hypertext Markup Language (HTML) (Internet Draft)*. IIIR Working Group, 1993.

[BLFG+99] T. Berners-Lee, R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, and P. Leach. *Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force - Network Working Group, 1999.

[Bra06] M. Brambilla. Generation of webml web application models from business process specifications. In *Proceedings of the 6th International Conference on Web Engineering (ICWE 2006), Palo Alto, California, USA, July 11-14*, 2006.

[Bra07a] R. Bradford. *The Art of Computer Networking*. Pearson Education, 2007.

[Bra07b] M. Brambilla. Building semantic web portals with webml. In *Proceedings of the 6th International Conference on Web Engineering (ICWE2007), Como, Italy, July 16-20*, 2007.

[BS04] M. Broy and R. Steinbrüggen. *Modellbildung in der Informatik*. Springer, 2004.

[Bus45] V. Bush. As we may think. *Antlantic Monthly*, 1945.

[CDR08] A. Cicchetti and D. Di Ruscio. Decoupling web application concerns through weaving operations. 70(1):62–86, 2008.

[CDRP06] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Weawing concerns in model based development of data-intensive web applications. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27*, 2006.

[CDS74] V. Cerf, Y. Dalal, and C. Sunshine. *Specification of Internet Transmission Control Program*. Internet Engineering Task Force - Network Working Group, 1974.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming*. Addison Wesley, 2000.

[CFB00] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[CFB+03] S. Ceri, P. Fraternali, A. Bongio, M. Matera, M. Brambilla, and S. Comai. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 2003.

[Che76] P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[CK74] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communication*, 20(5):637–648, 1974.

[Cla03] M. Clark. *Data Networks, IP and the Internet*. Wiley and Sons, 2003.

[Con03] J. Conallen. *Building Web Applications with UML*. Addison-Wesley, 2003.

[CS06] V. Claus and A. Schwill. *Duden Informatik*. Dudenverlag, 2006.

[Dat04] C. J. Date. *Database Systems*. Addison Wesley, 2004.

[DIMG95] A. Diaz, T. Isakowitz, V. Maiorana, and G. Gilabert. Rmc: A tool to design www applications. In *Proceedings of the 4th International World Wide Web Conference (WWW4), Boston, Massachusetts, USA, December 11-14*, 1995.

[Eng62] D. Engelbart. Augmenting human intellect: A conceptual framework. Technical report, Stanford Research Institute, 1962.

[FFK+98] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with strudel: Experiences with a web-site management system. In *Proceedings of the 14ème Journées Bases de Données Avancées, Hammamet, Tunisia, October 26-30*, 1998.

[FFLS97] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):323–382, 1997.

[FFLS98] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Web-site management: The strudel approach. *IEEE Data Engineering Bulletin*, 21(2):14–20, 1998.

[FFLS00] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Declarative specification of web sites with strudel. In *Proceedings of 26th International Conference on Very Large Data Bases, Cairo, Egypt, September 10-14*, 2000.

[FLSY99] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Optimization of run-time management of data intensive web-sites. In *Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, September 7-10*, 1999.

[FP00] P. Fraternali and P. Paolini. Model-driven development of web applications: the autoweb system. *ACM Transactions on Information Systems (TOIS)*, 18(4):323–382, 2000.

[FST99] M. Fernandez, D. Suciu, and I. Tatarinow. Declarative specification of data-intensive web sites. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL'99), Austin, Texas, USA, October 3-5*, 1999.

[GC00] J. Gillies and R. Cailliau. *How the Web Was Born: The Story of the World Wide Web*. Oxford University Press, 2000.

[GCP00] J. Gomez, C. Cachero, and O. Pastor. Extending a conceptual modeling approach to web application modeling. In *Proceedings of the 12th Conference on Advanced Information Systems Engineering (CAiSE00), Stockholm, Sweden, June 5-9*, 2000.

[GHSL02] A. Gu, B. Henderson-Sellers, and D. Lowe. Web modelling languages: the gap between requirements and current exemplars. In *Proceedings of the Eighth Australian World Wide Web Conference (WWW8), Toronto, Canada, May 11-14*, 2002.

[GMP93] F. Garzotto, L. Mainetti, and P. Paolini. Hdm2: Extending the e-r approach to hypermedia application design. In *Proceedings of the 12th International Conference on the Entity-Relationship Approach (ER'93), Arlington, Texas, USA, December 15-17*, 1993.

[GPS93] F. Garzotto, P. Paolini, and D. Schwabe. Hdm - a model-based approach to hypertext application design. *ACM Transactions on Information Systems*, 1(1):1–26, 1993.

[HFV03] G. J. Houben, F. Frasincar, and R. Vdovjak. Hera: Development of semantic web information systems. In *Proceedings of the 3rd International Conference on Web Engineering (ICWE2003), Oviedo, Spain, July 14-18*, 2003.

[HH08] I. Hickson and D. Hyatt. *HTML 5 - Vocabulary and associated APIs for HTML and XHTML*. World Wide Web Consortium, 2008.

[HK00] R. Hennicker and N. Koch. A uml-based methodology for hypermedia design. In *Proceedings of the Third International Conference of The Unified Modeling Language (UML2000), Advancing the Standard, York, UK, October 2-6*, 2000.

[Hun02] C. Hunt. *TCP/IP Network Administration*. O'Reilly Media, 2002.

[IKK97] T. Isakowitz, A. Kamis, and M. Koufaris. Extending the capabilities of rmm: Russian dolls and hypertext. In *Proceedings of the 30th Annual Hawaii International Conference on System Sciences (HICSS-30), Maui, Hawaii, January 7-10*, 1997.

[IKK98] T. Isakowitz, A. Kamis, and M. Koufaris. Reconciling top-down and bottom-up design approaches in rmm. *SIGMIS Database*, 29(4):58–67, 1998.

[Int86] International Organization for Standardization. *Standard Generalized Markup Language (SGML)*, 1986.

[ISB95] T. Isakowitz, E. A. Stohr, and P. Balasubramanian. Rmm: A methodology for structured hypermedia design. *Communications of the ACM*, 38(4):34–44, 1995.

[Jak03] M. Jakob. Ortsbezug für web-inhalte. Master's thesis, Diplomarbeit Nr. 2049, Universität Stuttgart, 2003.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[JGHN05] M. Jakob, M. Grossmann, N. Hönle, and D. Nicklas. Dcbot: Exploring the web as value-added service for location-based applications. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005), Tokyo, Japan, April 5-8*, 2005.

[JGNM05] M. Jakob, M. Grossmann, D. Nicklas, and B. Mitschang. Dcbot: Finding spatial information on the web. In *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA 2005), Beijing, China, April 17-20*, 2005.

[JKK+07] M. Jakob, D. Kiehne, F. Kaiser, H. Schwarz, and S. Beucker. Delphigestütztes szenario-management und -monitoring. Technical report, Fraunhofer IRB Verlag, 2007.

[JKS05] M. Jakob, F. Kaiser, and H. Schwarz. Semafor: A framework for an extensible scenario management system. In *Proceedings of the IEEE International Engineering Management Conference (IEMC2005), St. John's, Newfoundland, September 11-14*, 2005.

[JKSB06] M. Jakob, F. Kaiser, H. Schwarz, and S. Beucker. Generierung von webanwendungen für das innovationsmanagement. *it - Information Technology*, 48(4):225–232, 2006.

[JPMM04] S. Jablonski, I. Petrov, C. Meiler, and U. Mayer. *Guide to Web Application and Plattform Architectures*. Springer, 2004.

[JSKM06a] M. Jakob, H. Schwarz, F. Kaiser, and B. Mitschang. Modeling and generating application logic for data-intensive web applications. In *Proceedings of the 6th International Conference on Web Engineering (ICWE 2006), Palo Alto, California, USA, July 11-14*, 2006.

[JSKM06b] M. Jakob, H. Schwarz, F. Kaiser, and B. Mitschang. Towards an operation model for generated web applications. In *Proceedings of the 6th International Conference on Web engineering - Workshops (MDWE2006), Palo Alto, California, USA, July 11-14*, 2006.

[JSSK07] M. Jakob, O. Schiller, H. Schwarz, and F. Kaiser. flashweb: Graphical modeling of web applications for data management. In *Proceedings of the 26th International Conference on Conceptual Modeling (ER 2007) - Tutorials, posters, panels and industrial contributions, Auckland, New Zealand, November 5-9*, 2007.

[KHKR05] G. Kappel, M. Hitz, E. Kapsammer, and W. Retschitzegger. *UML@Work*. dpunkt Verlag, 2005.

[KJW07] F. Kaiser, M. Jakob, and S. Wiedersheim. Framework-unterstützung für aufwendige websuche. *Datenbank-Spektrum*, 7(23):13–20, 2007.

[KK02a] N. Koch and A. Kraus. The expressive power of uml-based web engineering. In *Proceedings of the Second International Workshop on Web-oriented Software Technology (IWWOST02), Malaga, Spain, June 10*, 2002.

[KK02b] A. Kraus and N. Koch. Generation of web applications from uml models using an xml publishing framework. In *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT)*, 2002.

[KKCM03] A. Koch, A. Kraus, C. Cachero, and S. Melia. Modeling web business processes with oo-h and uwe. In *Proceedings of the Third International Workshop on Web-oriented Software Technology (IWWOST03), Oviedo, Spain, Juli 15*, 2003.

[KKCM04] A. Kraus, N. Koch, C. Cachero, and S. Melia. Integration of business processes in web application models. *Journal of Web Engineering*, 3(1):22–49, 2004.

[KKMZ03] A. Knapp, N. Koch, F. Moser, and G. Zhang. Argouwe: A case tool for web applications. In *First International Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE2003), Geneva, Switzerland, September 2-5*, 2003.

[KKZH04] A. Knapp, N. Koch, G. Zhang, and H. M. Hassler. Modeling business processes in web applications with argouwe. In *Proceedings of 7th International Conference on The Unified Modelling Language: Modelling Languages and Applications (UML2004), Lisbon, Portugal, October 11-15*, 2004.

[KL86] B. Kantor and P. Lapsley. *Network News Transfer Protocol*. Internet Engineering Task Force - Network Working Group, 1986.

[KM00] D. Kristol and L. Montulli. *HTTP State Management Mechanism*. Internet Engineering Task Force - Network Working Group, 2000.

[KPRR06] G. Kappel, B. Pröll, S. Reich, and W. Retschitzegger. *Web Engineering*. John Wiley and Sons, 2006.

[KSJ06] F. Kaiser, H. Schwarz, and M. Jakob. Finding experts on the web. In *Proceedings of the 2nd International Conference on Web Information Systems and Technologies, Setúbal, Portugal, April 11-13*, 2006.

[KSJ07] F. Kaiser, H. Schwarz, and M. Jakob. Expose: Searching the web for expertise. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Amsterdam, The Netherlands, July 23-27*, 2007.

[KZE06] N. Koch, G. Zhang, and M. Escalona. Model transformations from requirements to web system design. In *Proceedings of the 6th International Conference on Web Engineering (ICWE 2006), Palo Alto, California, USA, July 11-14*, 2006.

[LF05] M. Lang and B. Fitzgerald. Hypermedia systems development practices: A survey. *IEEE Software*, 20(2):68–75, 2005.

[LH99] D. Lowe and W. Hall. *Hypermedia and the Web*. John Wiley and Sons, 1999.

[LL07] J. Ludewig and H. Lichter. *Software Engineering*. dpunkt.verlag, 2007.

[LS03] F. Lima and D. Schwabe. Application modeling for the semantic web. In *Proceedings of the 1st Latin American Web Congress (LA-WEB 2003), Sanitago, Chile, November 10-12*, 2003.

[MAM+00] P. Merialdo, P. Atzeni, M. Magnante, G. Mecca, and M. Pecorone. Homer: a model-based case tool for data-intensive web sites. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA, May 16-18*, 2000.

[MAM03] P. Merialdo, P. Atzeni, and G. Mecca. Design and development of data-intensive web sites: The araneus approach. *ACM Transactions on Internet Technology (TOIT)*, 3(1), 2003.

[MD01] S. Murugesan and Y. Deshpande. *Web Engineering: Managing Diversity and Complexity of Web Application Development*. Springer Verlag, 2001.

[MDHG99] S. Murugesan, Y. Deshpande, S. Hansen, and A. Ginige. Web engineering: a new discipline for development of web-based systems. In *Proceedings of the 1st ICSE Workshop on Web Engineering, Los Angeles, CA, USA, May 16-22*, 1999.

[MPSS99] C. Moschovitis, H. Poole, T. Schuyler, and T. Senft. *The History of the Internet*. ABC-CLIO, 1999.

[NS06] D. A. Nunes and D. Schwabe. Rapid prototyping of web applications combining domain specific languages and model driven design. In *Proceedings of the 6th International Conference on Web Engineering (ICWE 2006), Palo Alto, California, USA, July 11-14*, 2006.

[PAF01a] O. Pastor, S. Abrahao, and J. Fons. Building e-commerce applications from object-oriented conceptual models. *SIGecom Exchanges*, 2(2):28–36, 2001.

[PAF01b] O. Pastor, S. Abrahao, and J. Fons. An object-oriented approach to automate web applications development. In *Proceedings of the 2nd International Conference on Electronic Commerce and Web Technologies (EC-Web 2001), Munich, Germany, September 4-6*, 2001.

[PFG+94] M. Pierre, J. Fullton, J. Goldman, B. Kahle, J. Kunze, H. Morris, and F. Schiettecatte. *WAIS over Z39.50*. Internet Engineering Task Force - Network Working Group, 1994.

[PGIP01] O. Pastor, J. Gomez, E. Instfran, and V. Pelechano. The oo-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. *Information Systems*, 26(7):507–534, 2001.

[PIP+97] O. Pastor, E. Instfran, V. Pelechano, J. Romero, and J. Merseguer. Oo-method: An oo software production environment combining conventional and formal methods. In *Proceedings of the 9th Conference on Advanced Information Systems Engineering (CAiSE97), Barcelona, Spain, June 16-20*, 1997.

[PL09] R. Pressman and D. Lowe. *Web Engineering - A Practitioner's Approach*. McGraw-Hill, 2009.

[Pow98] T. Powell. *Web Engineering*. Prentice Hall, 1998.

[Pre05] R. Pressman. *Software Engineering - A Practitioner's Approach*. McGraw-Hill, 2005.

[RLHJ99] D. Ragett, A. Le Hors, and I. Jacobs. *HTML 4.01 Specification*. World Wide Web Consortium, 1999.

[Rod05] J. Rode. Web application development by nonprogrammers: User-centered design of an end-user web development tool. Virginia Polytechnic Institute and State University, 2005.

[Roy87] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering (ICSE'87), Monterey, CA, USA, March 30 - April 2*, 1987.

[RP06] P. Rechenberg and G. Pomberger. *Informatik Handbuch*. Carl Hanser Verlag, 2006.

[RPSO08] G. Rossi, O. Pastor, D. Schwabe, and L. Olisina. *Web Engineering - Modelling and Implementing Web Applications*. Springer, 2008.

[Sch07] O. Schiller. Entwicklung eines cawe-werkzeugs für die modellbasierte entwicklung von webanwendungen. Master's thesis, Universität Stuttgart, 2007.

[SKS06] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 2006.

[SL03] H. Schmid and F. Lyardet. Engineering business processes for web applications modeling and navigation issues. In *Proceedings of the Third International Workshop on Web-oriented Software Technology (IWWOST03), Oviedo, Spain, Juli 15*, 2003.

[SM98] W. Sonnenreich and T. Macinta. *Guide to Search Engines*. John Wiley and Sons, 1998.

[Som07] I. Sommerville. *Software Engineering*. Pearson Studium, 2007.

[SPM99] D. Schwabe, R. Pontes, and I Moura. Oohdm-web: an environment for implementation of hypermedia applications in the www. *ACM SIGWEB Newsletter*, 8(2), 1999.

[SR95a] D. Schwabe and G. Rossi. Building hypermedia applications as navigational views of information models. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS-28), Maui, Hawaii, January 3-6*, 1995.

[SR95b] D. Schwabe and G. Rossi. The object-oriented hypermedia design model. *Communications of the ACM*, 38(4):45–46, 1995.

[SR98] D. Schwabe and G. Rossi. An object oriented approach to web-based applications design. *Theory and Practice of Object Systems*, 4(4):207–225, 1998.

[SRB96] D. Schwabe, G. Rossi, and S. Barbosa. Systematic hypermedia application design with oohdm. In *Proceedings of the Seventh ACM Conference on Hypertext, Washington DC, USA, March 16-20*, 1996.

[SSML04] D. Schwabe, G. Szundy, S. S. Moura, and F. Lima. Design and implementation of semantic web applications. In *Proceedings of the 13. WWW 2004 Workshop on Application Design, Development and Implementation Issues in the Semantic Web: New York, USA, May 18*, 2004.

[SV05] T. Stahl and M. Völter. *Modellgetriebene Softwareentwicklung*. dpunkt.verlag, 2005.

[Tab06] P. Tabeling. *Softwaresysteme und ihre Modellierung*. Springer, 2006.

[Tan03] A. Tanenbaum. *Computer Networks*. Pearson Education, 2003.

[TC03] O. Troyer and S. Casteleyn. Modeling complex processes for web applications using wsdm. In *Proceedings of the Third International Workshop on Web-oriented Software Technology (IWWOST03), Oviedo, Spain, Juli 15*, 2003.

[TH07] D. Thomas and D. H. Hansson. *Agile web development with Rails*. Raleigh, NC, 2007.

[Tur02] V. Turau. A framework for automatic generation of web-based data entry applications based on xml. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC), Madrid, Spain, March 10-14*, 2002.

[Uni81] University of Southern California. *Internet Protocol*, 1981.

[URI92] World wide web. `http://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html`, 1992. Online, accessed on 2008-04-04.

[URI01] Daml+oil. `http://www.daml.org/2001/03/daml+oil-index.html`, 2001. Online, accessed on 2008-12-18.

[URI03] Information retrieval (z39.50): Application service definition and protocol specification. `http://www.loc.gov/z3950/agency/document.html`, 2003. Online, accessed on 2008-04-07.

[URI04a] Resource description framework (rdf). `http://www.w3.org/RDF`, 2004. Online, accessed on 2008-12-18.

[URI04b] Web ontology language (owl). `http://www.w3.org/2004/OWL`, 2004. Online, accessed on 2008-12-18.

[URI05] Ajax: A new approach to web applications. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, 2005. Online, accessed on 2008-09-01.

[URI08a] Apache cocoon. `http://cocoon.apache.org`, 2008. Online, accessed on 2008-12-18.

[URI08b] Apache cocoon. `http://www.webratio.com`, 2008. Online, accessed on 2008-12-18.

[URI08c] Cern web page. `http://public.web.cern.ch/Public/Welcome.html`, 2008. Online, accessed on 2008-04-04.

[URI08d] Google docs. `http://docs.google.com`, 2008. Online, accessed on 2008-04-10.

[URI08e] Html 4.01 strict dtd. `http://www.w3.org/TR/html4/strict.dtd`, 2008. Online, accessed on 2008-10-15.

[URI08f] Internet assigned numbers authority. `http://www.iana.org`, 2008. Online, accessed on 2008-10-14.

[URI08g] Internet domain survey. `http://www.isc.org/ops/ds`, 2008. Online, accessed on 2008-08-29.

[URI08h] Internet world stats. `http://www.internetworldstats.com`, 2008. Online, accessed on 2008-08-29.

[URI08i] Manifesto for agile software development. `http://agilemanifesto.org`, 2008. Online, accessed on 2008-12-17.

[URI08j] Ruby on rails. `http://www.rubyonrails.org`, 2008. Online, accessed on 2008-12-18.

[URI08k] Sesame. `http://www.openrdf.org`, 2008. Online, accessed on 2008-12-18.

[URI08l] Web server survey (august 2008). `http://news.netcraft.com/archives/web_server_survey.html`, 2008. Online, accessed on 2008-09-01.

[URI08m] The website of the world's first-ever web server. `http://info.cern.ch`, 2008. Online, accessed on 2008-04-03.

[URI08n] Wikipedia: The free encyclopedia. `http://www.wikipedia.org`, 2008. Online, accessed on 2008-09-01.

[URI09] The eclipse project web site. `http://www.eclipse.org`, 2009. Online, accessed on 2009-04-21.

[vdSHBC06] K. van der Sluijs, G. J. Houben, J. Broekstra, and S. Casteleyn. Hera-s: web design using sesame. In *Proceedings of the 6th International Conference on Web Engineering (ICWE2006), Palo Alto, California, USA, July 11-14*, 2006.

[VSDS00] P. Vilain, D. Schwabe, and C. De Souza. A diagrammatic tool for representing user interaction. In *Proceedings of the Third International Conference of The Unified Modeling Language (UML2000), Advancing the Standard, York, UK, October 2-6*, 2000.

[vW08] P. von Weitershausen. *Web Component Development with Zope 3*. Springer, 2008.

[War03] J. Warmer. *The Object Constraint Language*. Addison Wesley, 2003.

[Wöh04] H. Wöhr. *Web - Technologien*. dpunkt.verlag, 2004.

[ZD83] H. Zimmermann and J. Day. The osi reference model. *Proceedings of the IEEE*, 71(12):1334– 1340, 1983.