Vrije Universiteit Brussel

Faculteit Wetenschappen
Vakgroep Computerwetenschappen

Implementation Generation for WSDM using Web Applications Framework

Proefschrift ingediend met het oog op het behalen van de graad van Master in de Ingenieurswetenschappen: Computerwetenschappen

Kevin Van Wilder

Promotor:	Prof. Dr. Olga De Troyer
Begeleiders:	Dr. Sven Casteleyn

29 mei 2009

Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science

Implementation Generation for WSDM using Web Applications Framework

Graduation thesis submitted with intention to obtain the degree of Master in Engineering: Computer Science

Kevin Van Wilder

Promotor:	Prof. Dr. Olga De Troyer
Advisors:	Dr. Sven Casteleyn

May 29, 2009

Nederlandstalige Samenvatting

In deze thesis presenteren we het onderzoek en de implementatie van een code generatie tool voor de Web Site Design Methode. We spitsen ons toe op twee invalshoeken.

Enerzijds onderzoeken we verscheidene web frameworks zoals Django, Ruby on Rails en CakePHP voor inzicht te krijgen in de vereisten voor het genereren naar zulke frameworks. Daarna wordt de implementatie voorgesteld van de code generator die een WSDM beschrijving van een site neemt als input en automatisch de code genereert voor het Django web framework.

Anderzijds onderzoeken we de verscheidene overige methodes zoals OOH, OOHDM, UWE en WebML samen met hun respectievelijke tools om websites te genereren.

De case study en de verscheidene gelijkenissen met de andere code generatie tools bevestigen de keuzes gemaakt tijdens het ontwikkelproces.

Abstract

In this dissertation, we present the research and implementation of a code generation tool for the Web Site Design Method. Two avenues of website generation are examined.

On the one side, we investigate various web frameworks such as Django, Ruby on Rails and CakePHP to provide us with insights in the requirements for generation towards these web frameworks. We then implement a code generation tool which reads in the WSDM description of a web site and automatically generates the web site using the Django web framework.

On the other side, we investigate various other methodologies such as OOH, OOHDM, UWE and WebML along with their respective tools for generating websites.

A case study and various commonalities between other code generation tools validate choices made during development.

Acknowledgements

"I love deadlines. I like the whooshing sound they make as they fly by."
- Douglas Adams

This document would not be what it is today without the outstanding support of various people. Therefore I would like to express my gratitude towards:

Prof. Dr. Olga De Troyer for promoting this dissertation, giving me the opportunity to leave my mark on it and for the countless proof-reads and suggestions during the completion of it.

My advisor Sven Casteleyn for his great support, insight and honest remarks.

My girlfriend Freija Van Megroot, for her constant support, her care and love. I will make up for all those weekends spent in front of the computer working on my dissertation!

My parents, for giving me the opportunity to seize every chance in life and allowing me to study at the VUB.

Also a thank you to Bart Verhaegen, for giving me Frontpage '98 when I was 12, which started my interest in web development.

My friend Dries Harnie, for always being there to help me out in life and allowing me to bounce ideas off of him. As well as my American friend Robert Tiller for sharing his life experiences with me and offering me advice when I needed it. Also I would like to thank his wife Terri for proof-reading and helping me with grammar and style. Finally a thanks to Tom Strickx for his friendship and showing interest in my thesis.

I dedicate this dissertation to my grandmother Ludovica "Wiske" De Pauw.

Enjoy!

Kevin Van Wilder
Affligem, 2009

Contents

Chapter 1

Introduction

1.1 Context

’Web Engineering is a discipline concerned with the establishment and use of sound scientific engineering and development principles and systematic approaches for the successful development, deployment and maintenance of Web-based systems and applications. It is often compared with software engineering, and indeed, Web Engineering applies many techniques and tools from this field. However, the specificities of the Web and its applications, e.g. constant evolution of Websites, navigation-based user interface (UI), specific technologies and (mark-up) languages, client-server architecture, etc. makes Web Engineering a distinct discipline, with specific challenges and solutions.

The Web today is evolving at a dazzling pace. Since the traditional Web, now called Web 1.0, we saw the rise of Web 2.0 and the Semantic Web. Web 2.0 was actually nothing new, but rather a new application of existing technologies oriented towards user input and cooperation. Where in Web 1.0 the separation between content provider (i.e. the web server) and user (i.e. the user) was clear, in Web 2.0 this border became vague. Users provide content (Wikipedia), express themselves on social Websites such as MySpace, tag (Flickr) and interact (Friendster, Facebook, Last.fm, Twitter). It’s not so much the technology that has changed, but what people do with it. The Semantic Web is all about the study of language meaning. Where Web 1.0 and 2.0 are made for people, the Semantic Web is made for machines. Agents crawling the Web, understanding and interpreting the information that is available, reasoning about it and drawing conclusions. The Semantic Web is truly a Web of meaningful content, and promises a revolution on how we use and handle information.

Even now, the Web is evolving in new directions. More and more users are accessing the Web at any time, from any place and with any device, resulting in the so-called Mobile Web. Increasingly powerful mobile phones, mobile organizers and game consoles with Web capabilities have triggered this evolution. Lately, with the development of cheap and identifiable tags (i.e. Radiofrequency Identification (RFID) and Near Field Communication (NFC) technology), the Internet of Things is coming into existence.

The Web and Information Systems Engineering Lab (WISE) at the Free University Brussels has been present in the Web Engineering community since the early days of the WWW. In 1998, WISE developed one of the first so called Web Engineering methods, the Web Site Design Method (WSDM), which is well recognized in the scientific community. WSDM provides a systematic way to develop and deploy high quality Web applications, taking into account issues such as usability, accessibility, adaptation, personalization, device dependency, localization and semantic annotations" [1]

1.2 Purpose

All WSDM phases but the last are explained in detail. The final phase of the WSDM method is the so-called implementation phase. The actual implementation can be generated automatically from the information collected during the different design phases by means of the different design models, i.e. object chunks, navigational model, site structure model, page models, logical data models and mappings. These models will be explained in more detail.

During its life-span, the WSDM obtained an implementation for this "implementation phase": a transformation pipeline using XSLT. This implementation takes as input the object chunks (with corresponding data source mapping), navigational, site structure, style and template and page models, and outputs the actual implementation for the chosen platform and implementation language. This implementation proved to be rather restrictive, due to the limitations of XSLT transformations. Depending on the complexity of the models, they could require interpretation as well as transformation. Another disadvantage of the XSLT transformation pipeline is the creation of static pages. It would be interesting to be able to create dynamic web pages that will extract information from a database and present it.

This dissertation presents a new implementation for an automatic code generation process for interpreting WSDM models. This implementation overcomes the limitations of the previous by applying a fully interpretive approach. By using a modular framework, the generation tool provides the flexibility for generating code for multiple, freely available web application frameworks.

1.3 Outline of Dissertation

This dissertation is structured by explaining the meaning and form of the input information, followed by the specifications of the various output formats. An in-depth description of the generation tool follows these specifications. A case study follows suit and explains the generated results. Before any conclusions are drawn, we also discuss the various other methodologies and their code generators in our field of research.

[1] Text taken from thesis description page - http://wise.vub.ac.be/

Chapter 2 introduces in-depth the Web Site Design Method (WSDM). The five phases and their respective models will be introduced, followed by an explanation how these were formalized using OWL to enable formal specification of a web site using the WSDM models.

After presenting the input models, the Django, Ruby on Rails and CakePHP web application frameworks are presented in chapter 3 as our possible output results.

Chapter 4 introduces WSDMtool as the new code generation tool, by using the models described in chapter 2 and generating towards the Django web framework from chapter 3. The architecture and important decisions made during development will be presented.

Chapter 5 provides an overview and proof-of-concept of the flexibility of WSDMtool through the application of the Conference Review System case study.

We conclude this dissertation in Chapter 7, which provides a summary of our work and contributions, followed by an overview of future research directions and our final concluding remarks.

1.4 Notable Contributions

In this dissertation we present the following original contributions:

- An evaluation of the WSDM methodology as a technique for automatically generating web sites, as well as exploration of the methodology itself.
- An alternate approach to generating dynamic web sites using code generation instead of the already available XML/XSLT pipeline transformation process.
- A tool that provides a coupling between WSDM and pre-existing web application frameworks.
- A case study which demonstrates how WSDM can be used to automatically generate a fully working web site using the OWL models.
- Providing a WSDM ontology extension required for an unambiguous generation process.

Chapter 2

Web Site Design Method Overview

The Web Site Design Method (WSDM) was introduced [22] in 1998, making it one of the pioneer web modeling approaches. The methodology has continuously evolved and has been re-named [21] in 2007 as Web Semantics Design method. The methodology has been extended to allow more recent concepts such as localization [19], adaptivity [15], accessibility [38] and semantic annotations [16]. It differs from other web modeling methodologies by its orientation towards audience-driven websites, rather than the more common content-driven approaches.

WSDM has two basic characteristics:

- An audience driven approach which can be exemplified by the very early audience modeling phase where the emphasis is placed upon the target audience of the site.
- An explicit conceptual design phase describing a process to generate a pure conceptual design free of implementation and presentation details. This allows the possibility for different presentations for different users or hardware platforms.

2.1 WSDM Phases

WSDM is divided into 5 phases: mission statement specification, audience modeling, conceptual design, implementation design and a final implementation phase.

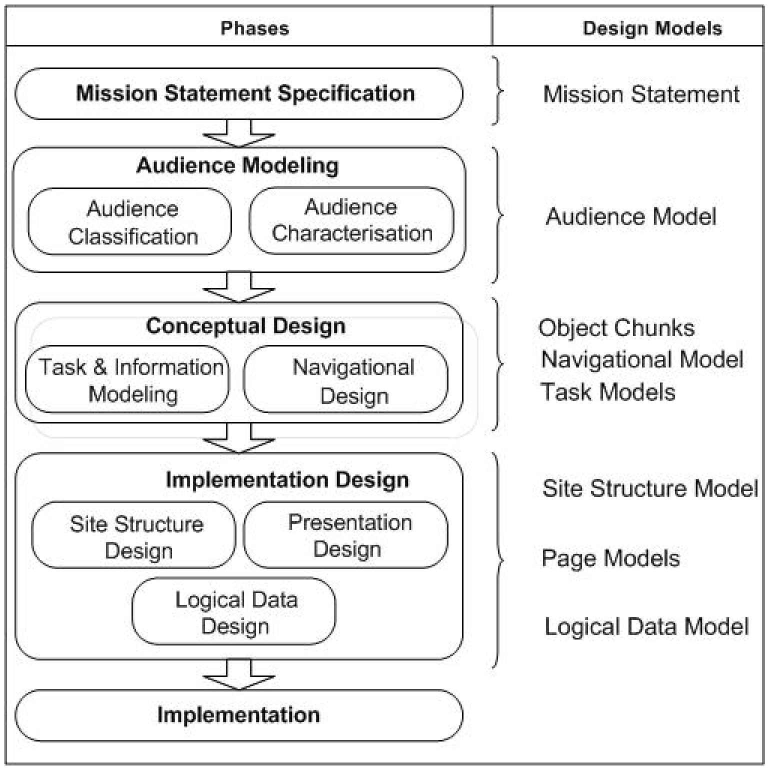


Figure 2.1: WSDM Methodology – Phase overview [22]

2.1.1 Mission Statement Specification

The first phase in the methodology describes the specification of the mission statement for a website. This statement expresses various aspects of the website, such as the purpose, the subject and the targeted demographic. This mission statement is important since it clearly outlines the scope and the borders of the website that needs to be created and how it should be used. It furthermore serves as a document which can be used for validating the functionality. There are no constraints on how this document is written but it is normally done as prose.

2.1.2 Audience Modeling

Based on the information obtained from the previous Mission Statement phase an Audience modeling phase is performed. This is a two-step process containing an Audience Classification and an Audience Characterization sub-phase.

In the Audience Classification phase, the target audience will be divided into different types of users called Audience Classes. Each person of an Audience Class has the same information and functional requirements as to how they will perceive the web site.

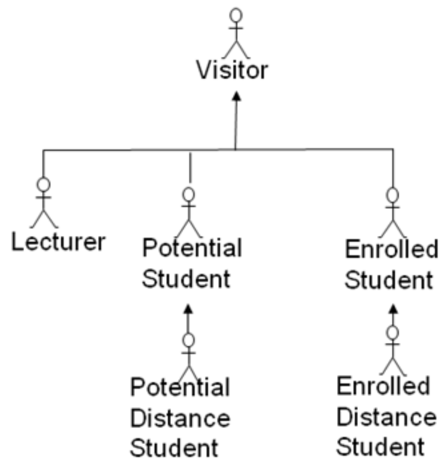


Figure 2.2: Audience Classes

After these classes are derived (Figure 2.2) the next sub-phase, called Audience Characterization, is initiated. Here the Audience Classes will be extended by an informal description of their informational, functional, navigational and usability requirements as well as their characteristics, such as age-group, computer knowledge and language.

2.1.3 Conceptual Design

Next the methodology enters the Conceptual Design [20] phase. The goal of this phase is to turn the informal requirements obtained from the previous phase into high-level, unambiguous, formal descriptions which can later be used in the implementation process of the web site.

This phase is divided into two sub-phases: Task Modeling and Navigational Design. [20] describes this phase as modeling the conceptual "what and how?" of the website, where the Task Modeling phase focuses on the "what?" and the Navigational Design phase on the "how?".

In contrast to content driven web site methodologies, WSDM does not start by creating an overall conceptual schema or Universe of Discourse but rather tiny conceptual schemas, called Object Chunks for each functional requirement in each Audience Class. Combining all Object Chunks will however result in the same Universe of Discourse as would be defined in other methodologies. These Object Chunks are created in the Task Modeling phase using an adapted modeling technique based on Concurrent Task Tree (CTT) [37] to describe their workflow. CTT is a tree-like methodology where each non-leaf node is an Abstraction Task and all leafs are either Interaction or Application Tasks.

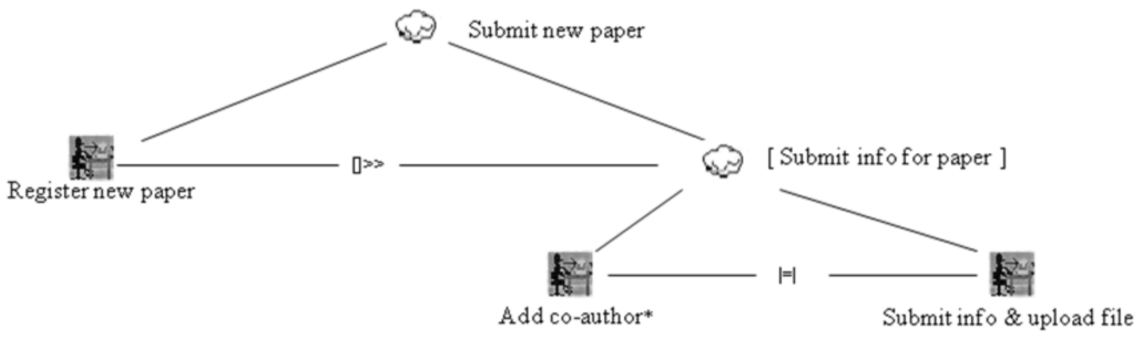


Figure 2.3: A decomposed task using CTT

Figure 2.3 provides an example[1] of Task Modeling:

- The task "Submit new paper" is decomposed into an Interaction task "Register new paper" and "Submit info for paper". Upon completion of the first task, the second task is enabled and information is passed from the first task to the second task.
- Similarly the "Submit info for paper" task is divided into two Interaction tasks "Add co-author" and "Submit info & upload file". These two tasks, marked by "|=|" are order independent.

Once the task models are created, each elementary task is associated with an Object Chunk (see example in Figure 2.4) which will fulfill this task. These chunks are modeled using an extended version [20] of Object Role Modeling (ORM) to allow the annotation of web-specific functionality. This extension allows, for instance, supporting user interactions. An example of such a user interaction is the requirement to fill out a form. To decide which information the user needs to fill in, an object type is annotated by "*p = ?", specifying that input is required for that object type and that the instantiation should be kept in a "variable" called 'p'. The "Paper" concept is highlighted in order to visualize it as the main concept in the object chunk. WSDM is currently the only methodology to allow the conceptual modeling of a web site through the usage of an ORM notation.

[1] Images 2.2, 2.3 and 2.5 obtained from the lecture slides for Web Engineering

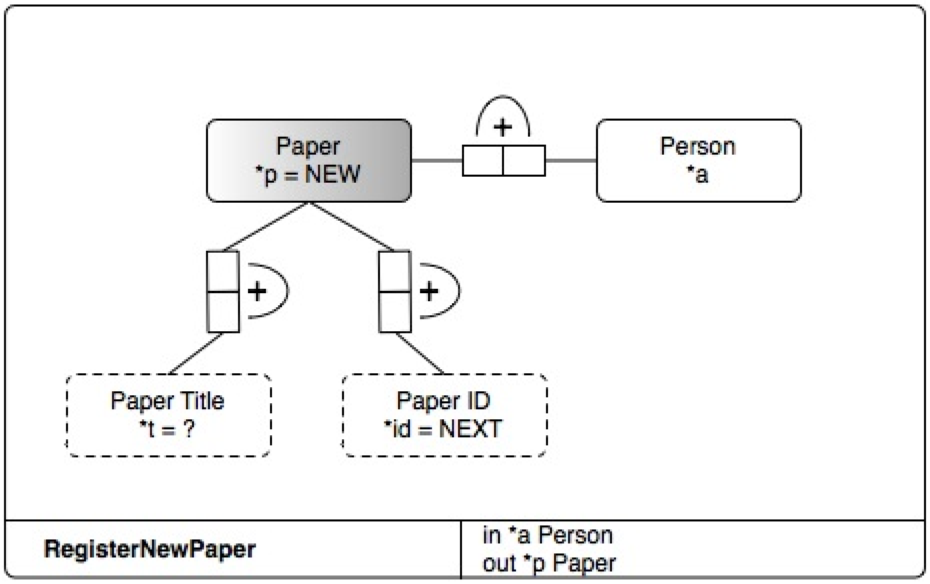


Figure 2.4: "RegisterNewPaper" Object Chunk

To allow communication between multiple Object Chunks they are annotated by input and output parameters, specifying the Concept (i.e. Object Type) that is being passed through.

The next sub-phase, Navigation Design, creates the navigational structure of the site. For each audience class a Navigation Track is created. This track can be considered as a sub-division of the web site that a specific audience class will be able to visit. The tracks are represented by a graph: Nodes connected through Links. Nodes are placeholders containing one or more Object Chunks to specify their functionality. This way, Object Chunks can be reused in other locations in the web site that require the same functionality. This is still part of the conceptual design phase, which means that these nodes do not represent the pages.

The choice of the type of navigational link is decided by inspecting the task models [18]. These links can have any cardinality, e.g. one-to-one, many-to-one, etc. and are either unidirectional or bidirectional.

There are four distinct categories of Links:

- Project Logic Links are used to connect nodes that belong to a single task navigation model and express the workflow or process logic in the task model modeled by the temporal CTT relations.

- Structural Links describe how a visitor navigates from one user task to another in the web site.

- Navigational Aid Links can be considered as secondary Structural Links and are added to aid the visitor navigating through the web site. A typical implementation for these links could be breadcrumb navigation or the typical "back to the home page" link on each page.

- Semantical Links are links which do not connect Nodes but rather Concepts in Object Chunks. Using the Amazon web site example in [18] the object chunk "View Item Info" showing the information of a book can have a semantic relationship with "View Author Info". An implementation of this would generate a hyperlink over the author's name to a page containing information about the author.

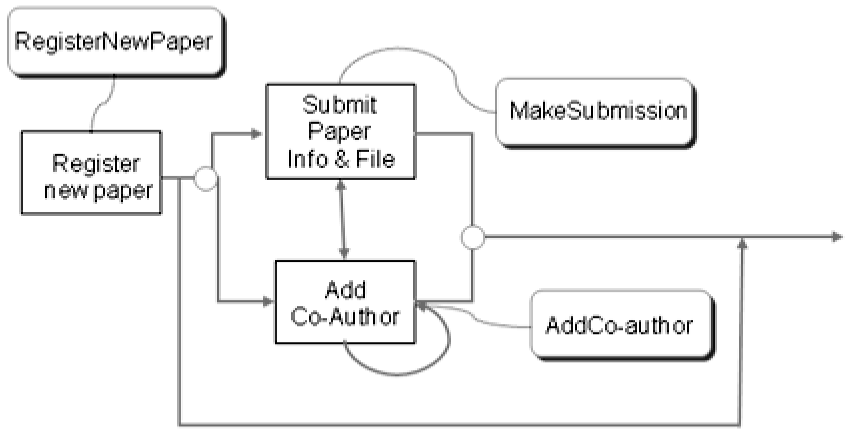


Figure 2.5: "Update Paper Submission" Navigation Track

Figure 2.5 describes the "Update Paper Submission" navigation track. The track contains three nodes: "Update Paper Info and File", "Delete Co-authors" and "Add Co-author". Every node contains one object chunk.

The end of the conceptual modeling phase results in a conceptual navigation path linking the various user tasks together as a whole. At this point the website has been formalized in context of its functional and navigational requirements.

2.1.4 Implementation Design

This phase will extend the conceptual models with extra information required for implementation on a specific platform. This happens in a three-step process: Site Structure Design, Presentation Design and Logical Data Design.

The Site Structure Design phase will decide which nodes are placed on the same page. For each device (e.g., computer, PDA) targeted, a different decision can be made. Because some devices have restrictions such as a small viewing resolution or low bandwidth it is important to put few nodes on the same page, thus restricting the amount of information present on each page and limiting the information that needs to be downloaded on the device.

During the Presentation Design phase, the layout, such as positioning and styling, of the pages will be described. Templates are designed initally. These can be accompanied by Cascading Stylesheets (CSS), which is the current standard for styling.

The final Logical Data Design phase is only needed for data intensive web sites. The conceptual models obtained during the Conceptual Design phase can now be mapped [16] upon new or existing data sources, such as relational databases or triple stores.

2.1.5 Implementation

The final part in the methodology is the actual implementation of the web site modeled thus far. The automation of this particular phase is the subject of this dissertation.

2.2 Formal Specification for Interpretation by Machines

Until now we have explained the creation of a WSDM specification for a web site through the means of graphical notations, such as the ORM-diagrams in the Object Chunks. In order to automatically generate web sites using these specifications, a notation has to be introduced which can be interpreted by software. To allow further processing of the different WSDM models expressing the content and structure of a website, the models are stored into the WSDM Ontology[2]. The ontology contains a set of meta-models which are used to design site-specific models and populate them. A choice was made to express the WSDM models using the Web Ontology Language (OWL). This provided the opportunity to define the meta-models of WSDM models as well as their semantics. Having a formal and semantic representation of these models made it possible to create a tool which could be able to interpret these models and generate a website matching the provided specifications.

2.2.1 Basic Concepts

Before we introduce the formal notation of WSDM, a couple of key aspects will be introduced.

2.2.1.1 The Semantic Web

The original Web dates back to 1989, when it was initially introduced as a universal means to exchange information between universities by Tim Berners-Lee and his associate Robert Cailliau [10]. This World Wide Web (WWW) was made up of wiki-like documents, linked together by hypertext.

The technology of the WWW was based on three parts:

- Uniform Resource Locator (URL): a method of addressing resources, in this case web sites and pages, without the need for specifying the physical location of the resource.

[2] http://wise.vub.ac.be/ontologies/WSDMOntology.owl

- Hypertext Markup Language (HTML): a markup language designed to specify information of various multimedia types, such as text or video. It also contains markup to specify hyperlinks based on the previously mentioned URL addressing method.

- Hypertext Transfer Protocol (HTTP): a protocol designed to exchange information between a web client and a web server.

The Semantic Web (often referred to as Web 3.0) extends the original Web, also called the Hypertext Web, with ideas that were already present during its original design [10], but were never implemented thus far. The intent was to create a semantically grounded web, where computers would be able to understand any given site as good as a human would. Day to day business would be partially fulfilled by communicating autonomous computer agents [10].

These ideas brought forth several new technologies such as RDF, OWL, and many more depicted in figure 2.6. This figure, presented by Tim Berners-Lee [9] represents the hierarchical structure how the various technologies extend and complement each other. The first and second layers are the original Web as we have known it before the Semantic Web.

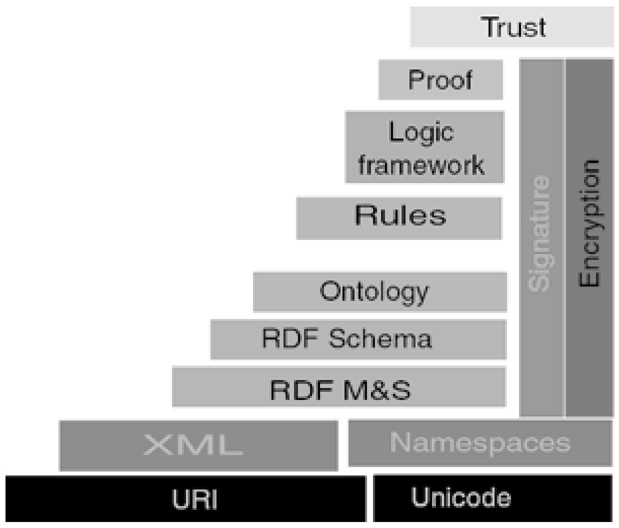


Figure 2.6: Semantic Web Technology Stack [9]

- XML only provides syntax to structure documents.

- XML Schema (XMLS) provides a platform for restricting the structure of XML and also extends it with data types.

- RDF provides the possibility to create resources and specify relations between them.

- RDF Schema (RDFS) provides a vocabulary for describing properties and classes of RDF resources.

- OWL extends RDF Schema even further by adding more vocabulary for descriptions such as relationships between classes, cardinality, equality, typing and characteristics of properties, etc.

2.2.1.2 Web Ontology Language (OWL)

As described in [25] OWL is intended to be used when information contained in files is processed by applications not by humans. It is used to ground the meaning of terms and their relationships in a given vocabulary, also called Ontology.

OWL, a revision of the DAML+OIL web ontology language, provides a richer set of tools to express meaning and semantics than XML, RDF and RDF-S.

The OWL language has been divided into three sublanguages with increasingly expressive specifications. This has been done to suit the different needs of users and communities.

- OWL Lite contains a classification hierarchy and provides simple constraints such as cardinality constraints for values of 1 and 0. The Lite version is mostly used as a transition platform for other taxonomies.
- OWL DL (Description Logic) provides a richer set of features for expressions but remains computational complete. This means that all conclusions are still computable.
- OWL Full is the all-encompassing language set which provides maximum expressiveness and syntactic freedom. This however, comes at a price, as the Full language is computational incomplete.

With these sublanguages in mind, OWL Full can be considered an extension of RDF-S, whilst both OWL Lite and OWL DL should be considered as a restriction.

2.2.2 WSDM Ontology

As an aid to fully explain the meta-model of the WSDM Ontology, we will visually and semantically [26] represent it using the VisioOWL [27] application.

2.2.2.1 Information and Functionality Modeling

The wsdm.I-F-ModelingConcept class is the parent for everything related to information and functionality modeling, to be more precise: this will specify a notation for the object chunks.

wsdm.ObjectChunk (meta model depicted in figure 2.9) is a subclass of wsdm.I-F-ModelingConcept containing the full specification of an Object Chunk as it would be described in the visual representation.

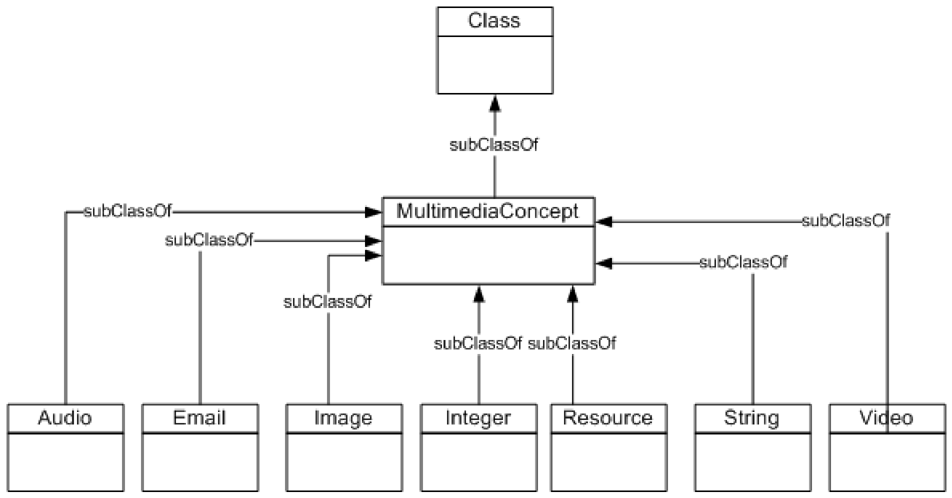


Figure 2.7: OWL representation of a wsdm.MultimediaConcept meta model

The relations between the various concept classes in ORM can be described as a list of statements (meta model depicted in figure 2.8) in the typical form of 3-tuples (Subject, Property, Object), e.g. (Person, hasName, Name).

The bi-directional roles in ORM can be translated to one wsdm.Statement. To specify the inverse relationship, i.e. the other direction, it is only necessary to define the rdfs.inverseOf of the wsdm.Property. If the inverse of the "hasName" property would be "isNameOf", then our statement would be: (Name, isNameOf, Person).

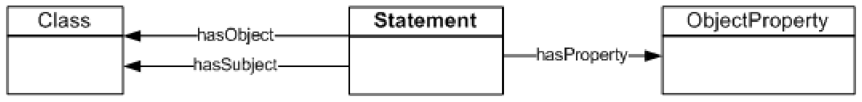


Figure 2.8: OWL representation of a wsdm.Statement meta model

A wsdm.Class (meta model depicted in figure 2.7) can be either a Lexical Object Types (LOT) or Non-Lexical Object Types (NOLOT), which need to be differentiated in the ontology as well.

- A NOLOT will be superclassed by wsdm.Class.
- A LOT will be superclassed by one of the seven already present subclasses of wsdm.MultimediaConcept.

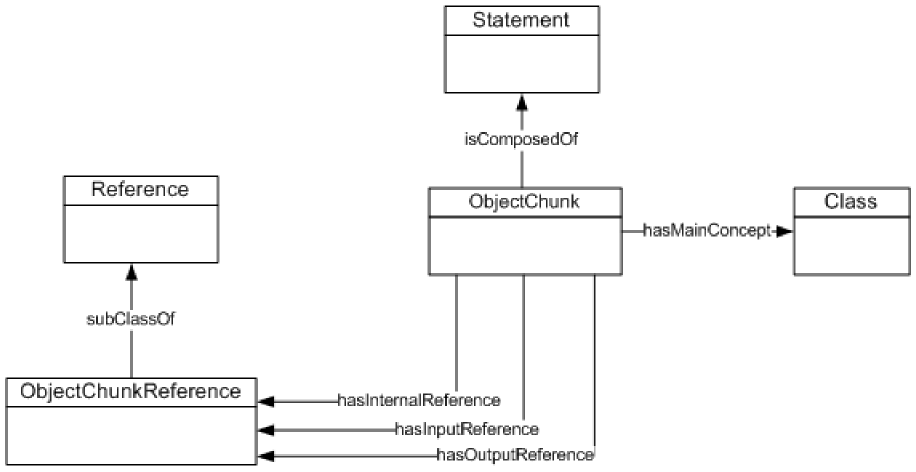


Figure 2.9: OWL representation of a wsdm.ObjectChunk meta model

An Object Chunk is an extended version of ORM. If the meta-model would only be composed of a collection of statements, it would still not be possible to express the input, output or the variable definitions inside object types (OT) as we exemplified by "*p = ?". We express these annotations via wsdm.ObjectChunkReferences (OCRs). These OCRs (meta model depicted in figure 10) are versatile references allowing us to refer to anywhere inside the Object Chunk.

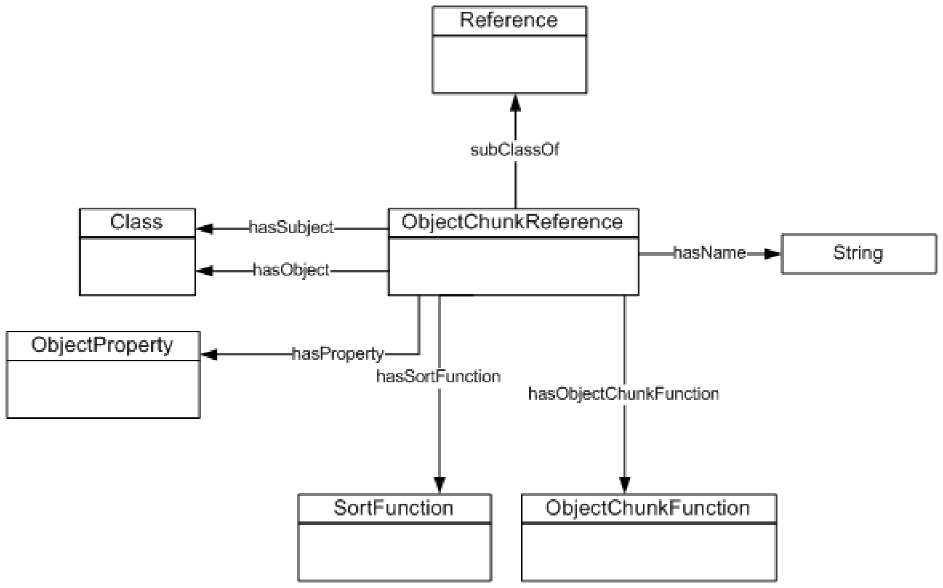


Figure 2.10: OWL representation of a wsdm.ObjectChunkReference meta model

The object chunk reference contains the following properties:

- hasName: the name of the variable inside the chunk, e.g. the variable name for "*p = ?" would be "p".

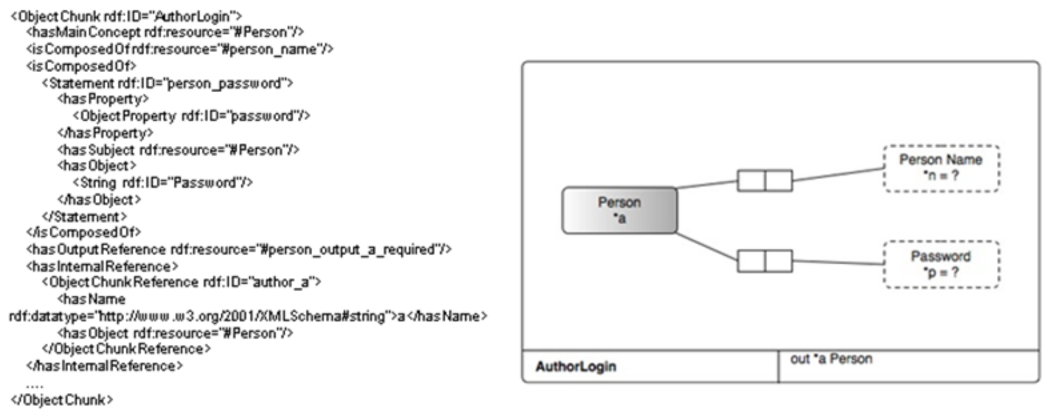
- hasSubject, hasProperty, hasObject: the statement this reference is pointing to.
- hasSortFunction a property specifying a function which decides how the data should be ordered.
- hasObjectChunkFunction is used for specifying user or system interaction.

Note that it is not necessary to instantiate all properties.

Example: Imagine we wish to model our example "*p = ?" placed inside a "PersonName" NOLOT. This means that the user is required to fill in the name of a person on the web site. This is modeled in the object chunk as an internal reference. Modeling this is done by filling in the following properties of the object chunk reference:

1. Fill in the hasName property with "p" to specify the variable name.
2. Link it to the LOT by selecting the PersonName recourse for the hasSubject property.
3. Denoting the reference to require a user input by selecting a "FillOutFunction" instance for the hasObjectChunkFunction property

Example: Figure 2.11 gives an idea how the visual representation of the object chunk is formally notated in OWL using the discussed meta models.



```
<ObjectChunk rdf:ID="AuthorLogin">
   <hasMainConcept rdf:resource="#Person"/>
   <isComposedOf rdf:resource="#person_name"/>
   <isComposedOf>
      <Statement rdf:ID="person_password">
         <hasProperty>
            <ObjectProperty rdf:ID="password"/>
         </hasProperty>
         <hasSubject rdf:resource="#Person"/>
         <hasObject>
            <String rdf:ID="Password"/>
         </hasObject>
      </Statement>
   </isComposedOf>
   <hasOutputReference rdf:resource="#person_output_a_required"/>
   <hasInternalReference>
      <ObjectChunkReference rdf:ID="author_a">
         <hasName
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">a</hasName>
         <hasObject rdf:resource="#Person"/>
      </ObjectChunkReference>
   </hasInternalReference>
   ....
</ObjectChunk>
```

Figure 2.11: Notational vs Graphical Object Chunk

2.2.2.2 Navigational Modeling

Following are the necessary classes required for modeling the navigational design.

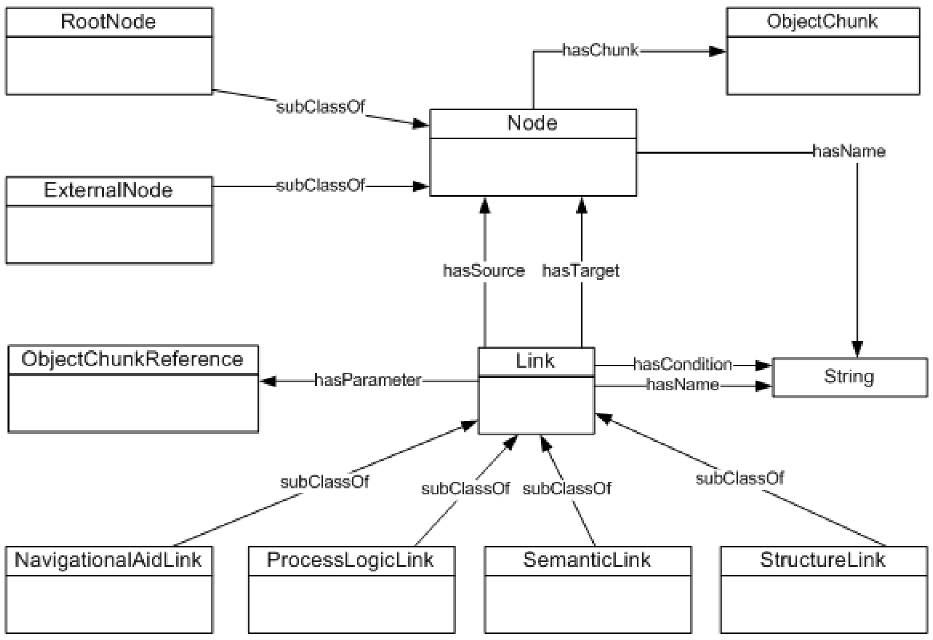


Figure 2.12: OWL representation for the navigation meta model

As previously mentioned, a link can be of any cardinality. To model this, we decompose such links into one-to-one or one-to-many links that can be modeled using the meta-model from figure 2.12.

2.2.2.3 Implementation Design Modeling

The structural model, as depicted in figure 2.13 defines how the nodes from the navigational modeling phase are put onto pages. These pages represent the web pages that the user will perceive.

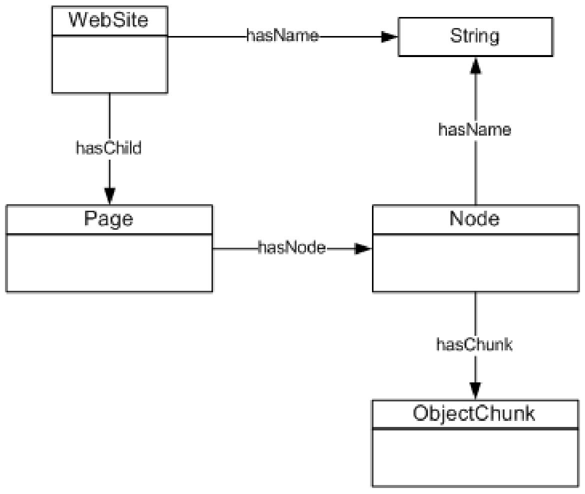


Figure 2.13: OWL representation for the structural design meta model

This represents the hierarchical structure of how WSDM defines a web site: a web site is composed of a collection of pages. These pages contain bits of information represented by the nodes. The nodes itself are typically made up of one Object Chunk.

2.3 Conclusions

In this chapter we have presented the Web Site Design Method for modeling and creating web sites which is based on a pipeline process consisting of 5 phases: mission statement specification, audience modeling, conceptual design, implementation design and the implementation.

We further introduced a formal specification for the models of this methodology to allow the automation of the final implementation phase. This specification, called the WSDM Ontology, is expressed in a Semantic Web technology called the Web Ontology Language and describes the meta-models of the method. This introduction only described the parts in the ontology that were used to implement the WSDMtool discussed in a later chapter, since the presentational aspects in the ontology are not fully defined yet to accommodate the diversity of web design methods.

By extending the ontology notation of object chunks with internal references and an hasMainConcept relation, we are now able to use this notation as input for our automatic code generation tool.

Chapter 3

Web Application Frameworks

The past two decades have seen a shift in software engineering towards software frameworks as the need for code reusability and less time-consuming software development grew. These frameworks facilitated the development of software by empowering the developers and designers to develop their application on a much higher abstraction level, letting the framework handle the more repetitive but time consuming work.

Pree stated [39] that software frameworks consist of two types of spots: frozen and hot spots. The frozen spots are part of the architectural design of a framework and define the basic components and their relationships. These components remain unchanged (i.e. frozen) in most revisions of the framework and provide consistency for the developers. Their counterparts are known as hot spots; these are the blocks of code the developer has to write to use the components of the framework. The business logic of applications will always be described in hot spots.

Software engineering for the web has always been lagged slightly behind traditional application development paradigms. However, recently there has been a growth of development frameworks such as Ruby on Rails, Django, and CakePHP.

This chapter will investigate these frameworks individually followed by a summary explaining their commonalities. Using these commonalities, a conceptual idea can be formalized on how an abstraction can be created for the code generator.

3.1 Django Framework

Django is an implementation of a web application framework written in Python. Its architecture follows loosely the Model-View-Controller pattern, but is called Model-Template-View (MTV-pattern) by the Django development community.

The development started closed source in the hands of the IT department of 'The World Company', a local news-oriented website in Lawrence, Kansas. As of July 2005, it was released to the open source community under a BSD license and was named after the famous jazz artist Django Reinhardt. In June 2008 the framework was adopted by the Django Software Foundation, who became its main supporter and promoter.

3.1.1 Project Structure

A Django project is divided into components called "applications". Each application adds a specific functionality to the project. The idea is that every application should be stand alone or with a minimum of prerequisites.

The project contains the applications and a general 'settings.py' file containing website related settings. This file also 'activates' the many applications a project can have. An important file is the 'urls.py' file. A request received by the server is linked to a specific view-function (the controller in terms of the MVC pattern) to handle the generation of a response to send back. The view-functions are linked to regular expressions which represent the static or dynamic URL's that are delegated to the function.

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^articles/2003/$', 'news.views.special_case_2003'),
    (r'^articles/(\d{4})/$', 'news.views.year_archive'),
)
```

This extract of code[1] contains two URL patterns which are linked to view functions. When a user requests URL '/articles/2003' from the server the framework calls the functions 'news.views.special_case_2003', which will load the data and render the template. The latter tuple contains a more dynamic way of representing URL's through regular expressions; if the URL contains 'articles/' followed by four digits, represented by "\d{4}" in the regular expression, it is sent to 'news.views.year_archive' with the four digits as an argument, e.g.

```
news.views.year_archive(2004)
```

Every application in the Django project contains various files and directories:

- 'models.py' defines all database tables for the application through Object-Relational Mapping, which will be discussed below.
- 'views.py' contains the controller functions. These functions react to events, as specified by the 'urls.py' file in the previous paragraph. They will process requests and pre compute information so it can be displayed on templates.

Optional is 'forms.py', which contains python classes abstracting the web forms that can be displayed on web pages and a templates directory containing html templates to be shown. These templates are the "view"-part of the MVC pattern.

[1] http://docs.djangoproject.com/en/dev/topics/http/urls/#topics-http-urls

3.1.2 Object-Relational Mapping

To facilitate the interaction with the database through a more object-oriented approach instead of the traditional SQL statements, the principle of ORM has been introduced. Tables in the database are represented as models in python classes.

```
from django.db import models

class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

We see here two class definitions[2] of albums and musicians. The framework knows these are models since they are located in a models.py file in the application directory and the specific classes are inherited by "django.db.models.Model". Django translates these models to SQL statements.

The example python code will be translated to the following SQL:

```
CREATE TABLE "application_musician" (
    "id" integer NOT NULL PRIMARY KEY,
    "first_name" varchar(50) NOT NULL,
    "last_name" varchar(50) NOT NULL,
    "instrument" varchar(100) NOT NULL
);
CREATE TABLE "application_album" (
    "id" integer NOT NULL PRIMARY KEY,
    "artist_id" integer NOT NULL REFERENCES "application_musician" ("id"),
    "name" varchar(100) NOT NULL,
    "release_date" date NOT NULL,
    "num_stars" integer NOT NULL
);
```

Django handles these conversions in the background of the framework, but it can be requested through 'python manage.py sql application', which will output all CREATE SQL statements for inspection purposes. Django also adds a primary key if no primary key fields were explicitely specified in the model definition.

[2] http://www.djangoproject.com/documentation/0.96/models/manipulators/

3.1.3 Architecture of Django

When a browser requests a URL it is sent to Django's URL dispatcher. This dispatches the request to the view function linked to the first regular expression that succeeds on the URL string. These view function can either be custom written or generic. They contain business logic for the functionality the URL should provide and will interact with the models inside the application. The view-functions will decide which template should be rendered and what data is passed to them. The templates follow an inheritance principle so it is easy to define the general structure of the website in one template (called a base template) and have page-specific templates inherit this structure. The templates themselves receive the data sent by the view-function and decide how to present this information for the user.

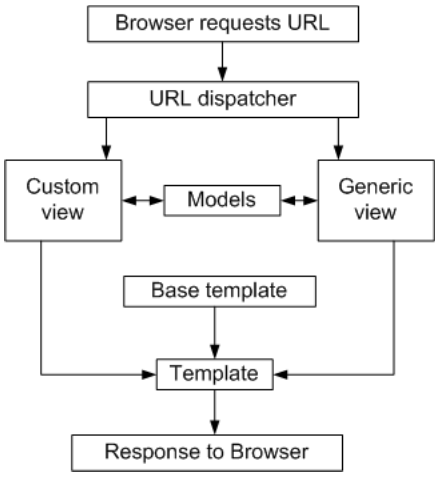


Figure 3.1: Django Framework – Request/Response workflow

3.2 Ruby on Rails Framework

The most common framework for rapid web development and prototyping is currently Ruby on Rails [44]. Even though development on both Ruby on Rails as Django started around the same time, Rails went public a year (July 2004) before Django. Because of this Rails obtained a head start in getting community contributions and reached milestone version 1.0 in December 2005. Currently it has already reached version 2.2.

Rails is largely build around three philosophies that define the architecture and mentality behind the framework:

- Don't Repeat Yourself, or DRY, suggests reusability of code.
- Convention over Configuration: Instead of hard coding information in the system, Rails will configure itself without bothering the developer. Through a series of conventions, such as file naming schemes etc. Rails instantly recognizes models, templates and relations between them.

• MVC architecture.

The major difference between Ruby on Rails and Django is the "Convention over Configuration" mentality. Django requires explicit notation of functionality, whereas Ruby on Rails assumes there is functionality by "guessing" for it based on specific conventions.

3.2.1 Project Structure

In contrast to Django, Ruby on Rails generates a lot of files already. Applications can be created by running a script which generates code and does scaffolding. This actually means that it will automatically generate basic creation, read, update and delete (CRUD) functionalities for the model controller and templates.

Basic URL routing is also automatically generated; this can be altered in 'config/routes.rb' which looks[3] like:

```
ActionController::Routing::Routes.draw do |map|
  map.connect 'products/:id', :controller => 'catalog',
                              :action => 'view'

  # Install the default route as the lowest priority.
  map.connect ':controller/:action/:id.:format'
  map.connect ':controller/:action/:id'
end
```

The keyword is again pattern matching; like Django, it will try and match the URL request to a pattern, defined in this do-statement. The code above for example contains a link "products/:id", which will get triggered when for example "products/521" is called. It will then activate the ActionController class "catalog" by executing the method "view". It also contains some defaults so it will automatically retrieve the controller, action method and parameters from the URL.

The controllers are defined using ruby classes:

```
class BookController < ApplicationController
   def list
      @books = Book.find(:all)
   end

   def show
      @book = Book.find(params[:id])
   end
```

[3] Code examples from http://guides.rubyonrails.org/

```
   def new
      @book = Book.new
   end

   # more actions such as create, edit, delete, update
end
```

This example defines a controller for Book objects. Most of these actions are self-explanatory. What can be noticed here is the controller does not specify any templates to be used to output; the naming conventions will specify what templates are used.

3.2.2 Object-Relational Mapping

Ruby on Rails uses similar Object-Relational Mapping techniques as Django, in which models are defined as classes which are inherited from 'ActiveRecord::Base', which adds basic CRUD operations to the models.

Models are represented as basic ruby classes:

```
class Post < ActiveRecord::Base
        validates_presence_of :title
        has_many :comments
end

class Comment < ActiveRecord::Base
        belongs_to :post
end
```

Ruby itself contains only validation information and relationships of the models. "Post" objects have a mandatory "title" attribute and a one-to-many relationship with "Comment". We see the "Convention over Configuration" aspect of Rails in effect. We do not specify that "has_many :comments" is linked to the Comment object, rather than that we use a convention that figures out "comments" is plural for our Comment.

3.2.3 Architecture of Ruby on Rails

Requests sent to the Ruby on Rails framework[4] (Figure 3.2) server are handled by the Dispatcher. this wil map the URLs from the request to an 'action controller'. The action controller processes incoming requests to the server, extracts its parameters and dispatches them to a specific action. The 'Action View' receives these requests and manages the business functionality of the view. The View function interacts with the models through an

[4] http://www.eclips3media.com/workshop/2007/06/introducing-ruby-on-rails/

'Active Record' pattern. All database transactions go through these classes inherited from Active Record class.

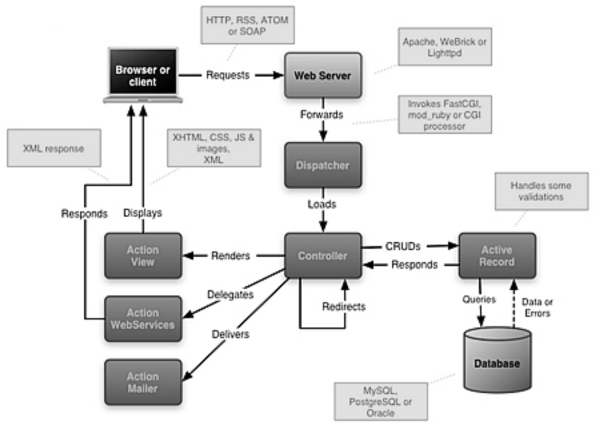


Figure 3.2: Ruby on Rails Framework – Request/Response workflow

3.3 CakePHP Framework

PHP, the recursive acronym for PHP: Hypertext Preprocessor is one of the major contenders in the web development scene. Originally inspired by Perl, PHP saw the light in early 1994 under development by Rasmus Lerdorf, a software engineer at IBM. Since then it has gained increasing momentum as perhaps the primary server-side scripting language available.

However with the presence of python, ruby and their respective rapid web development frameworks, the usage of PHP started decreasing (http://www.php.net/usage.php). Seeing the interest for web development frameworks grow, the PHP community decided it was time to develop their own framework; one of these is CakePHP (http://www.cakephp.org).

The community found the inspiration for the framework's architecture and functionality at Ruby on Rails. It is therefore easy to understand why there are so many similarities in their development philosophies. One of their main correlations is the same usage of the Model-View-Controller pattern, especially their definition of models.

3.3.1 Project Structure

The common CakePHP projects contain the following file structure:

- An 'app' directory where all the models, views and templates will be defined (the hot spots).
- A 'cake' directory containing all framework specific files (the frozen spots).
- A 'docs' directory containing all documentation, readme's and license text files for the project.

- A default 'index.php' file
- A 'vendors' directory where third-party PHP libraries can be placed.

Inside the 'app' directory other directories can be found for configuration, models, controllers, views, internationalization, etc.

An added difference is the fact that every project created in CakePHP contains the full web application framework. This is due to the way one has to program in PHP; instead of importing libraries which are on the execution path like in Python or Ruby, PHP only allows the inclusion of files using a relative or static file path.

Similar to Ruby on Rails, CakePHP[5] calls the way URL's are linked to controller functions "Routing".

```
URL to controller action mapping using default routes:

URL: /monkeys/jump
Mapping: MonkeysController->jump();

URL: /contents/view/chapter:models/section:associations
Mapping: ContentsController->view();
$this->passedArgs['chapter'] = 'models';
$this->passedArgs['section'] = 'associations';
```

This code excerpt shows two routing entries: a static and a dynamic one. The second (which is dynamic) catches URL requests such as "/contents/view/chapter:5/section:2/". What can be noticed is that the look of CakePHP's URL's differs from either Ruby on Rails and Django because of the "variable assignment" feeling to them.

3.3.2 Object-Relational Mapping

Similar to the previous two frameworks, models are again represented as PHP classes, extended by class 'AppModel' specified by the framework.

```
class Recipe extends AppModel {
    var $name = 'Recipe';
}
```

We defined a Recipe model named "Recipe". The $name variable is a PHP4 compatibility feature already introduced by the language. Similar to Ruby on Rails, no model attributes in the sense of database columns are defined.

Manipulation of these models happens inside their respective controller-functions, which is extended by the framework's 'AppController' class.

[5] Code examples from http://guides.rubyonrails.org/

```
class RecipeController extends AppController {
    var $uses = array('Recipe');
    function index() {
       $this->Recipe->find('all');
    }
}
```

Due to language limitations, the controller class needs to contain a $uses variable specifying which models it will be manipulating. Apart from this, the controller functions are implemented in a similar fashion as in Ruby on Rails.

3.3.3 Architecture of CakePHP

Page requests are handled[6] (Figure 3.3) the following way:

1. The client sends a request for a specific URL .
2. The router functionality of the framework receives the URL request and parses this by extracting the parameters, controller, action and arguments important for the controller.
3. After this, the router maps the URL to a controller action, being a method of the specific controller class. Any beforeFilter() functions defined in the controller are executed before the controller action itself is executed.
4. Inside the controller action the business logic for the application is executed. The controller manipulates the database through the use of models.
5. After the model has retrieved the data, it is returned to the controller, where actions such as session manipulation, authentication is applied.
6. The controller hands the information to the view function which handles presentation logic.
7. afterFilter() functions are executed if present and the information is sent back to the client.

[6] http://book.cakephp.org/nl/view/21/A-Typical-CakePHP-Request

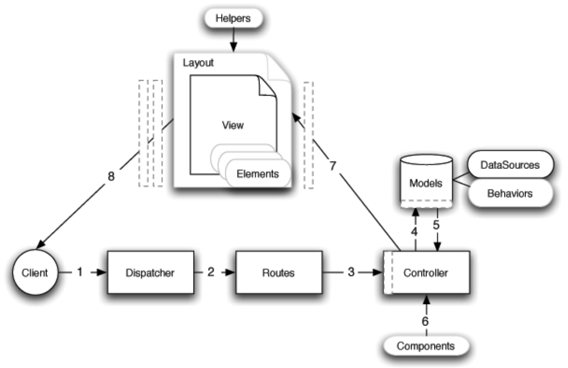


Figure 3.3: CakePHP Framework – Request/Response workflow

3.4 Framework Abstraction

In the previous sections we've investigated three major web development frameworks. We will now investigate if those frameworks have some common principles. This can then be used to create an abstraction layer for web development frameworks, which on its turn can be used for our code generation to make part of the code framework-independent.

3.4.1 Project and Application Structure

In the three frameworks discussed, the specific websites that are implemented are called 'projects'. Unlike CakePHP, Django and Ruby on Rails divide the project itself into multiple 'applications', each complementing the project with a specific functionality. For example a project can contain a "calendar" application and another "blog" application. These applications are not necessarily independent to each other. In the example it is possible that the blog application generates posts which are also visible on the calendar.

3.4.2 Model-View-Controller Architecture

We notice that the previous frameworks are all implemented with a Model-View-Controller architecture, which will also be our main focal point of the code generator.

3.4.2.1 Models

Models are always represented by classes, which are inherited by framework specific classes. These parent classes contain the framework implementation for data manipulation, i.e. Object-Relational mappings. A major difference that could be found is what the models contain; in Django these models contain attributes which represent relationships and the various columns to be found in the database table, whereas in Ruby on Rails and CakePHP only contain relationships and validation annotations.

3.4.2.2 URL Routing

URL routing is fairly similar among the three investigated frameworks; input is always the URL which the client sends to the server, which is parsed and compared to the routing instances defined by the user.

3.4.2.3 Controllers

Django differs greatly from Ruby on Rails and CakePHP by implicitly linking the controller to the view, whereas Ruby on Rails and CakePHP link them to the model. They also follow specific naming conventions to figure out what controller and view to use. However this does not concern us at this point because it is an implementation issue for the specific framework and not for the code generator abstraction. Summarizing, every page in Ruby on Rails and CakePHP is controlled by one or more methods in a controller whereas in Django every page has a separate view-function. (Remember that Django uses the word "view" for controllers and "template" for views)

3.4.2.4 Views and Templates

Views are the presentation part of the data obtained by the controller. It is therefore necessary to have communication from the controller functions to the view functions, because the view functions need to have an idea of what variables to which they have access.

For example a controller function specifies a publications-variable, containing a list of Publication objects. The View needs to know that it can output Publications through iterating over the publications-variable.

3.5 Conclusions

In this chapter we have presented the three most prominent web development frameworks: Django, Ruby on Rails and CakePHP. For each of these frameworks we have discussed its project structure, i.e. how they divide a web site into smaller coherent portions, the architecture and how their communication layer with the persistence layer, e.g. a database, works.

Following this in-depth presentation, we have compared the frameworks to each other in order to extract their similarities. These similarities have been combined in the Framework Abstraction section of this chapter and provide us with an abstract framework which provides platform independence, yet describes all discussed platform specific implementations.

Chapter 4

WSDMtool Architecture

4.1 Introduction

In the previous chapter we described the various frameworks that can be used to generate towards. Furthermore we have extracted the similarities and presented (section 3.4) an abstract framework containing them. As we are targeting a tool that can, in principle, generate code for any of these frameworks, we decided to keep most of the tool as generic as possible, so that only a small part needs to be re-implemented when other frameworks are considered. This is where the abstract framework comes in place. The tool for generating web applications using any of the above frameworks has to adhere to this abstract framework. So, most of the generation tool is common for all frameworks, while the more specific portions are handled by the framework-specific code generator module.

The architecture, as depicted in figure 4.1, is made up of three layers. The file containing the site description formalized by the WSDM ontology is read by a third-party library called Jena, which translates the raw XML-based input into a Java API which enables us to query the information inside the OWL file. A secondary layer introduces another indirection which translates these OWL-based classes into WSDM-based classes. Finally the third layer contains the generator routines. It is possible to define a framework-specific code generation module through the use of the Interface pattern. The platform independent generator will communicate through this interface to the platform-specific code generator.

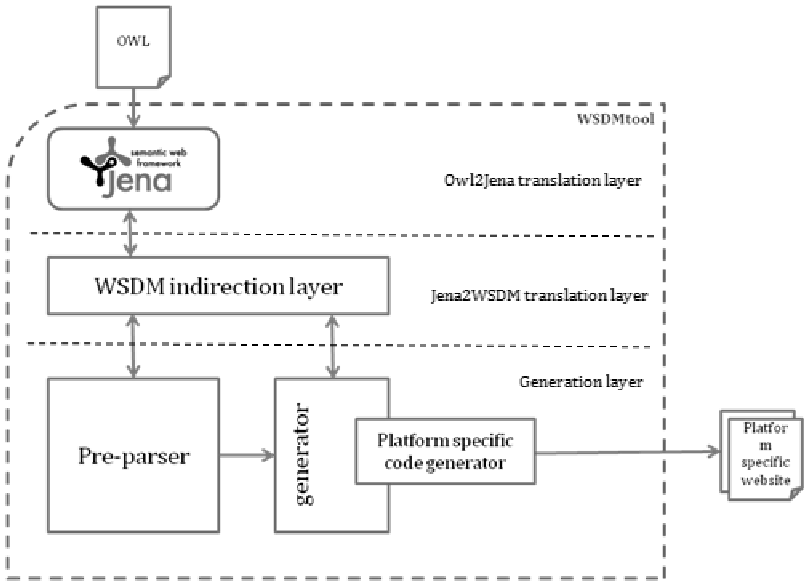

OWL
WSDMtool
semantic web framework
Jena
Owl2Jena translation layer
WSDM indirection layer
Jena2WSDM translation layer
Generation layer
Pre-parser
generator
Platform specific code generator
Platfor m specific website

Figure 4.1: WSDMtool architecture

Generation of the web site happens in the third layer, which has been split up into a number of separate tasks:

- The Database generator creates the models which are used for communication with the database and provide the object-role mapping.
- The Controller generator handles the generation process of the controller functions that contain the business logic for each page.
- A Template generator completes the MVC-pattern by generating templates which will be used to render the front-end of the web site.
- Accompanying this MVC pattern is the Routing generator in charge of defining the dispatcher that maps the URL address requests onto the controller functions, i.e. this decides which page you will view when requesting a specific URL.
- List output generator contains the routines for generating information chunks, which is described in the Template Generator section.

These classes make up the abstraction layer of the WSDMtool and provide a common base for each platform-specific code generator created.

First we will elaborate the aspects of the platform-independent generation process, such as the pre-parser, ontology layer and the abstraction choices made for the template generator, followed by an in-depth overview on how our platform-specific Django generator was created.

4.2 Pre-Parsing of Information

The information contained in the OWL file is preprocessed before the various tasks are executed. This processing step will be the same for every web framework considered and it is therefore important to be fully platform independent so it can always be used.

With the research into various web frameworks such as Django and Ruby on Rails, it became apparent that only the model part of all the MVC-based frameworks could be generalized. Object Chunks in the framework however contain only the necessary model information to perform a specific task for a web page. In order to create the models of the framework we require the business information model, which is spread across all object chunk s. To solve this problem two solutions can be suggested:

1. If we adopt the idea of creating an object chunk containing all the statements of the web site defined in the ontology, we could use this object chunk to generate the model and database tables.

2. Because the code generator needs to generate all the pages before the site can actually be used, we could create the models incrementally; the models can be expanded every time a new object chunk with new statements has to be generated.

There is however still a small problem that has to be resolved: the ontology does not speak about models. The code generator must distinguish which statements are part, i.e. attributes, of a specific model and which classes are the actual models. However, figuring out which are the actual models is not complicated; they can be easily extracted because one of their super classes is the wsdm.I-F-ModellingConcept.Class class. Once identified the wsdm.Statements linking two of these classes can be considered as relationships between their respective models. All other wsdm.Statements would then describe the model attributes.

Our Pre-Parser converts the business information model (BIM), obtained by combining all concepts and statements about these concepts, into internal Model instances. Each of these instances represents a table in the database that is required for the website. Such a model contains ModelField instances, each representing a column in the table and for each ModelField we have a ModelFieldType, which represents the type of data contained in the column, indicating whether it is e.g. a string type or if it contains a relationship to another model. In that case the model to which it is related will also be present in the ModelFieldType.

A current limitation of our implementation is that the relationships between the various models are interpreted as many-to-many relationships . This was chosen because every relationship can be expressed as a many-to-many relationship. By also interpreting the cardinality of the relationship in the WSDM OWL specification, it would become possible to choose the correct type of relationship between two models. If various object chunks would define the cardinality of a relationship between two models differently, this would be resolved by choosing the least restrictive relationship and using this as a ModelFieldType.

To allow generation towards various web frameworks, an implementation of an abstract factory pattern has been chosen to facilitate the separation. The interface classes themselves contain a "generate(PreParser op, String frameworkLocation)" method. This method only requires the preprocessor class to retrieve the preprocessed information and a location where it is allowed to generate its platform specific code.

4.3 WSDM Ontology Layer

An initial idea for WSDMtool was to give more functionality to the pre-parser. The idea was to have the pre-parser directly read in the OWL classes through Jena and create so-called "internal classes" for the WSDMtool. These internal classes were considered to be the equivalent for platform-independent models (PIM) commonly used in the model-driven architecture approach. However this idea was discarded because the WSDM specification itself could already be considered as PIMs and another way of modeling them was deemed unnecessary. Furthermore the differences between the various web frameworks only had the object-relational mapping in common and it became apparent that no general internal representation could be agreed upon for the template and controller objects.

A new idea was found to create a separate indirection layer allowing us to communicate directly with the WSDM model classes instead of the OWL classes. The code generator never communicates directly to the Jena API but uses the WSDM API to get its information. This made it possible to work for example with Object Chunk, Node and Concept classes and drastically decreases the complexity of the implementation.

To summarize: we have decoupled the generation layer from the OWL2Jena layer by adding an Jena2WSDM indirection layer. The generation layer only communicates to this WSDM layer.

If at some point in time it is decided to represent the WSDM models with something other than OWL, the only modifications that need to happen are within the WSDM indirection layer so it would not use the RDF-specific Jena library.

4.4 Unambiguous Interpretation of Object Chunks

Visually interpreting object chunks appears deceptively straightforward, because the human brain allows us to figure out the meaning of the object chunk by its name and establish an interpretation by inspecting the various concepts connected to each other. It is however not so straightforward for a piece of software to perform this same task; a formal methodology must be decided upon to achieve similar results as if done manually.

To provide such a methodology, we have divided the object chunks into two categories: information and interaction chunks. The information chunks category contains object chunks that only present the user with information; examples of this are the "SubmissionsInfo" object chunk from the case study described in chapter 5. The interaction chunks category contains object chunks requiring an action to be taken by the user or system, i.e. filling out a form.

4.4.1 Information Chunks

A first problem arose early during development: ”Where do we start with the interpretation process of an object chunk?”. If we have an object chunk called ”ListPapersAndTheirAuthors”, it would be only logical to start iteration at the Paper-concept and show the name of the paper and authors for each iteration step. This problem had already been observed in previous work [23] and had been resolved by introducing the idea of a ”Main Concept” inside every object chunk. This addition to the WSDM methodology had not yet found its way into the OWL description for web sites but has now been added. With annotating a concept as being the main concept of an object chunk the algorithm for generating templates now has a position to start from.

The following depth-first graph traversal algorithm templateGenerator handles the generation process of information chunks.

```
DEFINE templateGenerator(objectChunk, currentConcept)
  FOREACH concept IN neighbours(currentConcept, objectChunk)
    IF concept is LOT
      RETURN output concept instance
    ELSE concept is NOLOT
      RETURN templateGenerator(objectChunk, concept)

templateGenerator(objectChunk, get-main-concept(objectchunk))
```

The algorithm considers our object chunks to be mere graphs. The nodes are represented by object types (OT) such as lexical and non-lexical object types. More particularly the lexical object types (LOT), as defined by ORM [29], are concepts such as ”name”, ”telephone number”, ”title” which are atomic and can be represented by wsdm.MultimediaConcept's such as wsdm.String or wsdm.Integer and are always the leafs of the graph. The intermediary nodes in our graph always consist of non-lexical object types (NOLOT) representing abstract Concepts such as a person (which can be referred by LOTs). Finally the roles between the various OTs make up for the edges between the nodes.

ORM states that every role between two Object Types is bidirectional. This assertion would mean that we would constantly keep revisiting concepts. The direction of each role is therefore made unidirectional, making sure we don't revisit nodes through the same or inversed role.

4.4.2 Interaction Chunks

Generation of interaction chunks is different than generation of information chunks. This is because the user has to be able to fill in information in the system. For generating web sites this boils down to creating web forms.

Generating a web form is only the first phase in interpreting interaction chunks. These types of object chunks also explain what has to be done with the information received by the user. This means that apart from generating an output, it is also required to generate code routines which handle the filled in form request sent by the user by submitting the form.

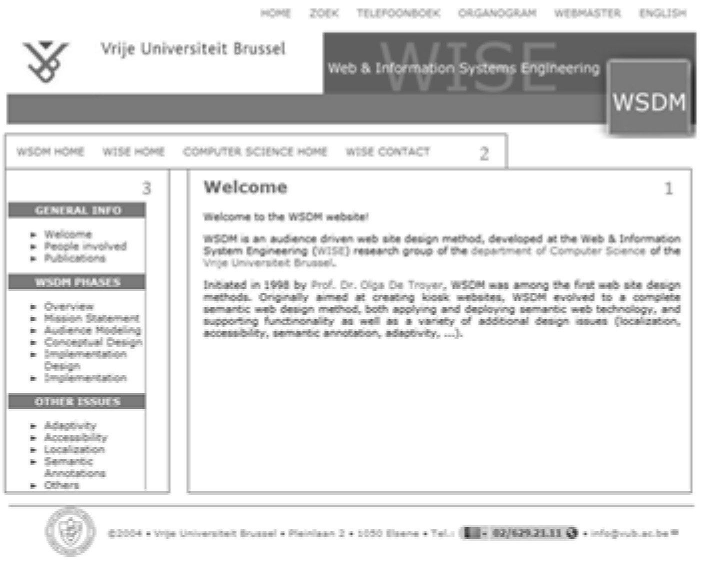
4.4.3 Generating Structure without Presentation

Whenever we are speaking about views (Ruby on Rails and CakePHP) or templates (Django) we are referring to the HTML output not the HTML and Cascading Stylesheets (CSS) output. The code generator application will not contain routines to generate CSS output. This is done for multiple reasons.

The primary goal of WSDM is to conceptually model the content of a website. Initially, modeling the look and feel of the website was not considered in WSDM. After a couple of iterations the WSDM ontology was extended to also allow storing the (conceptual) modeling of the layout of the website. However this layout design cannot yet be considered as a full-fledged to model the look and feel of a web site. This is one of the main reasons why page styling will not be implemented. The World Wide Web is one of the fastest evolving domains in existence. Trends for web page styling and typesetting rapidly evolve and customers become demanding in this respect. This need cannot be reflected efficiently in the ontology as currently it only allows general layout concepts such as three-columns, navigation menu's, bullet lists. but does not yet allow the increasingly more popular "Web 2.0" features such as websites with AJAX enabled web calls. While the requirements for the content do not evolve, those for layout and styling do and this is an important argument not to focus on page style generation for our code generator.

A second reason is based upon another fact of web development in the business world. As Jacob Kaplan-Moss stated [30] web development is a two-fold process; the programming team creates the database models, the controller functions to handle requests, basic templates and these are then sent to another team who re-order the layout, add CSS and finally publish it. His argument for this strategy is that good programmers are not necessarily good graphics designers and suggests that two separate professions should do these two processes. The inverse is also an argument; graphic designers should not have to be fully aware of all the intricacies of the framework or the ontology to present the information.

The code generator will therefore act as the programmer's team; it will generate the templates without styling. Several features have however been implemented to easily support the use of style sheets. Every generated template is inherited from a so-called base-template which has to be provided by the user. This template defines the general layout of the website, for example whether the site uses a two or three column layout. It also defines something called "blocks" to specify where the generated information is to be displayed inside this template. This is important, as we want to be able to specify where the navigational links and information has to be placed.



Welcome

Welcome to the WSDM website!

WSDM is an audience driven web site design method, developed at the Web & Information System Engineering (WISE) research group of the department of Computer Science of the Vrije Universiteit Brussel.

Initiated in 1998 by Prof. Dr. Olga De Troyer, WSDM was among the first web site design methods. Originally aimed at creating kiosk websites, WSDM evolved to a complete semantic web design method, both applying and deploying semantic web technology, and supporting functionality as well as a variety of additional design issues (localization, accessibility, semantic annotation, adaptivity, ...).

©2004 • Vrije Universiteit Brussel • Pleinlaan 2 • 1050 Elsene • Tel.: 02/629.21.11 • info@vub.ac.be

Figure 4.2: Example of a possible layout

In figure 4.2 we would define a base template to contain a header, footer, horizontal navigation (2), vertical navigation column (3) on the left and a content column (1) on the right side. The content appearing in the red boxes will be different for every page we visit. It is this and only this information that the generator will create.

4.5 Platform-Specific Django Generator

As a proof of concept for a generator we have implemented a platform specific code generator towards the Django web development framework, described in the previous chapter. As previously discussed, a Django project is a combination of applications. Because the WSDM ontology uses a navigational-oriented methodology, it is not immediately possible to extract the different applications from the site description. Therefore we will currently restrict our code generator to the functionality of generating the web site as a single application, thus making it possible for generation to frameworks which support either single or multiple applications.

Django was chosen because of its many advantages over other frameworks; the framework is based on the Python programming language. As such, it is a widely available, high-level, object-oriented programming language which runs on all operating systems. Furthermore the choice of whether the database is MySQL, Oracle or any other database management system does not matter since Django provides support for each of these databases, drastically simplifying the task.

4.5.1 Generating Database Models

There are two options on how to store data for a web site: storing the content in the WSDM ontology itself or only storing the content structure in the ontology and storing the actual content elsewhere (e.g., in a database).

The WSDM ontology already supports storage of data via WSDM statements. These are based on the RDF Triple principle, which links a Subject to an Object via a predicate "(Subject, Predicate, Object)". For example:

```
(Person, hasName, Name)
```

Because the "hasName" property is unique for the entire ontology description of the website we can "instantiate" it through

```
(ID0001, hasName, "Bob T.")
```

This links an instantiation ID0001 of a "Person" to the instantiation of "Name" and describes that Person ID0001 is named Bob T. Let's say we also link the "Person" to a specific "Age", we can also give Bob an age by using

```
(ID0001, hasAge, 34)
```

A database that is specifically built for storing information as these statements is called a Triplestore.

The code generator will not use a Triplestore. A Triplestore is a good way of working for rapid prototyping tool but may be inappropriate for a production environment where large volumes of data need to be maintained.

Another possibility is to describe only the content in the WSDM ontology, such as

```
(Person, hasName, Name) and (Person, hasAge, Age)
```

With these OWL statements, we can create the models for our framework and instantiate them in the working framework itself. However, this requires a methodology to convert the conceptual models describing the structure of the content into some implementation format, like a relational database.

It is possible to have a database generator which is solely dependent on the pre-processed Models. The database generator will translate this internal representation into the platform specific representation of a model in the MVC-pattern.

All that was required of the generator was to create a models.py file inside a Django application where the models can reside. The conversion from the internal model structure is as follows:

1. Each instance of the Model becomes a Python class, which inherits from the 'django.db.models.Model' class or another model we have defined to allow model inheritance. For example: a Student model can inherit from the Person model. The end of the inheritance hierarchy is ended by a model inheriting from 'django.db.models.Model'.

2. Attributes are then added to the Django model by iterating over the ModelFields of a Model. The type of the attribute is decided through a mapping of the different WSDM.MultimediaConcepts over the attribute fields made available in Django.

3. Finally the model is registered in the administration to create scaffolding in the administration site for CRUD operations.

As an exception to WSDM.SystemFunction the NEXT-function is not implemented in the controllers' generator but rather in the database generator. If there exists an object chunk that has a NEXT-function as an internal reference to a specific concept, the model field representing this concept will be interpreted as being an automatically incrementing field in the database.

4.5.2 Generating Controller Functions

For Django, every object chunk can be generated in the corresponding page's controller-function. The problem with Ruby on Rails and CakePHP however lies with the requirement to generate a method and associate it to the correct controller, e.g., associate a "create" action for "Paper" to the "PaperController" and not to the "AuthorController". The specific framework code generator should therefore contain a mechanism to establish which model it has to choose. Studying various object chunks we conclude that this could be based on the wsdm.hasMainConcept property introduced in the previous section.

On close inspection we notice that not a lot of the information contained in the object chunk is represented in the controller function. This is because the object chunk contains statements which represent the data to be displayed which belong in the template/view model and not in the controller model. Regarding the controller, only the ObjectFunction aspects of the WSDM ontology and the object chunks in/out values are of importance. This describes what the controller needs to know: which data is coming in, which data has to go out and what manipulations it has to perform on them.

A comparison between the WSDM notation and the preferred framework specific implementations can be found in appendix B.

Our Django implementation generates a controller function resembling the code below.

```
DEFINE pagename(inputref1, inputref2, ...)

  - fetch required database objects referenced by the inputrefs

  IF post request sent:
    - populate form instance with user input
    IF user input valid:
      - retrieve user interaction references from form instance
      - execute system interaction references
      - save changes to database
      - fetch objects referenced by all other references

      IF action is outboundlink1
         RETURN redirect routine to next page on navigation
                path parametrized by all output references
      IF action is outboundlink2
         RETURN redirect routine to next page on navigation
                path parametrized by all output references
         ...

    ELSE
      - Create form instance

  RETURN redirect routine to next page containing all necessary
         references of the next page
```

The following can be said about the code above:

- The code highlighted in bold is only generated for interaction chunks.
- Input references are handled as function arguments.
- Each outbound link creates a new if-test to handle the specific link.

The internal references have been split up into several types of internal references:

- User interaction reference (e.g. "*query = ?") denotes that a model instance called 'query' has to request its value from the visitor. This generally means that the user is required to fill in a form field for this value.
- A system interaction reference (e.g. "*date = TODAY") is a reference similar to the user interaction, but instead of asking the user for input for the variable it represents, the system automatically generates it. Implemented examples are:
  - Create Instance: creates a new instance of a model, e.g. creating a new Person object.
  - Create Property: adds a new property to a model instance, e.g. adding a first name to the Person object.
  - Delete Instance: removes a model instance from the database.
  - Today: generates the current time and date using python's datetime.datatime.now()

- A data-lookup reference (e.g. "*p") does not perform any user interaction or system interaction in the same sense as the system references, but rather does a database lookup to retrieve its instantiation value. It does so by querying for all model instances of the model and filters the query by every instantiated reference directly and indirectly connected to it.

4.5.3 Generating Templates

With both the models and controller generation processes already addressed, we can now focus on the third and final requirement for our Django code generator for the WSDMtool: the generation of templates. The platform-independent part of the code generator already contains the logic for generating templates. Specifying this for the Django generator is a matter of inheriting the abstract class and defining the abstract functions. These functions represent the abstract routines as defined in the templateGenerator algorithm which can be specialized for the Django generator. For example the code below specifies that the encapsulation of objects will be done using HTML div-tags.

```
protected String startListContainer(String listName, int tabDepth) {
    return tabs(tabDepth)+"<div class=\""+listName+"\">\n";
}
```

Defining these procedures will handle the generation of information chunks. The generation of interaction chunks is handled by defining a new form class for each interaction chunk. This class contains the information of what form fields are visible and which input field type, i.e. an option-list, radio checklist or text input fields, etc. Generation of the actual HTML form is handled by Django itself.

4.6 Conclusions

The WSDMtool interprets the OWL specification as described in chapter 2. This XML syntax is translated to Jena classes available for the Java programming languages. An indirection layer translates these Jena classes and specifies an API which can be used by the code generator.

The generation process was split into multiple phases targeting different aspects of the MVC-pattern. Each phase has been deliberately kept as generic as possible in order to serve as a base for generating towards a multitude of frameworks.

Some of the tool's most distinguishing features are:

- Support of the WSDM methodology

- Possibility for generating towards multiple unmodified web frameworks through the use of framework-specific code generation modules.

As a proof of concept a platform specific generator towards the Django web framework was implemented.

Chapter 5

Case Study

5.1 Introduction

The Conference Review System has been a case study used by many web modeling methods, such as OO-H [14], OOHDM [42] and WSDM [17]. The case study [41], created by Daniel Schwabe, was introduced in 2001 for the First International Workshop on Web-Oriented Software Technology [1] (IWWOST) in Valencia, Spain.

The case study of the conference review system describes a web application that supports the business logic of submitting, evaluating and selecting papers for a conference.

The web application supports the following actors:

- Author actors submit the papers for acceptance at the conference.
- Program Committee (PC) chairman is responsible for creating a conference and assigning members as PC members.
- A PC member is responsible for evaluating a collection of papers and selecting people as reviewers.
- A Reviewer reviews the submitted papers.

After reviewing the case study, we chose to only focus on implementing the author navigation track as done in [41] due to all tracks having the same complexity in regards to code generation.

The complete collection of object chunks used to create the author track part of the web application can be found in appendix A.

5.2 Model Generation

The models are generated by interpreting the business information model (BIM).

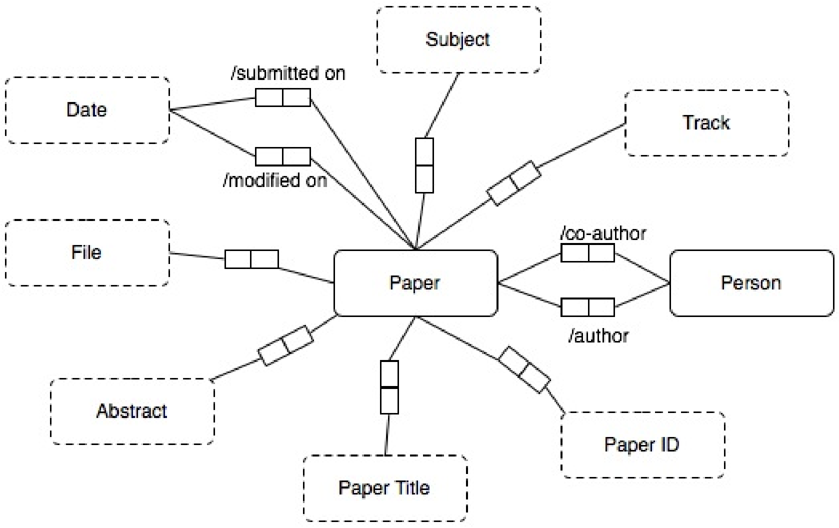


Figure 5.1: Business Information Model of the Paper concept

Figure 5.1 depicts how the Paper concept is connected to other object types in the business information model. By applying the interpretation techniques from the previous chapter, each NOLOT is translated into a class definition. Roles linked to LOTs are interpreted as table columns whilst roles linked to other NOLOTs are interpreted as many-to-many relationships.

```
class Paper(models.Model):
    abstracted_to = models.CharField(max_length=255, null=True)
    in_track = models.CharField(max_length=255, null=True)
    last_modified_on = models.DateTimeField(null=True)
    uploaded_to = models.FileField(upload_to='documents', null=True)
    submitted_on = models.DateTimeField(null=True)
    about = models.CharField(max_length=255, null=True)
    paper_identified_by = models.IntegerField(null=True)
    authored_by = models.ManyToManyField('Person', related_name="author", null=True)
    coauthored_by = models.ManyToManyField('Person', related_name="coauthor", null=Tru
    titled = models.CharField(max_length=255, null=True)

admin.site.register(Paper)
```

The different concepts defined in the WSDM specification as wsdm.MultimediaConcept instances are mapped onto the model fields (CharField, DateTimeField, etc.) provided by the Django framework.

5.3 Controller Generation and Object Chunk Reference Handling

We will now describe the generation of the Django controller functions, which handle the business logic described by the object chunks.

An abstract idea of this code has already been given in chapter 4 in the section on generating controller functions for the Django framework. We will now give some practical examples to explain this code more in-depth.

An interesting example to start with is the following object chunk:

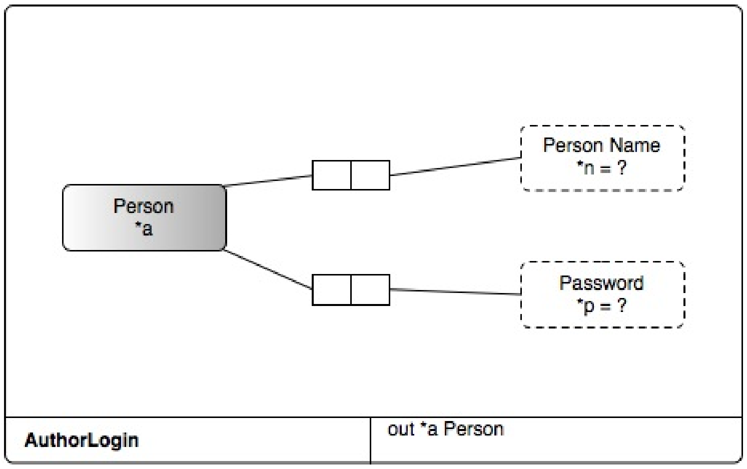


Figure 5.2: AuthorLogin object chunk

The object chunk in figure 5.2 represents the functionality required to log in as a specific Person by providing a correct username and password. First the object chunk requires the user to enter a username and password, denoted by the "*p = ?" and "*n = ?" references in respectively the Password and Name LOTs. The main concept is denoted with a gradient background.

If a Person instance a is found with a specific Password p and Name n the output reference required to exit the object chunk and the outbound navigational link can be followed to the next object chunk.

This object chunk shows how the WSDMtool handles user interaction references, such as "*p = ?" as well as references that forces a database lookup using other instantiated references.

Interpreting this object chunk results in the following controller-function code for the Django framework:

```
def home(request):
    if request.method == 'POST':
        authorloginform = AuthorLoginForm(request.POST, request.FILES)
        if authorloginform.is_valid():
            p = authorloginform.cleaned_data['password']
            n = authorloginform.cleaned_data['named']
            try:
                a = Person.objects.get(Q(password = p, named = n))
            except ObjectDoesNotExist:
                return HttpResponseNotFound("Your request was incorrect.")
            if request.POST['action'] == "author's submissions":
                return HttpResponseRedirect(
                            reverse('wsdmsite.views.submissions',
                                    kwargs = {'a_id': a.pk}))
    else:
        authorloginform = AuthorLoginForm({})

    return render_to_response('home.html',
                {
                    'authorloginform': authorloginform,
                })
```

As well as a class definition, which contains the form description:

```
class AuthorLoginForm(forms.Form):

    def __init__(self, *args, **kwargs):
        super(AuthorLoginForm, self).__init__(*args, **kwargs)

        self.fields['password'] = forms.CharField(required=False)
        self.fields['named'] = forms.CharField(required=False)
```

The following can be observed from both code excerpts:

- The controller function has the name of the Page: "home".
- The branch created by the first if-test contains all the information to be handled after the user has provided the information. All code below the if-test concerns the creation of a web form, which the user will be able to fill in.
- A form variable in the form-processing branch is instantiated with the POST request as an argument, binding the user input to the form fields.
- All Object Types containing references requiring user interaction, i.e. "*p = ?", are turned into variables which extract information out of the form instance.

- Other intermediary references such as "*p" that do not require any user or system interaction are then handled in a separate try-except clause.
- After successfully handling a form request, the visitor is taken to the next page according to the navigational modeling, in this case the "submissions"-page.
- The web form itself is an instantiation of a separately generated AuthorLoginForm class containing all the fields ("named" and "password") as required by the object chunk.
- The HTML itself is created using Django's built-in render_to_response function, which takes a dictionary containing variables required by the template.
- Errors which occur when no Person could be found containing a particular username and password is handled by generating a 404 error page. The handling of the error could also be done differently: by changing the return HttpResponseNotFound("Your request was incorrect.") to a simple pass statement, a user would return to the login form to try again after submitting incorrect credentials.

The login object chunks makes it clear how internal references such as user interaction references (*p=?) and data lookup references (*a) are handled. This demonstrates most of the internal references except the system interaction reference which will be discussed now using a new example.

The object chunk in figure 2.4 represents the registration of a new co-author for a paper in the conference review system.

When the navigation enters a node containing the AddCo-Author object chunk, a paper is already known, as described by the input reference "in *p Paper". The user is able to enter the name for a person ("*pn = ?" in the Name LOT) and the system will create a new person ("*a = NEW" in Person NOLOT) and add the name and a newly generated id ("*id = NEXT" in PersonID) properties to the model instance. The object chunk has now finished and the outbound link is followed, sending along the paper ("out *p Paper") it received at the start.

The WSDMtool interprets this object chunk and generates the following code for it:

```
def register_addcoauthors(request, p_id):
    p = get_object_or_404(Paper, pk=p_id)
    if request.method == 'POST':
        addcoauthorform = AddCoAuthorForm(p_id,
                                          request.POST,
                                          request.FILES)
        if addcoauthorform.is_valid():
            pn = addcoauthorform.cleaned_data['named']
            # CREATE: Person
            a = Person()
            # NEXT: Person_ID
            pid = 10
            # CREATE PROPERTY: coauthor
            a.save()
            a.coauthor.add(p)
            # CREATE PROPERTY: person_identified_by
            a.person_identified_by = pid
            # CREATE PROPERTY: named
            a.named = pn
            a.save()
            if request.POST['action'] == "save":
                return HttpResponseRedirect(
                           reverse('wsdmsite.views.register_information',
                                   kwargs = {'p_id': p.pk}))
            if request.POST['action'] == "save and add another":
                return HttpResponseRedirect(
                           reverse('wsdmsite.views.register_addcoauthors',
                                   kwargs = {'p_id': p.pk}))
    else:
        addcoauthorform = AddCoAuthorForm(p_id, {})

    return render_to_response('register_addcoauthors.html',
                {
                    'p': p,
                    'addcoauthorform': addcoauthorform,
                })
```

Once again a few interesting things can be discovered:

- The system interaction references introduce a new aspect which has to be kept in mind: persistency; when a new model instance is created in Django it requires to be saved at some point in the database. Updating the model instance does not cause any database access. Whenever a .save() method is called on the model instance, the database is accessed to save the model. This is required at the end of the business logic to perform the final save. Sometimes it is required that the object already has to be saved once, so it has a primary key, e.g. when a many-to-many relationship (a.coauthor.add(p)) has to be performed. The WSDMtool is able to figure out when this has to happen by sorting the generation order of the internal references.

- The AddCo-Author object chunk also shows what happens when there is more than one outgoing navigation link. The navigation path defines two outgoing links from the node containing this object chunk. The choice whether to follow the first or second link will be derived from which button was pressed in the form.

- In contrast to the previous object chunk the type signature contains an extra parameter. This is because this object chunk has an input reference, stating it requires a model instance of the type Paper. The code generator demands the primary key of the model when it is a NOLOT. As a first action in the controller it will retrieve the real model instance from the database using this primary key. In case of a LOT it will simply demand the lexical value.

5.4 Template Generation

Using the templateGeneration algorithm discussed in the previous chapter, we get the following code for the AuthorSubmissions object chunk:

```
<div class="authorsubmissions">
{% for object in authorsubmissions_list.all %}
  <div class="person">
    <div class="coauthor">
      <h4>coauthor</h4>
     {% for paper in object.coauthor.all %}
        <div class="paper">
          <span class="paper_identified_by">
            <b>paper_identified_by:</b> {{ paper.paper_identified_by }}
          </span>
          <span class="titled">
            <b>titled:</b> {{ paper.titled }}
          </span>
        </div>
      {% endfor %}
    </div>
    <div class="author">
      <h4>author</h4>
      {% for paper in object.author.all %}
        <div class="paper">
          <span class="paper_identified_by">
            <b>paper_identified_by:</b> {{ paper.paper_identified_by }}
          </span>
          <span class="titled">
            <b>titled:</b> {{ paper.titled }}
          </span>
        </div>
      {% endfor %}
    </div>
  </div>
{% endfor %}
</div>
```

This HTML shows how the algorithm traverses over the image and generates a hierarchical HTML structure. Furthermore a value is added to each DIV or SPAN tag to provide more flexible support for using style sheets.

The form class generated during the controller generation process is intantiated and sent to the following template where it is outputed as a collection of paragraphs using the .as_p method in the template.

```
<form enctype="multipart/form-data" method="post">
    {{ authorloginform.as_p }}
    <input type="submit" name="action" value="author's submissions" />
</form>
```

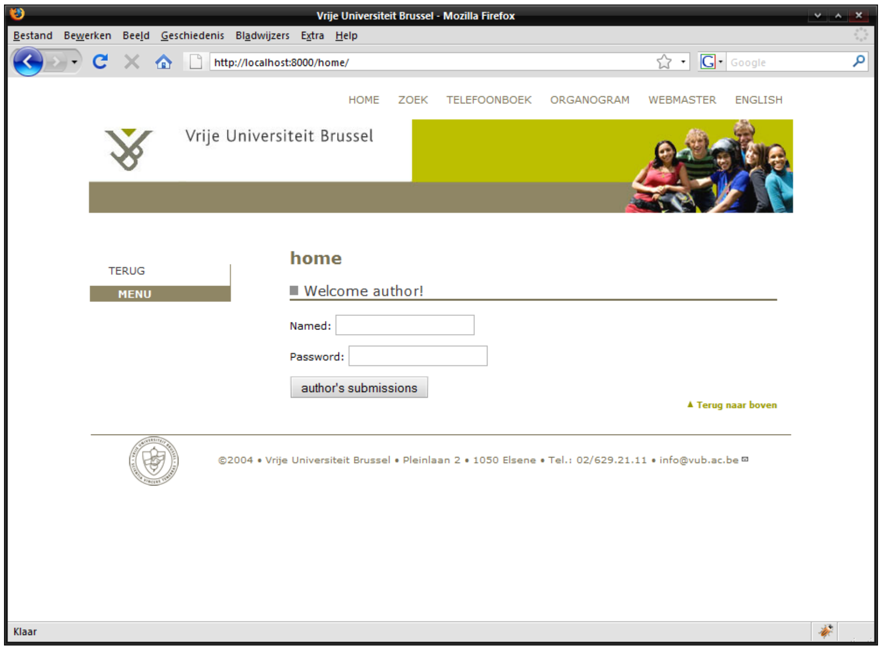
5.5 Routing Generation

Finally each page is generated as an entry in the urlpatterns variable, linking a regular expression representing possible URL requests to the controller function handling that request. Input references are extracted from the URL if necessary and passed to the function as arguments.

```
urlpatterns = patterns('',
    (r'home/', home),
    (r'register/(?P<a_id>\d+)/', register),
    (r'paper_update/(?P<p_id>\d+)/', paper_update),
    (r'register_addcoauthors/(?P<p_id>\d+)/', register_addcoauthors),
    (r'paper_update_addcoauthors/(?P<p_id>\d+)/', paper_update_addcoauthors),
    (r'paper_update_deletecoauthors/(?P<p_id>\d+)/',
         paper_update_deletecoauthors),
    (r'submissions/(?P<a_id>\d+)/', submissions),
    (r'register_information/(?P<p_id>\d+)/', register_information),
    (r'paper/(?P<p_id>\d+)/', paper),
)
```

5.6 Final Result

For the purpose of illustrating the result of the code generation process we have included the HTML representations of the AuthorLogin and SubmissionInfo object chunks.


Vrije Universiteit Brussel - Mozilla Firefox
Bestand Bewerken Beeld Geschiedenis Bladwijzers Extra Help
http://localhost:8000/home/
Google
HOME ZOEK TELEFOONBOEK ORGANOGRAM WEBMASTER ENGLISH
Vrije Universiteit Brussel
TERUG
MENU
author's submissions
▲ Terug naar boven
Klaar

home

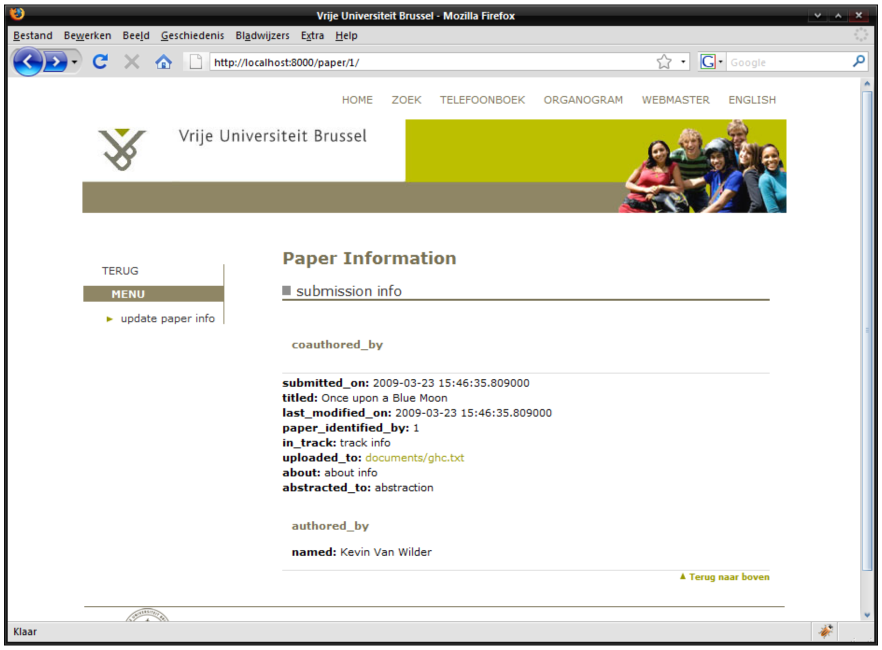
Welcome author!

Named:

Password:

©2004 • Vrije Universiteit Brussel • Pleinlaan 2 • 1050 Elsene • Tel.: 02/629.21.11 • info@vub.ac.be

Figure 5.3: AuthorLogin object chunk result


Vrije Universiteit Brussel - Mozilla Firefox
Bestand Bewerken Beeld Geschiedenis Bladwijzers Extra Help
http://localhost:8000/paper/1/
Google
HOME ZOEK TELEFOONBOEK ORGANOGRAM WEBMASTER ENGLISH
Vrije Universiteit Brussel
TERUG
MENU
▸ update paper info
▲ Terug naar boven
Klaar

Paper Information

submission info

coauthored_by

submitted_on: 2009-03-23 15:46:35.809000
titled: Once upon a Blue Moon
last_modified_on: 2009-03-23 15:46:35.809000
paper_identified_by: 1
in_track: track info
uploaded_to: documents/ghc.txt
about: about info
abstracted_to: abstraction

authored_by

named: Kevin Van Wilder

Figure 5.4: SubmissionInfo object chunk result

Chapter 6

Related Work

At the end of the '90s a lot of researchers started to recognize the lack of design methods for creating web sites and web-based information systems. A number of methods have been proposed. Some methodologies are data-driven whilst others are geared towards user-driven approaches.

In this chapter, we will present the most prominent web modeling methodologies currently in use and discuss how they deal with the implementation of the web site under design.

6.1 UML-based Web Engineering (UWE)

The UML-based Web Engineering approach [32] tries to adhere to the principle of using UML to describe a web application as much as possible. Its premise is the same as Model-Driven Engineering by using the meta-models and applying complex transformations to them. It combines various ideas found in other methodologies: such as a user-centered approach (WSDM), separation between navigation and presentation (OOHDM) and the usage of graphic notations (RMM).

UWE is a set of tools, notations and techniques, including a modeling language for graphically representing models, a formal definition of their meta-models, a development process and tools for semi-automatic generation of a web site based on the models.

The authoring process contains four phases for the creation of the following models: use case, conceptual, navigation space, navigation structure and a presentation model. Each iteration of the process refines these models further to provide more functionality.

The first part of the process is the creation of use case models. This is done by considering the application domain and defining the types of users as well as the functionality the application has to offer them. First the different type of actors have to be identified. Activities are then linked to the actors able to perform them. Activities logically belonging together are then grouped into use cases. An example is the submission use case in figure 6.1. Relationships between use cases and actors are established as well as inclusion and extension relationships between use cases. Finally the model is simplified by defining inheritance between actors and use cases. UML already contains a graphical representation to model these aspects.

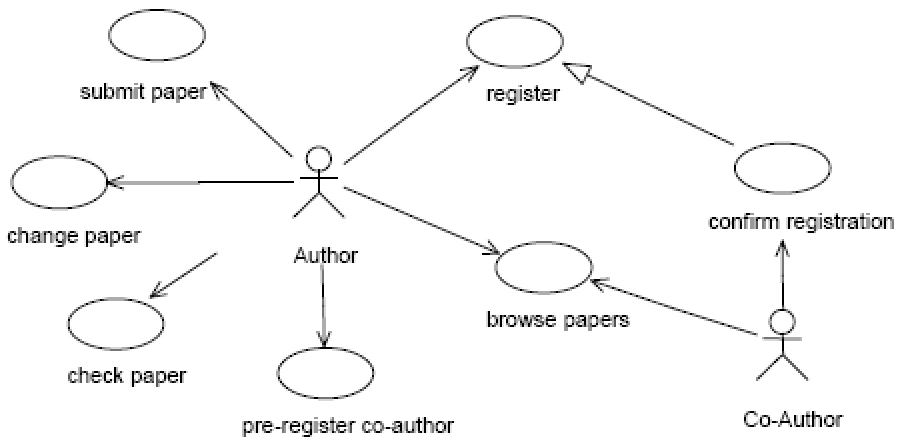


Figure 6.1: Submission Use Case Package in UWE [32]

After creating the use case model a conceptual model can be defined. This is done by using plain UML to describe the concepts in the business information model as UML classes. Alongside the common information found in these classes, such as a class name, attributes and methods, additional information for modeling adaptivity can be found; these are modeled using the UML comments. Finally, related classes are grouped together using UML packages. This is once again done using plain UML, requiring no additional semantics.

The navigational modeling phase generates two separate models: a navigation space model and a navigation structure model. The first one explains which concepts are able to be visited in the web application for each user type, whilst the second indicates the navigational structure specifying how to reach them.

The navigation space model, consisting of the objects that are reachable for a specific actor, contain navigation classes with the same names as the objects from the conceptual model. Using a separate model presents the possibility for restricting the permissions of a specific actor on certain attributes or methods of a conceptual object. Associations between the classes in the navigation space model represent direct links between the various objects. These navigation classes are represented by plain UML classes stereotyped as "navigation class". The associations are also stereotyped by "direct navigability". An example implementation [32] of this model can be seen in figure 6.2. This model is created by first including all conceptual models relevant to the navigation. Second, unnecessary information is omitted according to the actor's requirements and permissions. Associations between the models are kept and new ones can be added to improve direct navigability. Finally constraints are added to specify further restrictions.

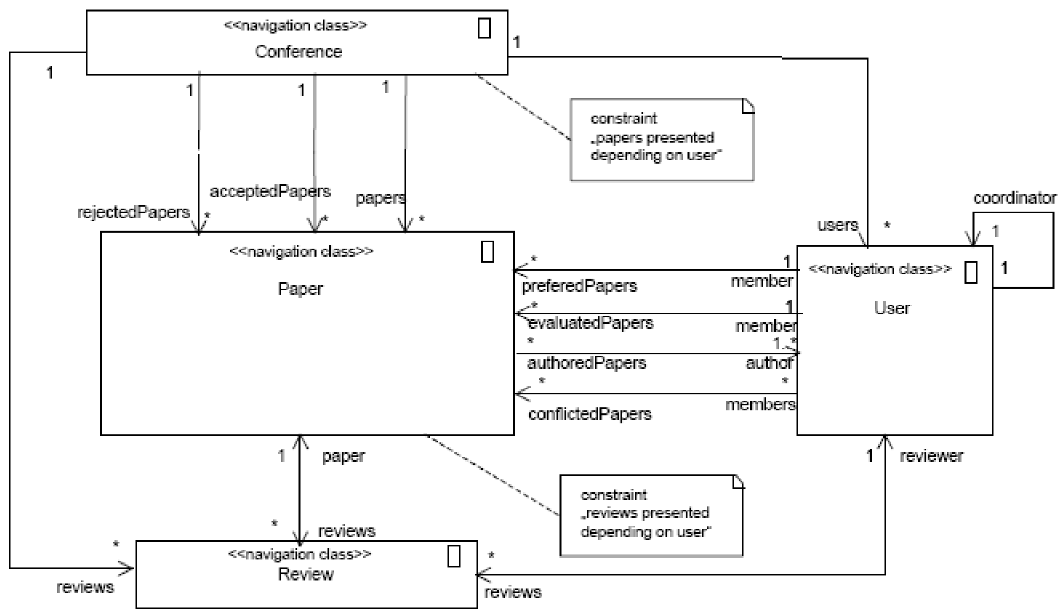


Figure 6.2: Navigation Space Model in UWE [32]

Modeling how the user will be able to navigate through the website is done by enhancing the navigation space model from the previous step by access elements, type indices, guided tours and queries. All bi-directional associations are replaced with a set of unidirectional associations representing the decomposed equivalent. Each association is annotated with an equivalent query required for retrieving the correct instances. This model is now called the navigation structure model.

One of these access elements or access primitives is a menu. This is used to represent different views on a model, such as a guided tour, index, a query, an instance of a navigation class or a composition of other menus.

The last model is the presentation model which is created through the use of storyboarding, a technique used to give a first look and feel of the user interface, similar to a collection of mock ups. The meta-model in figure 6.3 describes the relationship among the various classes in the presentation model.

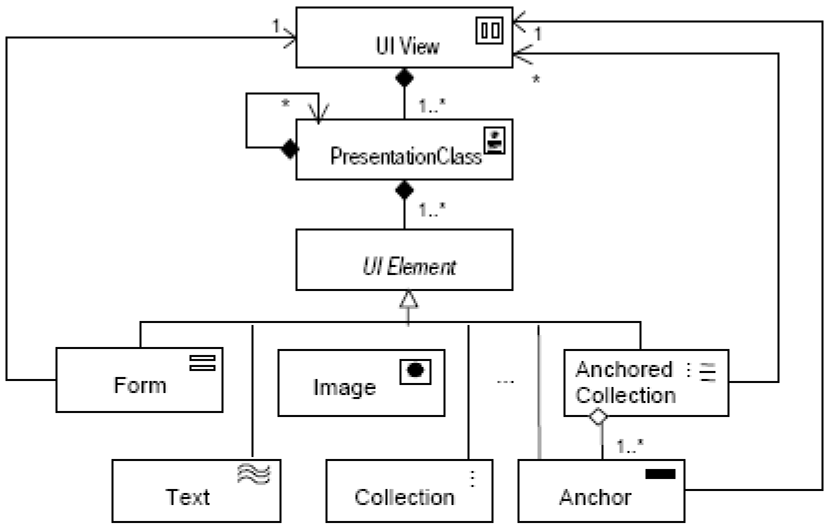


Figure 6.3: Meta Model for Abstract User Interface Elements [32]

Now we have elaborated on the methodology itself it is time to present the tool used for modeling and generating a website. ArgoUWE [31] is a plugin for the open-source UML modeling tool ArgoUML which provides the methodology with its own CASE tool for modeling and generating towards the Spring framework.

Kraus, Knapp and Koch explain [32] that UWE-based generation of web applications is straightforward in the case of content and presentational modeling. The conceptual model is easily translated into its implementation specific counterpart. Similarly, the UML-based presentation model can be seen as a decomposed abstraction of a web page. However ArgoUWE does not provide a full fledged automated generator. for example, the implementation of model methods still has to happen manually.

The generation of platform specific code is handled by a transformation pipeline using ATL (ATLAS Transformation Language).

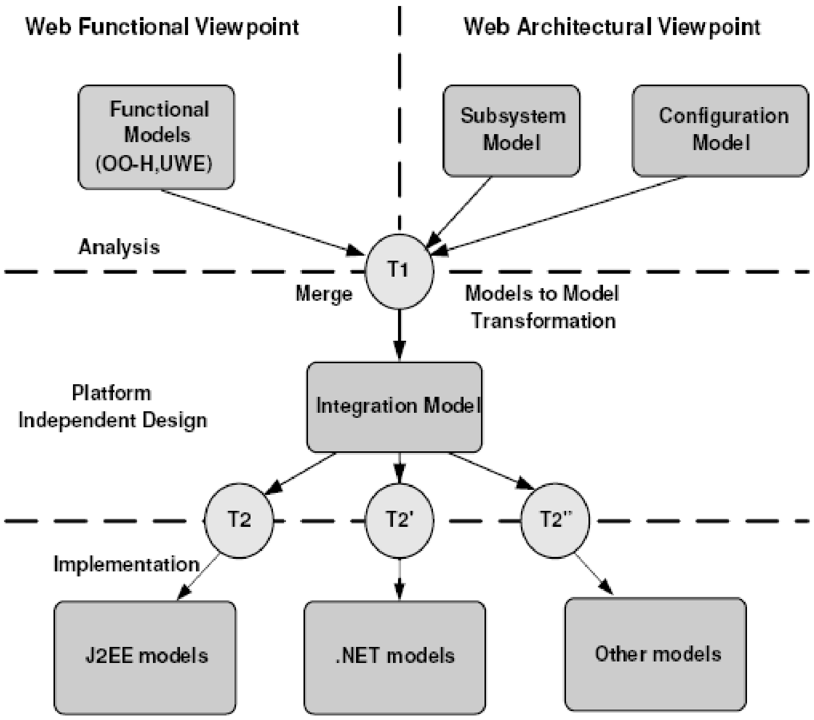


Figure 6.4: Transformation Pipeline [32]

Figure 6.4 represents the three-layered transformation pipeline. The first (top) layer contains content independent models (CIMs) describing the meta models. The middle layer contains the platform independent models (PIMs) describing a website using the CIMs from the first layer. Finally the bottom layer contains the platform specific models (PSMs) which implement the web site for a specific framework.

However it is not straightforward for modeling the business logic behind the website, based on the navigation model. The typical transformation pipeline used by UWE does not provide enough support for transforming the workflow notations. An interpretational approach [32] has been chosen for this part.

6.2 Object-Oriented Hypermedia Method (OO-H)

The Object-oriented Hypermedia methodology, or "OO-H Method", is a design proposal [28] for modeling web sites based through an object-oriented approach.

OO-H follows the precept that the modeling techniques used for traditional software engineering can be successfully applied for also developing web applications. As a result OO-H is an extension to the traditional UML-modeling technique [5] commonly used in software engineering.

The proposal enables the developer to create web applications by providing a set of notations, tools and techniques which complement the UML, such as: a design process, a pattern catalog, a Navigational Access Diagram (NAD), an Abstract Presentation Diagram (APD) and a CASE tool that allows automation of the development of web applications using this methodology.

The design process lays out a sequence of phases the developer must cover in order to create a functional interface:

1. A first step is to create a UML-compliant class diagram that specifies the domain information structure.

2. A NAD is constructed for each user type.

3. Generate a default APD.

4. Refine the APD, the designer refines the structure and usability for the different interface modules.

5. Finally the website interface is automatically generated.

The NAD and APD make extensively use of the OO-H Method Pattern Catalog. This is a collection of patterns which can be used throughout the development. The patterns focus on best practices to address certain problems, similar as traditional software engineering. The catalog is divided into three categories:

- Information patterns that provide feedback for the users

- Interaction patterns provide best practices in handling user-interface communication issues, both functional and navigational.

- Schema Evolution Patterns which cover more structural advanced features

A pattern itself is composed of a name, its application level (specifying whether it can be used in APDs or NADs), a context, problem statement, verbose solution, default implementation defined through transformation rules and secondary implementations.

The navigational access diagram or NAD is built upon the UML-compliant class diagram from step 1. The combination of all defined NADs will result in the navigational model for the given site. For each view required by the specifications a separate NAD should be constructed. Furthermore each user-type should have their own set of NADs, since most of the time different user types will require different specifications.

The diagram is made out of four types of constructs:

- Navigational classes enrich the domain classes by restricting the visibility of the methods and attributes in accordance to the user's access permissions. This can be done using three types of attributes: V-attributes (visible), H-attributes (hidden) and R-attributes (referenced, i.e. visible at the demand of the user).

- A Navigational Target (NT) is the structure to group various elements of a model for an NAD.
- Navigational Links (NL) define the paths the user will be able to follow. These links are divided into a couple of categories: Internal Links (Li) define the structure inside an NT (comparable to process-logic links in WSDM), Traversal Links (Lt) for creating links between different NTs (similar to structural links in WSDM), Requirement Links (Lr) point to the start of the an NT and Service Links (Ls) show what services are available to the user-type.
- Collections are structures which help the user with accessing information.

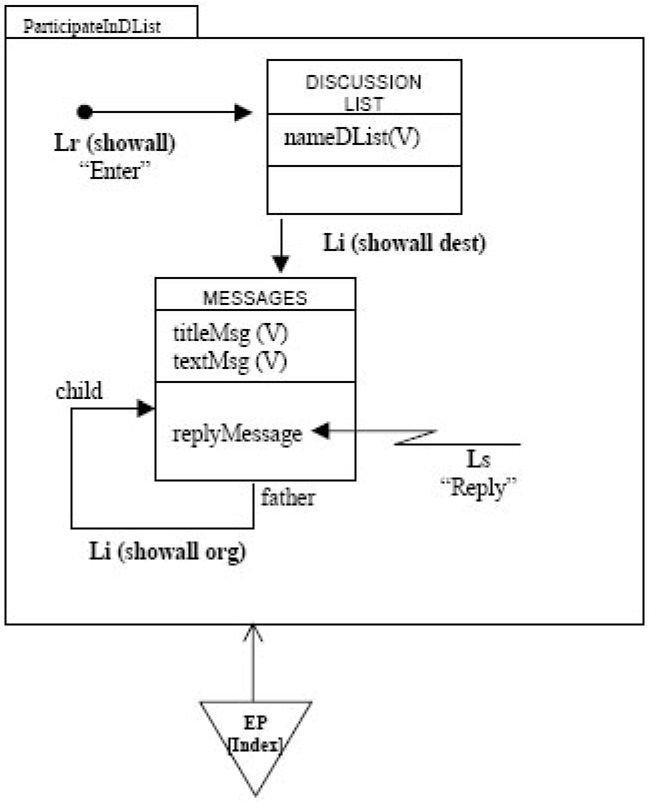


Figure 6.5: An NAD of the Discussion List System [28]

The NAD as presented in figure 6.5 has its attributes annotated with V-attributes, which means that the information contained in nameDList (the name of the discussion list), titleMsg (title of a message) and textMsg (text body of a message) in all messages within a discussion list will be visible to the user.

The Internal link between "Discussion List" and "Messages" states that the NT should show all ("showall" modifier) messages belonging to that discussion list instance, but that they will appear on a separate page than the name of the discussion list, denoted by the "dest" modifier. Also a Service Link is defined to allow the user to reply to a message by pressing a hyperlink called "Reply".

Finally the Classifier Collection pointing to the NT specifies that this particular NT will be chosen as the entry point (EP) for the web application, called "Index".

Although the OO-H tool generates basics Abstract Presentation Diagrams, they would require refinement to add more sophistication and usability to the interfaces. These templates are expressed using XML. To specify the type of the template a couple of Document Type Definitions (DTD) have been defined, such as structural, style, forms, client functionality and a composite-like window-type which can hold multiple views.

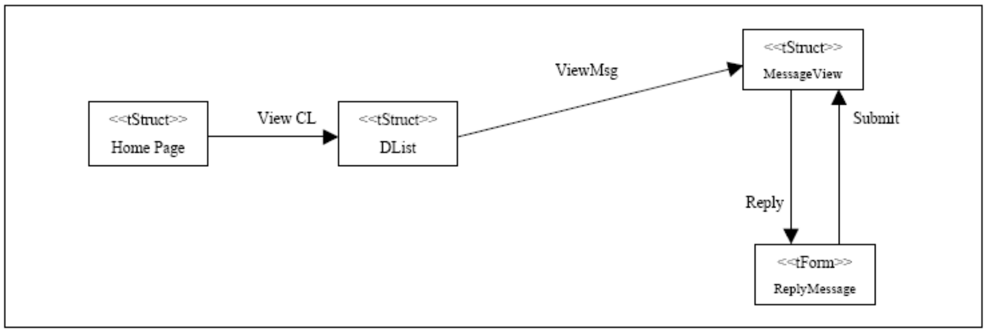


Figure 6.6: An APD for the Discussion List System [28]

As mentioned the tool generates a usable initial version of the APDs; one of these is shown on figure 6.6. This figure describes that everything except the "ReplyMessage" action should be presented using a structural template. For the "ReplyMessage" action, the user should be able to fill out information, such as the title and body of a message and thus requires a Form template. We can also see that the links, defined in the NADs, are also present.

This APD is generated through a series of basic APD Mapping Rules [28]:

- V-attributes, I, T and R-links appear as elements inside a tStruct template.
- C and S-collections are represented through a tree-like composition structure consisting of tStructs and tForms, depending on the need for user-input or not.
- S-Links generate tForm pages so the user is able to input information into form fields. Depending on the return-type of the method the link is pointing to, a page might be generated to present the user with the final result.
- R-attributes also create tStruct pages which are linked together through hyperlinks, referenced by a value entered in the model.
- Depending on the patterns, a different type of page can be generated. For example the ShowAll Pattern will create link elements on a single page.

This initial APD version provides a usable, but rather simple implementation. It is therefore possible to refine this diagram by using constructs available from the Pattern Catalog.

6.3 Web Modeling Language (WebML)

The Web Modeling Language [12] was introduced in 1998 as a visual notation for creating large complex data-intensive web sites. Similarly to WSDM, its graphical notation is translated to a formal specification which is used by a visual CASE tool such as WebRatio.

WebML differs from other methodologies by providing only a limited number of concepts which can be combined together in various ways to offer a large flexibility. Another trait is the close connection to the industry: alongside the academic research around WebML an industrial line developed the WebRatio tool utilizing the methodology. This tool has become the most prominent and actively used web site modeling tools in real-world situations.

The first version of the method only provided support for modeling read-only websites with a lot of data. The focus was deliberately kept small in order to create a decent foundation for modeling the organization of the interface, navigation and content. Version two added descriptors for business logic to the method in order to model read-write web applications containing authentication, content management etc. A third iteration of the methodology implemented the possibility of creating model plug-ins, so developers could add their own primitives to the methodology for modeling more specific business-domain related problems. The fourth version used this newly introduced plug-in system to introduce new extensions to the core of the methodology, providing support for web services and distributed applications.

The methodology is based on an iterative process [11]. Each cycle has a requirements analysis followed by an implementation through data design. After this a prototype of the partial web site is created and undergoes testing and validation, returning back to the original requirement analysis for further refinement and additional functionality for the next iteration. This process is ideally suited for the Web since it follows a "deploy fast, deploy often" mentality. The requirement analysis process consists of collecting information about the business domain and specifying the functionality and target group of the web application. The methodology does not specify which format has to be used to create these specifications; it leaves this up to the designers.

First the types of users are identified. Similar to WSDM they are grouped together according to the same characteristics and access rights. The user groups are then complemented by with their respective functional requirements. After this a phase called "identification of core information objects" extends the information about the user groups once again by deciding what information the user group is allowed to access and manipulate.

The phase is ended by decomposing the web application into site views for each user group, i.e. "creating different hypertexts designed to meet a well-defined set of functional and user requirements" [12].

During the conceptual modeling phase a Data, Hypertext and Business Process Model are defined.

A Data Model is created using either an entity-relationship diagram or an equivalent subset of the UML class diagram to model the relationship between the various entities, i.e. domain concepts. These relationships can be extended with constraints and cardinalities.

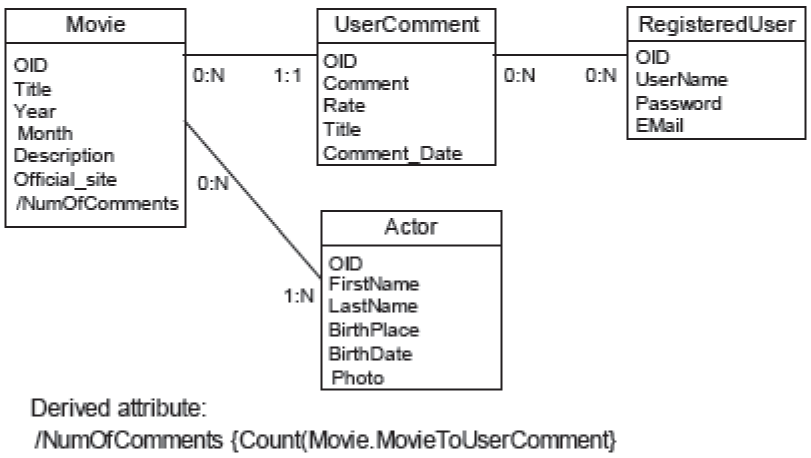


Figure 6.7: WebML Data Model using UML [12]

The Hypertext Model provides an initial definition of the front-end, specifying what the user is able to view. Data and operations are divided into components, called control units, which are in turn organized in pages. This is complemented by the addition of links between these pages and control units. Combining all these aspects create a site view, which is targeted

Each page in a site view can be a combination of any three types:

- The home page, denoted by an "h", is the page which will be entered when browsing to the web site.
- A default page, denoted by a "d", is the page presented when the user enters the enclosing area.
- A landmark page, denoted by "l", is an additional navigational link which is reachable from all pages or areas in the enclosing area.

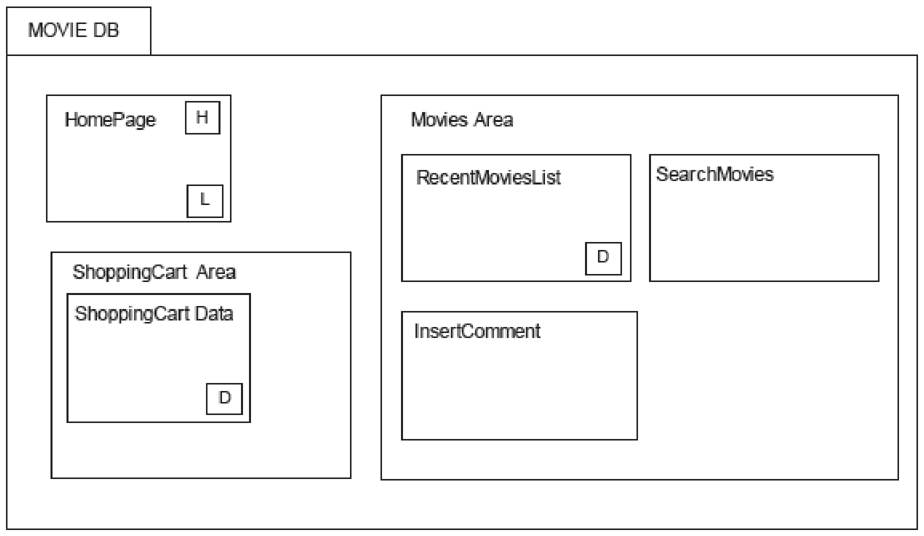


Figure 6.8: WebML Hypertext Model [12]

Figure 6.8 depicts the site view "Movie DB" made up as a hierarchical structure. At the top level the site consists of a home page and two navigation area (the shopping cart and movies areas). Each area consists of a collection of pages where at most one is annotated as the default page for that area. Each page is composed of various content units, representing elementary information elements. There are five predefined types of content units:

- Data Units represent a collection of attributes of a given entity instance.
- Multidata Units does the same for a set of entity instances.
- Index Units generate a list of index units from a set of entity instances allowing the user to select specific instances from that list.
- Scroller Units allow browsing of an ordered set of objects
- Entry Units create forms allowing users to input values into the system.

A Business Process Model, consisting of a UML workflow diagram, was added in version 2 to allow the modeling of user operations, i.e. data manipulation. It defines the activities, precedence constraints and the user roles in charge of executing all operations.

6.3.1 Automatic Generation using WebRatio

WebRatio is a model driven development environment for modeling web sites using the WebML method and automatically generating the resulting web site. It was developed by Web Models, a spin-off of the Politecnico di Milano[35], established in 2001. The generated web application is Java-based and can thus be installed on any web platform.

WebRatio consists of a two-layer architecture: the design layer and the run-time layer. The design layer contains the graphical user interface for visually modeling the data and hypertext models of the web site. It also includes a data mapping module that maps the Data Model over pre-existing entities and relationships to support legacy databases and a final module that provides the designer with functionality to define the presentation style and organization of the data on the page using XSL style sheets for XHTML pages.

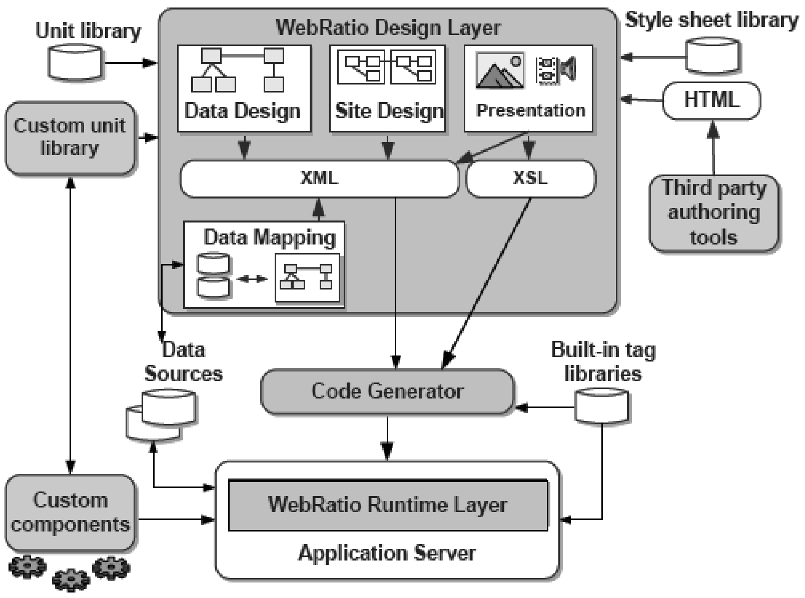


Figure 6.9: WebRatio Architecture [12]

These visual notations are translated to XML specification by the design layer. It is this specification which is passed to the run-time layer to be used to generate the website. The XML specifications are then translated into application code using XSL transformations in the run-time layer. This code can then be executed by the Jave2EE platform.

The main idea of the generation process is as follows: the XSL transformation process produces dynamic page templates and unit descriptors. A dynamic page template is a JSP file which contains the content and the markup of a specific page in the web application in a given markup language (HTML, XML, RSS, etc.). Unit descriptors are XML files contain the database interactions required for populating the dynamic page template.

Both layers in the tool have been extended to allow the addition of plug-in modules to increase the expressiveness of the tool allowing additional concepts to be modeled as discussed before.

6.4 Object-Oriented Hypermedia Design Method (OOHDM)

The main focus of OOHDM lies on navigating hypermedia in contrast to the audience-driven approach used in WSDM. Building a web-application with OOHDM is done by creating "navigation objects" that represent a view (i.e. queries) on the conceptual objects. Navigational contexts are introduced to abstract and organize the navigational space between these various objects. The interface is completely separated from the navigational space definition.

OOHDM divides the development process [43] into 4 activities: conceptual design, navigational design, abstract interface design and finally the implementation. This choice was made since the creation process of web-based applications is not intrinsically different from conventional software engineering, which is generally made up by the following activities: analysis & modeling, design, implementation, testing and maintenance. The model is built and enriched incrementally by applying a mix of iterative and incremental development styles.

The conceptual design phase is similar to most methodologies since it can be generalized to obtaining the requirements and determine the Universe of Discourse (UoD). This UoD is modeled using an object-oriented approach containing extra annotations. The model created during this phase will contain the conceptual objects which will later be used by the navigational objects and secondary objects that define the computational aspects of the application, such as algorithms and database access.

The Navigation space will be constructed during the navigational design phase. This phase will introduce the navigation objects, which are customized views on the conceptual objects. This is analog to the controller-functions from the MVC-pattern, providing a specific view on the information, i.e. the conceptual objects). These objects are then divided into sets called navigational contexts.

As with the MVC-pattern, the navigation objects are not directly perceived by the user, but rather through interfaces defined in the abstract interface design phase. These interface models decide which objects the user will be able to view and how the navigation between them is activated.

Finally the implementation phase will bring all these parts together, mapping the conceptual objects, navigation objects and interface object over a runtime environment. More specifically this means creating the HTML pages, scripts and queries. This will represent the final result of the methodology. .

6.4.1 HyperDE Rapid Prototyping Environment

For some time the center of attention of academics has shifted from web methodologies to semantic web methodologies. As a result, OOHDM gained a successor called Semantic Hypermedia Design Method (SHDM). It is for this methodology that a tool for automatic implementation was built, called HyperDE [4]. Figure 6.10 depicts an abstract view of how the new methodology is defined and executed.

As a result of the close resemblance [35] to the MVC-pattern, the mapping between the objects and the pattern can be made. Existing web frameworks primarily utilizing this pattern are therefore interesting candidates for the implementation generator. The generation of models can be based on the UML diagrams, specifying the various models, its attributes and the relations between them. Similarly the interface models can be generated to view components and the navigation nodes and interface behaviors to the controller components. Having commonalities with MVC-patterns, it was understood [34] that generation towards a web framework such as Ruby on Rails would be very interesting. Ruby on Rails [33] was chosen over various other frameworks due to its position as market leader in its category. Its persistence layer, implemented as an Active Record-pattern, has been replaced by one that is based on a Sesame [13] RDF triple store. All information such as the navigation models and instances of the UoD are stored in this database.

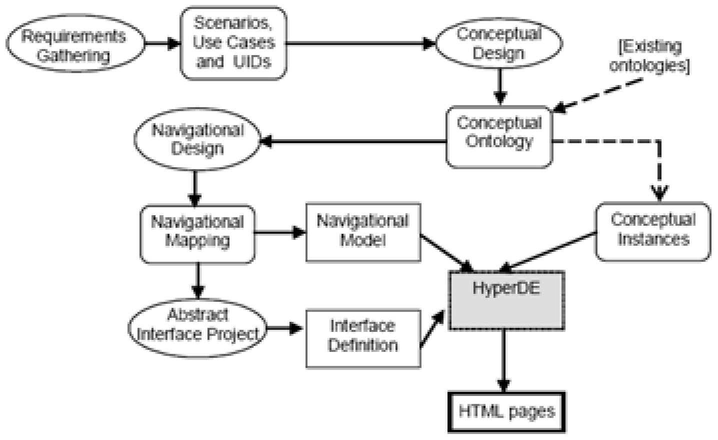


Figure 6.10: Workflow of SHDM and HyperDE [33]

HyperDE is implemented as a so-called MNVC framework, which extends a framework based on the MVC-pattern by adding extra models for describing navigation. It allows the user to input models created according to SHDM and will generate a complete application adhering to the specifications.

This advantageously affects the portability of HyperDE, since Ruby on Rails only requires an interpreter for the Ruby [8] programming language to be available on the deployment server. A disadvantage of this approach however is the use of a specific version of the Ruby on Rails framework which used as a base for the implementation. It is likely that newer versions of the framework, containing security patches etc., will appear and would require to be manually merged into HyperDE in order to remain secure.

In order to generate towards Ruby on Rails HyperDE extensively uses Domain Specific Languages (DSLs) [34]. This combined with a Model Driven Development approach allows the developer to generate code by simply manipulating the models that make up the application.

SeRQL is used to directly query the Sesame triple store. Because of the assumption that the users of HyperDE might not be familiar with such a specialized language such as SeRQL, they have added a new layer on top of SeRQL containing a ”simple DSL” to abstract. This has an added benefit that the underlying database can be switched and the new querying language would simply need to be mapped to the DSL layer.

As we already mentioned in the study of Ruby on Rails and other web frameworks, the framework communicates to the database through the use of models. HyperDE generates these models for us with the use of another DSL. It does this in the following way:

- Instances of the NavClass become Ruby model classes.
- Each NavAttribute inside a NavClass becomes an attribute in the corresponding Ruby class. Extra methods, such as "find_by_" are defined to enable the framework to look up information with the class.
- NavOperations become methods in the Ruby class and represent the controller part of the MVC pattern.
- Links containing a NavClass as the source will generate an array-attribute in that Ruby class.

This covers the model and controller aspects of the MVC-pattern required by Ruby on Rails. The templates representing the View aspect of the pattern are a combination of HTML interspersed with Ruby on Rails. A new DSL in introduced to specify the generation towards these templates, e.g.

```
<input type="text" name="new_email" value="<%=
@node.email %>">
<%= op "change_email", { :label => "Change
Email", :view => "attributes", :update =>
"node_attributes", :label_loading => "wait..." },
[ '<input type=button value="%s" onclick="%s">',
:label, :onclick ] %>
```

This code [33] specifies the generation of an HTML form interspersed with Ruby using HyperDE's pre-defined templates.

While HyperDE's approach closely resembles the one taken with WSDMtool, it differs greatly in terms of flexibility. Due to the fact that HyperDE subsumes Ruby on Rails, it is inherently linked to this web application framework and lacks the possibility of generating towards other frameworks. Another difference is the fact that HyperDE's approach requires modifications to the Ruby on Rails framework.

6.5 Other Web Methodologies

Other methodologies which will not be discussed in detail include Hera [3], a collaborate effort by various Dutch and Belgian universities. This is a model-driven design approach for context-dependent and personalized web information systems. Its main characteristic is the focus on adaptation of web sites towards the user.

Another method is the Object Oriented Web Solution [36] (OOWS) methodology, which is based on OO-H. The OOWS method improves OO-H by introducing new models for representing presentational aspects and navigation.

6.6 Conclusions

The need to create methodologies to simplify and formalize the process of creating web applications has been filled. Whilst some methodologies are only used on an academic level, some have found their way into businesses.

Although their approaches may vary, the method proposals have many similarities. All methodologies discussed make it clear that they should be complemented by a computer aided development tool that supports their methodology. It is also apparent that each method has a clear separation of the content, navigation and presentation/design part, mostly expressed in separate phases and data structures. Finally all methodologies agree that the creation of web application should be platform independent and that the step towards platform dependent code should be done as a last phase. The usage of implementing towards an MVC-pattern in the web application creation process described by other methodologies validates our abstraction conclusions taken in Chapter 3. The process for generating Ruby on Rails models in HyperDE has large similarities to our approach.

Chapter 7

Conclusions and Future Work

7.1 Summary

In this dissertation we took the first steps towards implementing WSDMtool for generating web sites using the Web Site Design Methodology.

In chapter 2 we first introduced the WSDM methodology, the focus of this dissertation, its models and their formal definitions, the WSDM ontology. This ontology will be used to formally define web sites which will be read in by WSDMtool.

By inspecting various existing web frameworks in chapter 3 we extracted the similarities to create an abstract framework. WSDMtool uses this abstraction to ensure platform independence, whilst providing as much implementation as possible to shorten the development time for platform specific code generation modules.

Chapter 4 introduced the WSDMtool architecture and discusses the various choices made during the development process. It also describes the implementation process of the Django code generation module.

To demonstrate the functionality and features of the WSDMtool we designed a web site using the Conference Review System case study in Chapter 5 and present our results and findings.

Finally in chapter 6 we introduce various other methodologies and their approaches taken towards automatic code generation.

7.2 Future Work

Future work of WSDMtool includes not only improvements to each generation process, but also creating new modules for generation towards other frameworks as well as introducing a tool combining conceptual design and automatic generation. These will now be discussed in more detail.

7.2.1 Better Model Generation

The current implementation of the database generator for Django provides functional yet limited model definitions. A first limitation is the usage of many-to-many relationships to express the relationships between the models. This type supports the various sub-sets such as one-to-many and one-to-one relationships. The correct cardinality constraints for each individual relationship would be preferred. It is possible to extract this cardinality constraint by iterating over all the object chunks containing the statement specifying the relationship chose the least restricting statement to base the model relationship on.

Because of the usage of tiny conceptual schemas in the views, it is possible to model the same information using different concepts. For example: an ”Full Name” concept could exist in one object chunk whereas it is defined as ”First Name” and ”Last Name” in another. It should be possible for the code generator to understand these subtleties. This problem has already been addressed in [23] and [24] by defining additional steps during the conceptual modeling phase. One of these steps is annotating the concepts in the ontology with pre-defined relationships such as equivalency, sub-types, overlapping and combination relationships.

Finally it is currently also not possible to specify which model fields are allowed to contain a null value or not. All model fields allow the null value by default. Once again it is possible to extract this requirement by going over all object chunks and inspecting the specific statements.

7.2.2 Semantic Links

WSDM divides navigational links in four disjoint categories: process-logic, structural, navigational aid and semantic links. Currently the WSDMtool supports the first three types of links. The semantic links were not implemented due to the lack of a correct formal notation in the WSDM ontology. When a formal notation is decided upon it will be possible to create the routines for generating the semantic links.

7.2.3 Usability and Presentation

In the WSDMtool architecture chapter we brought forth the notion of internal references to model the ORM extension proposed by the WSDM methodology. These internal references were then subdivided into three categories: user, system and data-lookup references. A data-lookup can generate two different results: a collection of objects found or a value representing no objects could be found. In Django such lookups generate ObjectNotFound exceptions. WSDM does not provide a method to model scenarios when no objects could be found. Currently the WSDMtool handles these scenarios by generating an HTML 404 error page along with the message that no models could have been found.

In the same chapter, we formulated arguments as to why presentation and content should be handled separately. The current WSDM ontology however contains notations to specify specific presentational aspects such as multi-columned designs. This could still be used to create a hierarchy of template files using Django's template inheritance. This also provides an opportunity to implement a formalization of defining so-called "static object chunks", i.e. semantically annotated static pages. Practical examples of this can be "About Us" web pages. This page could be annotated using existing ontologies such as FOAFCorp [2].

The generation of Django form classes is currently limited and does not provide full support for WSDM user interaction references. For example: a "First Name" concept referenced by "*fn = ?" will generate the same form as when it would contain a reference such as "*fn = ??" or "*fn = !". It also does not support the "->" notation (e.g. "*fn -> ?") for updating values. This was done with the assumption that assignments can also be value updates.

A final suggestion can be the development of a full-fledged CASE tool allowing the user to graphically create the object chunks and the navigation diagram and incorporate the WSDMtool which automatically generates the given website for these models.

7.3 Conclusion

In this dissertation we took the first steps towards implementing a tool for generating web sites using the Web Site Design Methodology.

We conclude that the The WSDM methodology is a viable approach to modeling ubiquitous web applications. The formal OWL specifications have been extended where necessary to allow unambiguous interpretation of the object chunks. This enabled us to create the algorithms that make up the generation tool.

Appendix A

Author Track for the Conference Review System

A.1 Navigation Track

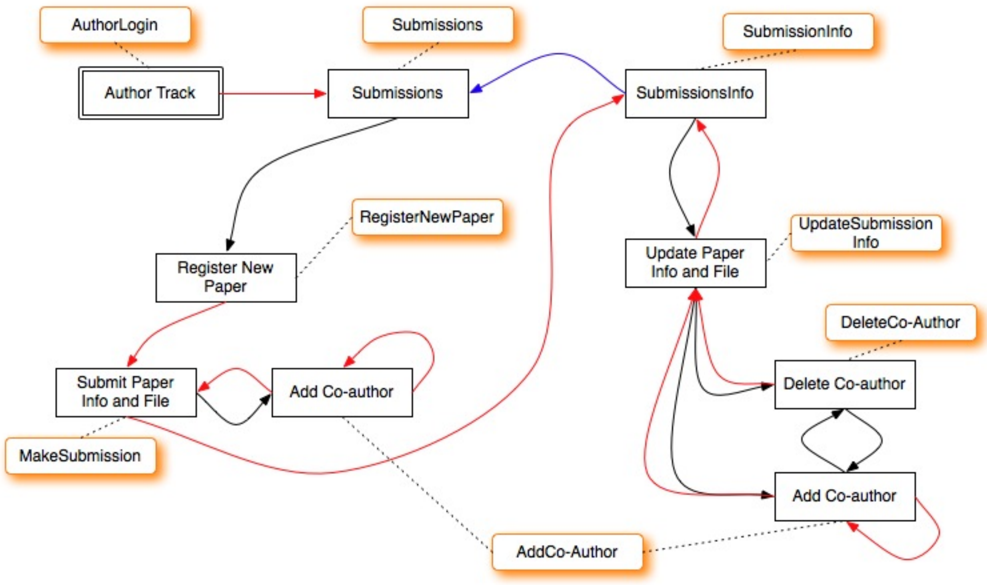


Figure A.1: Author Navigation Track

A.2 Object Chunks

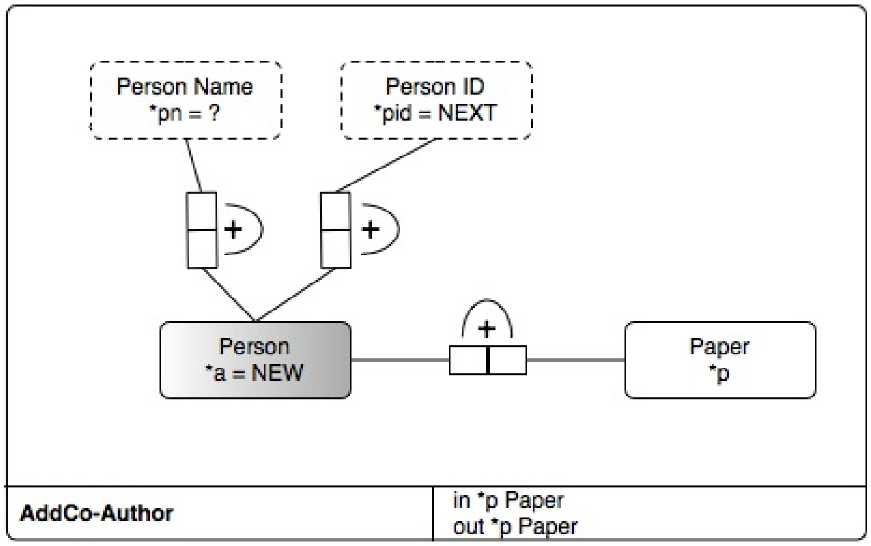


Figure A.2: AddCo-Author Object Chunk

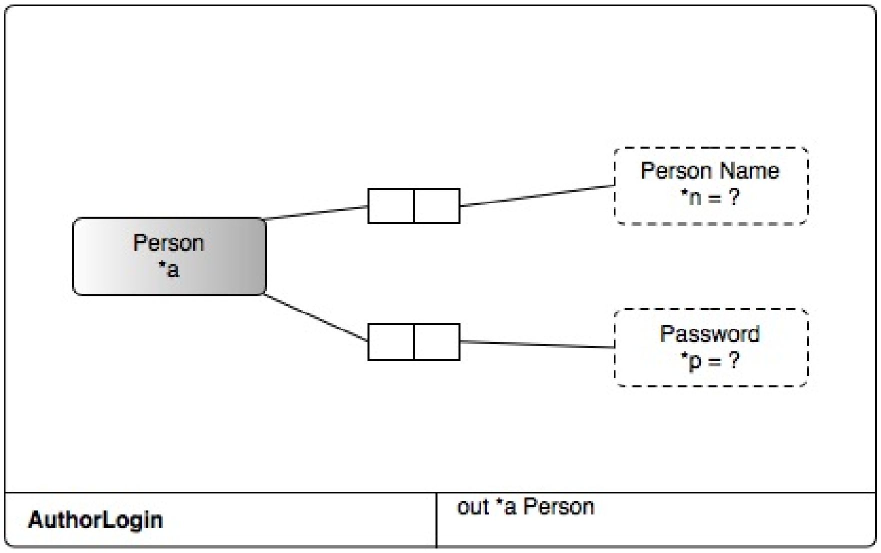


Figure A.3: AuthorLogin Object Chunk

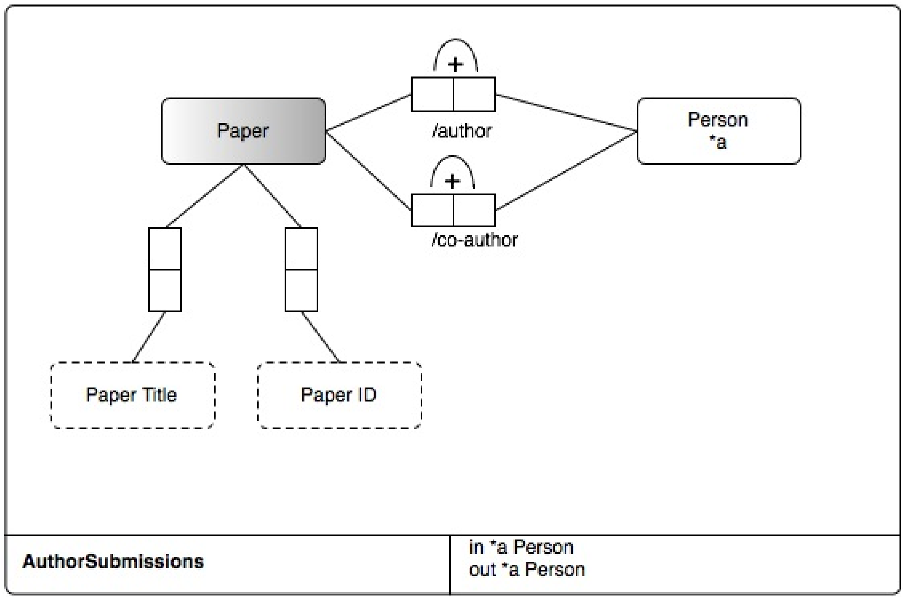


Figure A.4: AuthorSubmissions Object Chunk

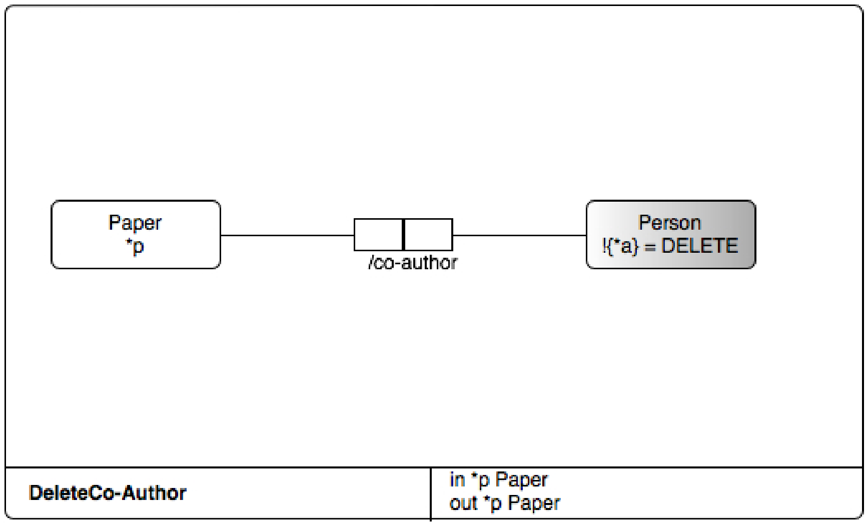


Figure A.5: DeleteCo-Author Object Chunk

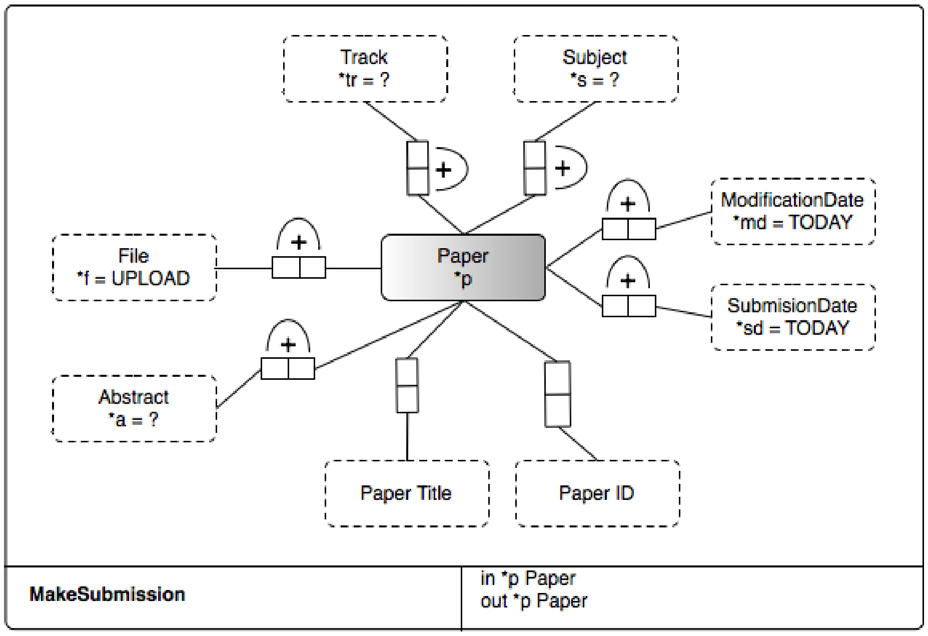


Figure A.6: MakeSubmission Object Chunk

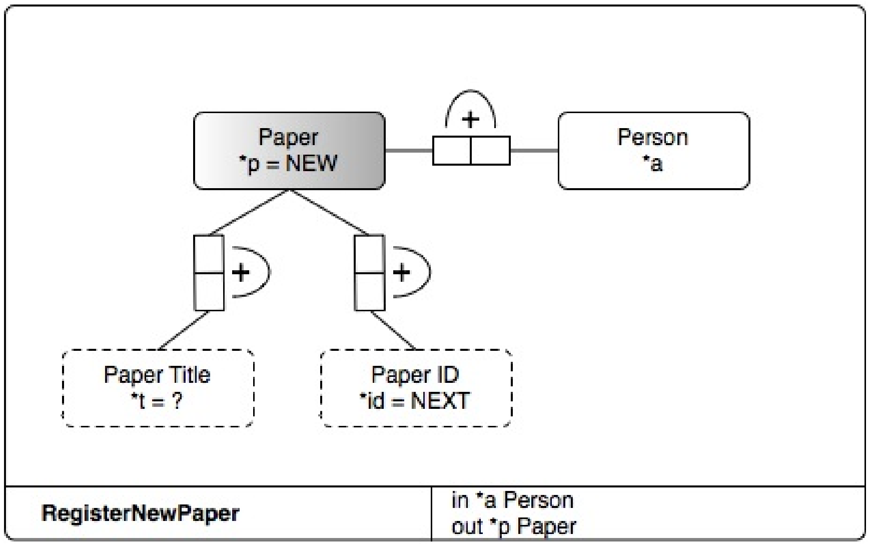


Figure A.7: RegisterNewPaper Object Chunk

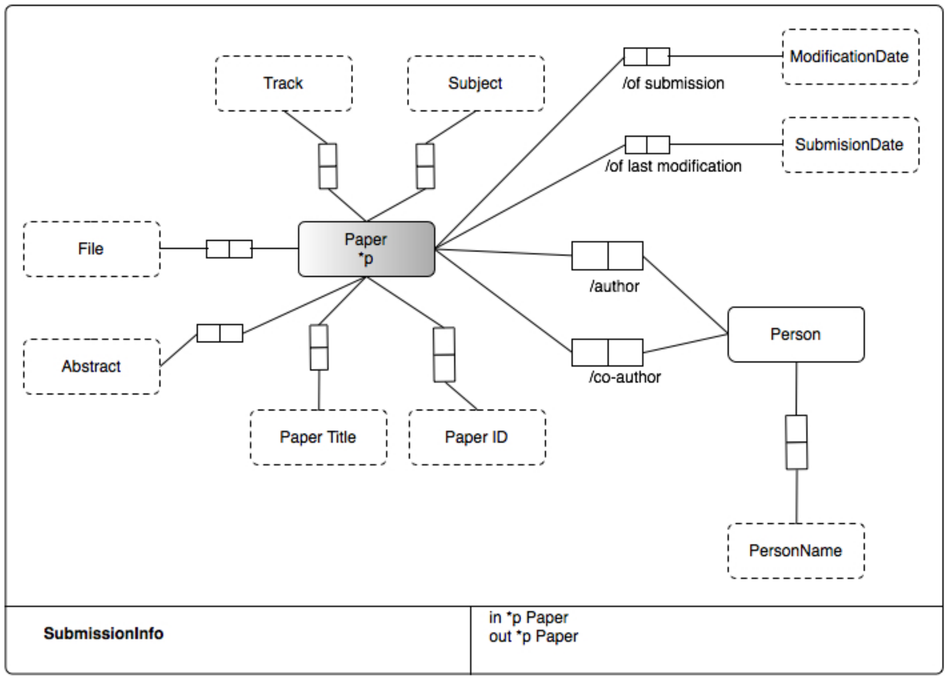


Figure A.8: SubmissionInfo Object Chunk

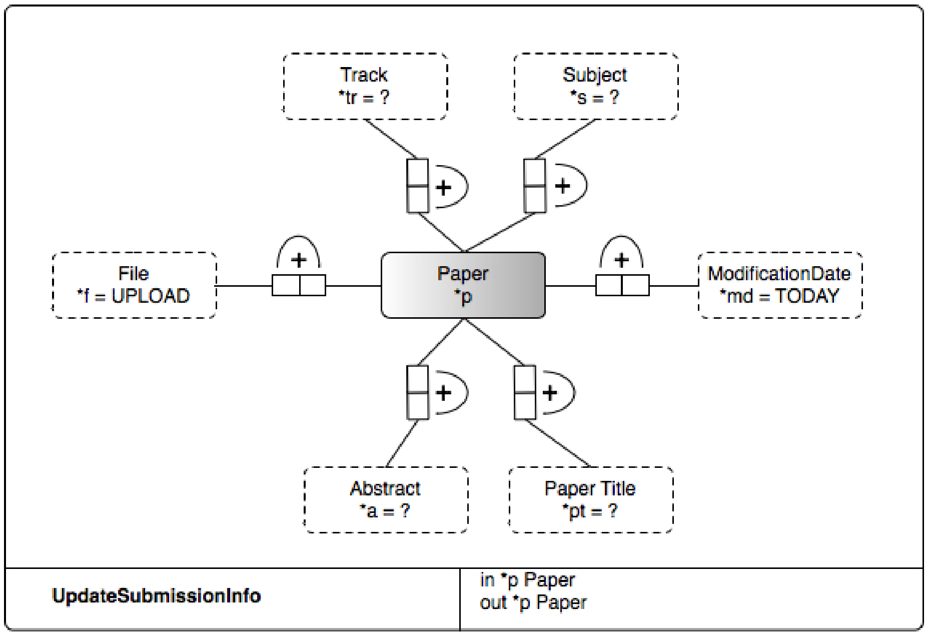


Figure A.9: UpdateSubmissionInfo Object Chunk

Appendix B

Comparing WSDM description to MVC-pattern

B.1 Model Comparison

B.1.1 WSDM

```
Paper, locatedAt, File(Resource)
Paper, submittedOn, SubmitDate(Date)
Paper, lastModifiedOn, ModDate(Date)
Paper, hasMainAuthor, Author
Paper, hasCoAuthor, Author
Paper, hasAbstract, Abstract(String)
Paper, hasTitle, Title(String)
Paper, hasId, PaperID(Integer)
```

B.1.2 Django Representation

```
Class Paper(Model):
    locatedAt = FileField()
    submittedOn = DateField()
    lastModifiedOn = DateField()
    hasMainAuthor = Many2ManyField(Author)
    hasCoAuthor = Many2ManyField(Author)
    hasAbstract = TextField()
    hasTitle = TextField()
    hasID = IntegerField()
```

B.1.3 Ruby on Rails Representation

```
class Paper < ActiveRecord::Base
    has_many :authors,
        :through => :hasMainAuthor
    has_many :authors,
        :through =>:hasCoAuthor
end
```

B.1.4 CakePHP Representation

```
Class Paper extends AppModel
{
var $name = "Paper";
var $hasMany = array(
    'hasMainAuthor' => array(
        'className' => 'Author'
         ),
    'hasCoAuthor' => array(
         'className' => 'Author'
         )
     );
}
```

B.2 Controller Comparison

```
Paper, locatedAt, File(Resource)
Paper, submittedOn, SubmitDate(Date)
Paper, lastModifiedOn, ModDate(Date)
Paper, hasMainAuthor, Author
Paper, hasCoAuthor, Author
Paper, hasAbstract, Abstract(String)
Paper, hasTitle, Title(String)
Paper, hasId, PaperID(Integer)

Objectchunk name: SubmissionInfo
Objectchunk in: *p Paper
```

B.2.1 Django Representation

```
def SubmissionInfo(request, paperID):
   # get the chunk's in-context
   paper = get_object_or_404(
                        Paper,
                        pk=paperID)
   return render_response(
                   request,
                   "submissioninfo.html",
                   {'paper': paper})
```

B.2.2 Ruby on Rails Representation

```
class PapersController < ApplicationController
    def submissioninfo
        @paper = Paper.find(params[:paper_id])
end
```

B.2.3 CakePHP Representation

```
function submissioninfo($paperID)
    {
      $this->Paper->id = $paperID;
      $this->set('paper', $this->Paper->read();
    }
```

Bibliography

[1] First international workshop on web-oriented software technology.
URL: http://users.dsic.upv.es/~west/iwwost01/.

[2] Foafcorp rdf vocabulary.
URL: http://xmlns.com/foaf/corp/.

[3] The hera research program.
URL: http://wwwis.win.tue.nl/~hera/.

[4] Hyperde.
URL: http://www.tecweb.inf.puc-rio.br/hyperde/.

[5] Omg unified modeling language specification. Technical report.
URL: http://www.rational.com/uml/.

[6] Politecnico di milano.
URL: http://www.polimi.it/.

[7] Ruby on rails web framework.
URL: http://www.rubyonrails.com.

[8] Ruby programming language.
URL: http://www.ruby-lang.org/en/.

[9] T. Berners-Lee. Semantic Web-XML2000 http://www. w3. org/2000. *Talks/1206-xml2k-tbl/slide10-0. html*.

[10] T. Berners-Lee and M. Fischetti. *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*. Harper San Francisco, 1999.

[11] B. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

[12] M. Brambilla, S. Comai, P. Fraternali, and M. Matera. Designing web applications with WebML and WebRatio. *Web Engineering: Modelling and Implementing Web Applications*, pages 221–260, 2007.

[13] J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, page 197, 2003.

[14] C. Cachero, J. Gómez, A. Párraga, and O. Pastor. Conference review system: A case of study. In *First Int. Workshop on Web-Oriented Software Technology*, 2001.

[15] S. Casteleyn, O. De Troyer, and S. Brockmans. Design time support for adaptive behavior in Web sites. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 1222–1228. ACM New York, NY, USA, 2003.

[16] S. Casteleyn, P. Plessers, and O. De Troyer. On Generating Content and Structural Annotated Websites Using Conceptual Modeling. *Lecture Notes in Computer Science*, 4215:267, 2006.

[17] O. De Troyer and S. Casteleyn. The conference review system with WSDM. In *First International Workshop on Web-Oriented Software Technology*, volume 5, 2001.

[18] O. De Troyer and S. Casteleyn. Modeling complex processes for Web applications using WSDM. In *Proceedings of the 3rd International Workshop on Web-Oriented Software Technologies*, pages 27–50, 2003.

[19] O. De Troyer and S. Casteleyn. Designing localized web sites. *Lecture notes in computer science*, pages 547–558, 2004.

[20] O. De Troyer, S. Casteleyn, and P. Plessers. Using ORM to model web systems. *Lecture notes in computer science*, 3762:700, 2005.

[21] O. De Troyer, S. Casteleyn, and P. Plessers. WSDM: Web Semantics Design Method. *Web Engineering: Modelling and Implementing Web Applications*, 2007.

[22] O. De Troyer and CJ. Leune. Wsdm: a user centered design method for web sites. *Computer Networks and ISDN Systems*, 30(1-7):85–94, 1998.

[23] O. De Troyer, P. Plessers, and S. Casteleyn. Conceptual view integration for audience driven web design. In *CD-ROM Proceedings of the WWW2003 Conference, Budapest, Hongary*, 2003.

[24] O. De Troyer, P. Plessers, and S. Casteleyn. Solving Semantic Conflicts in Adience Driven Web Design. In *Proceedings of the WWW/Internet 2003 Conference, Algarve Portugal*, 2003.

[25] M. Dean and G. Schreiber. Web ontology language. Technical report, Web Ontology Working Group,, World Wide Web Consortium, 2004.

[26] C. Fillies, F. Weichhardt, and B. Smith. Semantically correct Visio Drawings. In *Proceedings of the Workshop on User Aspects of the Semantic Web (UserSWeb2005)*, pages 85–92, 2005.

[27] J. Flynn. Visioowl.
URL: http://mysite.verizon.net/jflynn12/VisioOWL/VisioOWL.htm.

[28] J. Gómez, C. Cachero, and O. Pastor. On Conceptual Modeling of Device-Independent Web Applications: Towards a Web-Engineering Approach. *IEEE Multimedia*, 8(2):26–39, 2001.

[29] T. Halpin. *Information modeling and relational databases: from conceptual analysis to logical design*. Morgan Kaufmann, 2001.

[30] Jason Kaplan-Moss. Django: Web development for perfectionists with deadlines., 2001.
URL: http://video.google.com/videoplay?docid=-70449010942275062.

[31] A. Knapp, N. Koch, F. Moser, and G. Zhang. ArgoUWE: A CASE tool for Web applications. In *Proceedings of the 1st International Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE 03)*, 2003.

[32] A. Kraus, A. Knapp, and N. Koch. Model-driven generation of web applications in UWE. In *Proceedings of the International Workshop on Model-Driven Web Engineering, Como, Italy*, 2007.

[33] C. Mesnage and E. Oren. Extending ruby on rails for semantic web applications. *Lecture Notes in Computer Science*, 4607:506, 2007.

[34] Lasse Motroen. Developing web applications using oohdm and ror. 2006.

[35] Demetrius A. Nunes and Daniel Schwabe. Rapid prototyping of web applications combining domain specific languages and model driven design. In *ICWE '06: Proceedings of the 6th international conference on Web engineering*, pages 153–160, New York, NY, USA, 2006. ACM.
ISBN: 1-59593-352-2.

[36] O. Pastor, J. Fons, and V. Pelechano. Oows: A method to develop web applications from web-oriented conceptual models. In *International Workshop on Web Oriented Software Technology (IWWOST)*, pages 65–70, 2003.

[37] F. Paterno. Model-based design of interactive applications. *intelligence*, 11(4):26–38, 2000.

[38] P. Plessers, S. Casteleyn, Y. Yesilada, O. De Troyer, R. Stevens, S. Harper, and C. Goble. Accessibility: a web engineering approach. In *Proceedings of the 14th international conference on World Wide Web*, pages 353–362. ACM New York, NY, USA, 2005.

[39] W. Pree. Framework development and reuse support. *Visual Object-Oriented Programming, Concepts and Environments. M. Burnett, A. Goldberg, T. Lewis (eds.). Manning-Prentice Hall*, 1995.

[40] Gustavo Rossi and Daniel Schwabe. Object-oriented design structures in web application models. *Ann. Softw. Eng.*, 13(1-4):97–110, 2002.
ISSN: 1022-7091.

[41] D. Schwabe. A conference review system. 2001.

[42] D. Schwabe and G. Rossi. A Conference Review System with OOHDM. In *First International Workshop on Web-Oriented Software Technology*, volume 5, 2001.

[43] Daniel Schwabe and Gustavo Rossi. An object oriented approach to web-based applications design. *Theor. Pract. Object Syst.*, 4(4):207–225, 1998.
ISSN: 1074-3227.

[44] B. Tate and C. Hibbs. *Ruby on Rails: Up and Running*. O'Reilly Media, Inc., 2006.

List of Figures